

wolfSSL Documentation



2026-07-04

Contents

1 序章	18
1.1 なぜ wolfSSL を選ぶのですか？	18
2 wolfSSL のビルド	19
2.1 wolfSSL ソースコードの取得	19
2.2 *nix 上でビルド	19
2.3 Windows の上でのビルド	20
2.3.1 Visual Studio 2008	20
2.3.2 Visual Studio 2010	20
2.3.3 Visual Studio 2013 (64 ビットソリューション)	20
2.3.4 Cygwin	20
2.4 各社 IDE によるビルド	21
2.4.1 1. wolfSSL ライブラリー生成プロジェクト	21
2.4.2 2. アプリケーション実行ファイル生成プロジェクト	22
2.5 非標準環境でのビルド	22
2.5.1 Yocto Linux へのビルド	22
2.5.2 Atollic TrueSTUDIO でのビルド	23
2.5.3 IAR でのビルド	23
2.5.4 macOS や iOS でのビルド	24
2.5.5 GCC ARM でのビルド	25
2.5.6 Keil MDK-ARM でのビルド	25
2.6 C プリプロセッサマクロとして定義される機能	26
2.6.1 機能の削除	26
2.6.2 機能の有効化 (デフォルトで有効)	30
2.6.3 機能の有効化 (デフォルトで無効)	30
2.6.4 wolfSSL のカスタマイズ及び移植	45
2.6.5 メモリまたはコードの使用量の削減	48
2.6.6 パフォーマンスを向上させる	50
2.6.7 GCM パフォーマンスチューニング	50
2.6.8 wolfSSL の数学ライブラリオプション	50
2.6.9 スタックまたはチップ固有の定義	55
2.6.10 OS 特有の定義	58
2.7 ビルドオプション	58
2.7.1 --enable-debug	58
2.7.2 --enable-distro	59
2.7.3 --enable-singlethread	59
2.7.4 --enable-dtls	59
2.7.5 --disable-rng	59
2.7.6 --enable-sctp	59
2.7.7 --enable-openssh	59
2.7.8 --enable-apachehttpd	59
2.7.9 --enable-openvpn	59
2.7.10 --enable-opensslextra	59
2.7.11 --enable-opensslall	59
2.7.12 --enable-maxstrength	60
2.7.13 --disable-harden	60
2.7.14 --enable-ipv6	60
2.7.15 --enable-bump	60
2.7.16 --enable-leanpsk	60
2.7.17 --enable-leantls	60
2.7.18 --enable-bigcache	60

2.7.19 --enable-hugecache	61
2.7.20 --enable-smallcache	61
2.7.21 --enable-savesession	61
2.7.22 --enable-savecert	61
2.7.23 --enable-atomicuser	61
2.7.24 --enable-pkcallbacks	61
2.7.25 --enable-sniffer	61
2.7.26 --enable-aesgcm	62
2.7.27 --enable-aesccm	62
2.7.28 --disable-aescbc	62
2.7.29 --enable-aescfb	62
2.7.30 --enable-aesctr	62
2.7.31 --enable-aesni	62
2.7.32 --enable-intelasm	62
2.7.33 --enable-camellia	63
2.7.34 --enable-md2	63
2.7.35 --enable-nullcipher	63
2.7.36 --enable-ripemd	63
2.7.37 --enable-blake2	63
2.7.38 --enable-blake2s	63
2.7.39 --enable-sha3	63
2.7.40 --enable-sha512	63
2.7.41 --enable-sessioncerts	63
2.7.42 --enable-keygen	63
2.7.43 --enable-certgen	63
2.7.44 --enable-cert	63
2.7.45 --enable-certreq	63
2.7.46 --enable-sep	64
2.7.47 --enable-hkdf	64
2.7.48 --enable-x963kdf	64
2.7.49 --enable-dsa	64
2.7.50 --enable-eccshamir	64
2.7.51 --enable-ecc	64
2.7.52 --enable-ecccustcurves	64
2.7.53 --enable-compkey	64
2.7.54 --enable-curve25519	64
2.7.55 --enable-ed25519	64
2.7.56 --enable-fpecc	65
2.7.57 --enable-eccencrypt	65
2.7.58 --enable-psk	65
2.7.59 --disable-errorstrings	65
2.7.60 --disable-oldtls	65
2.7.61 --enable-sslv3	65
2.7.62 --enable-stacksize	65
2.7.63 --disable-memory	65
2.7.64 --disable-rsa	65
2.7.65 --enable-rsapss	65
2.7.66 --disable-dh	65
2.7.67 --enable-anon	65
2.7.68 --disable-asn	65
2.7.69 --disable-aes	66
2.7.70 --disable-coding	66
2.7.71 --enable-base64encode	66
2.7.72 --disable-des3	66

2.7.73 --enable-arc4	66
2.7.74 --disable-md5	66
2.7.75 --disable-sha	66
2.7.76 --enable-webserver	66
2.7.77 --enable-fips	66
2.7.78 --enable-sha224	66
2.7.79 --disable-poly1305	66
2.7.80 --disable-chacha	66
2.7.81 --disable-hashdrbg	67
2.7.82 --disable-filesystem	67
2.7.83 --disable-inline	67
2.7.84 --enable-ocsp	67
2.7.85 --enable-ocspstapling	67
2.7.86 --enable-ocspstapling2	67
2.7.87 --enable-crl	67
2.7.88 --enable-crl-monitor	67
2.7.89 --enable-sni	67
2.7.90 --enable-maxfragment	67
2.7.91 --enable-alpn	68
2.7.92 --enable-truncatedhmac	68
2.7.93 --enable-renegotiation-indication	68
2.7.94 --enable-secure-renegotiation	68
2.7.95 --enable-supportedcurves	68
2.7.96 --enable-session-ticket	68
2.7.97 --enable-extended-master	68
2.7.98 --enable-tlsx	68
2.7.99 --enable-pkcs7	68
2.7.100 --enable-pkcs11	68
2.7.101 --enable-ssh	68
2.7.102 --enable-scep	69
2.7.103 --enable-srp	69
2.7.104 --enable-smallstack	69
2.7.105 --enable-valgrind	69
2.7.106 --enable-testcert	69
2.7.107 --enable-iopool	69
2.7.108 --enable-certservice	69
2.7.109 --enable-jni	69
2.7.110 --enable-lighty	69
2.7.111 --enable-stunnel	69
2.7.112 --enable-md4	69
2.7.113 --enable-pwdbased	70
2.7.114 --enable-scrypt	70
2.7.115 --enable-cryptonly	70
2.7.116 --disable-examples	70
2.7.117 --disable-crypttests	70
2.7.118 --enable-fast-rsa	70
2.7.119 --enable-staticmemory	70
2.7.120 --enable-mcapi	70
2.7.121 --enable-asynccrypt	70
2.7.122 --enable-sessionexport	70
2.7.123 --enable-aeskeywrap	71
2.7.124 --enable-jobserver	71
2.7.125 --enable-shared[=PKGS]	71
2.7.126 --enable-static[=PKGS]	71

2.7.127--with-liboqs=PATH	71
2.7.128--with-libz=PATH	71
2.7.129--with-cavium	71
2.7.130--with-user-crypto	71
2.7.131--enable-rsavyf	71
2.7.132--enable-rsapub	72
2.7.133--enable-armasm	72
2.7.134--disable-tls12	72
2.7.135--enable-tls13	72
2.7.136--enable-all	72
2.7.137--enable-xts	72
2.7.138--enable-asio	72
2.7.139--enable-qt	72
2.7.140--enable-qt-test	72
2.7.141--enable-apache-httpd	72
2.7.142--enable-afalg	73
2.7.143--enable-devcrypto	73
2.7.144--enable-mcast	73
2.7.145--disable-pkcs12	73
2.7.146--enable-fallback-scsv	73
2.7.147--enable-psk-one-id	73
2.7.148--enable-cryptocb	73
2.7.149--enable-reproducible-build	73
2.7.150--enable-sys-ca-certs	74
2.8 特別な数学最適化フラグ	74
2.8.1 --enable-fastmath	74
2.8.2 --enable-fasthugemath	74
2.8.3 --enable-sp-math	74
2.8.4 --enable-sp-math-all	74
2.8.5 --enable-sp-asm	75
2.8.6 --enable-sp=OPT	75
2.9 クロスコンパイル	78
2.9.1 サンプルツールチェーンを使ったクロスコンパイルのコンフィグオプション例	79
2.10 移植向けビルド	81
2.11 NXP CAAM 向けのビルド	82
2.11.1 i.MX8(Linux)	82
2.11.2 i.MX8 (QNX)	92
2.11.3 i.MX6 (QNX)	92
2.11.4 IMXRT1170 (FreeRTOS)	92
3 入門	94
3.1 概要	94
3.2 テストスイート	94
3.3 クライアントサンプルプログラム	96
3.4 サーバーサンプルプログラム	98
3.5 エコーサーバーのサンプルプログラム	100
3.6 エコークライアントのサンプルプログラム	100
3.7 ベンチマーク	101
3.7.1 相対パフォーマンス	103
3.7.2 ベンチマークに関する補足	103
3.7.3 組み込みシステムでのベンチマーク	104
3.8 クライアントアプリケーションで wolfSSL を使用する	105
3.9 サーバーアプリケーションで wolfSSL を使用する	106

4 機能	108
4.1 機能概要	108
4.2 プロトコルサポート	108
4.2.1 サーバー関数	108
4.2.2 クライアント関数	108
4.2.3 堅牢なクライアントとサーバーのダウングレード	109
4.2.4 IPv6 サポート	109
4.2.5 DTLS	109
4.2.6 LwIP (軽量インターネットプロトコル)	109
4.2.7 TLS 拡張機能	110
4.3 暗号サポート	110
4.3.1 暗号スイートの強度と適切な鍵サイズの選択	110
4.3.2 サポートしている暗号スイート	112
4.3.3 AEAD スイート	114
4.3.4 ブロック暗号とストリーム暗号	115
4.3.5 ハッシュ関数	115
4.3.6 公開鍵オプション	115
4.3.7 ECC サポート	116
4.3.8 PKCS サポート	116
4.3.9 特定の暗号スイートの使用を強制する	118
4.3.10 OpenQuantumSafe の liboqs 統合	118
4.4 ハードウェアによる支援機構の活用	118
4.4.1 AES-NI	118
4.4.2 STM32F2	119
4.4.3 Cavium NITROX	119
4.4.4 ESP32-WROOM-32	119
4.4.5 ESP8266	119
4.4.6 EFR32	120
4.4.7 MAX32665/MAX32666	120
4.5 SSL 検査 (スニファー)	120
4.6 静的バッファ確保オプション	121
4.6.1 静的バッファ確保の基本操作	121
4.6.2 静的バッファ使用の指定	122
4.6.3 静的バッファ確保の有効化	122
4.6.4 静的バッファ確保の使用	124
4.6.5 TLS なしでの静的メモリの使用 (暗号のみ)	125
4.6.6 静的バッファ確保の調整	128
4.6.7 静的バッファ利用状況のトラッキング	129
4.6.8 静的バッファ管理 API	130
4.7 圧縮	132
4.8 事前共有鍵	132
4.9 クライアント認証	133
4.10 Server Name Indication	133
4.11 ハンドシェイクの変更	134
4.11.1 ハンドシェイクメッセージのグループ化	134
4.12 短縮 HMAC	134
4.13 wolfSSL のタイミング耐性	134
4.14 固定 ABI	135
5 移植性	137
5.1 抽象化レイヤー	137
5.1.1 C 標準ライブラリ抽象化レイヤー	137
5.1.2 カスタム入出力抽象化レイヤー	138
5.1.3 オペレーティングシステム抽象化レイヤー	138

5.2	サポートしているオペレーティングシステム	138
5.3	サポートしているチップメーカー	139
5.4	C# ラッパー	139
6	コールバック	141
6.1	ハンドシェイクコールバック	141
6.2	タイムアウトコールバック	141
6.3	ユーザーアトミックレコードレイヤー処理	142
6.4	公開鍵コールバック	143
6.4.1	DH コールバック	143
6.4.2	Ed25519 コールバック	143
6.4.3	X25519 コールバック	144
6.4.4	Ed448 コールバック	145
6.4.5	X448 コールバック	145
6.4.6	RSA PSS コールバック	146
6.4.7	ECC コールバック	147
6.5	暗号コールバック (cryptocb)	149
6.5.1	暗号コールバックの使用方法	149
6.5.2	暗号コールバックの作成	150
6.5.3	リクエストの処理	152
6.5.4	トラブルシューティング	152
6.5.5	実装例	152
7	鍵と証明書	153
7.1	サポートしているフォーマットとサイズ	153
7.2	サポートしている証明書拡張	153
7.3	証明書の読み込み	155
7.3.1	CA 証明書の読み込み	155
7.3.2	クライアントまたはサーバー証明書の読み込み	155
7.3.3	秘密鍵の読み込み	155
7.3.4	信頼されるピア証明書の読み込み	156
7.4	証明書チェーンの検証	156
7.5	サーバー証明書のドメイン名チェック	156
7.6	ファイルシステムなしでの証明書の使用	157
7.6.1	テスト証明書と鍵バッファ	157
7.7	シリアル番号の取得	157
7.8	RSA 鍵生成	157
7.8.1	RSA 鍵生成に関する注意	158
7.9	証明書生成	159
7.10	証明書署名要求 (CSR) の生成	161
7.10.1	制限事項	162
7.11	生の ECC 鍵に変換	162
7.11.1	実装例	163
8	デバッグ	164
8.1	デバッグとロギング	164
8.2	エラーコード	164
9	ライブラリデザイン	165
9.1	ライブラリヘッダー	165
9.2	起動と終了	165
9.3	構造体の使用	165
9.4	スレッドセーフ	165
9.5	入力と出力バッファ	166

10 wolfCrypt の使用法	167
10.1 ハッシュ関数	167
10.1.1 MD4	167
10.1.2 MD5	167
10.1.3 SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512	168
10.1.4 BLAKE2b	168
10.1.5 RIPEMD-160	169
10.2 鍵付きハッシュ関数	169
10.2.1 HMAC	169
10.2.2 GMAC	169
10.2.3 Poly1305	170
10.3 ブロック暗号	170
10.3.1 AES	170
10.4 ストリーム暗号	171
10.4.1 ARC4	171
10.4.2 ChaCha	172
10.5 公開鍵暗号	173
10.5.1 RSA	173
10.5.2 DH (Diffie-Hellman)	174
10.5.3 EDH (Ephemeral Diffie-Hellman)	174
10.5.4 DSA (デジタル署名アルゴリズム)	175
11 SSL/TLS チュートリアル	176
11.1 はじめに	176
11.1.1 このチュートリアルで使用される例	176
11.2 SSL/TLS の簡単な概要	176
11.3 ソースコードの入手	176
11.4 基本例の修正	177
11.4.1 echoserver (tcpserv04.c) への修正	177
11.4.2 echoclient (tcpcli01.c) への修正	177
11.4.3 unp.h ヘッダーへの修正	177
11.5 wolfSSL のビルドとインストール	177
11.6 初回コンパイル	179
11.7 ライブラリ	180
11.8 ヘッダー	180
11.9 起動/シャットダウン	180
11.10 WOLFSSL_CTX ファクトリ	181
11.11 WOLFSSL オブジェクト	183
11.11.1 EchoClient	183
11.11.2 EchoServer	183
11.12 データの送受信	184
11.12.1 EchoClient での送信	184
11.12.2 EchoServer での受信	185
11.13 シグナル処理	185
11.13.1 Echoclient / Echoserver	185
11.14 証明書	186
11.15 まとめ	187
12 組み込みデバイスのベストプラクティス	188
12.1 秘密鍵の作成	188
12.2 wolfSSL によるデジタル署名と認証	188
13 OpenSSL 互換性	189
13.1 OpenSSL との互換性	189
13.2 wolfSSL と OpenSSL の違い	189

13.3 サポートしている OpenSSL 構造体	190
13.4 サポートしている OpenSSL 関数	190
13.5 x509 証明書	191
14 ライセンス	192
14.1 オープンソース	192
14.2 商用ライセンス	192
14.3 FIPS 140-2/3 検証	192
14.4 サポートパッケージ	192
15 サポートとコンサルティング	193
15.1 サポートの受け方	193
15.1.1 バグの報告とサポート	193
15.2 コンサルティング	193
15.2.1 機能追加と移植	193
15.2.2 デザインコンサルティング	193
16 wolfSSL のアップデート	194
16.1 製品のリリース情報	194
A wolfSSL API リファレンス	195
A.1 証明書マネージャー API	195
A.1.1 Functions	195
A.1.2 Functions Documentation	197
A.2 メモリ処理	213
A.2.1 Functions	213
A.2.2 Functions Documentation	219
A.3 OpenSSL API	234
A.3.1 Functions	234
A.3.2 Functions Documentation	238
A.4 wolfSSL 証明書と鍵	258
A.4.1 Functions	258
A.4.2 Functions Documentation	269
A.5 wolfSSL 接続、セッション、I/O	326
A.5.1 Functions	326
A.5.2 Functions Documentation	340
A.6 wolfSSL コンテキストとセッションのセットアップ	387
A.6.1 Functions	387
A.6.2 Functions Documentation	407
A.7 wolfSSL エラー処理と報告	485
A.7.1 Functions	485
A.7.2 Functions Documentation	487
A.8 wolfSSL の初期化/シャットダウン	493
A.8.1 Functions	493
A.8.2 Functions Documentation	496
B wolfCrypt API リファレンス	511
B.1 ASN.1	511
B.1.1 Functions	511
B.1.2 Functions Documentation	518
B.2 ベースエンコーディング	560
B.2.1 Functions	560
B.2.2 Functions Documentation	561
B.3 圧縮	565
B.3.1 Functions	565

B.3.2	Functions Documentation	565
B.4	エラー報告	567
B.4.1	Functions	567
B.4.2	Functions Documentation	567
B.5	IoT-Safe モジュール	568
B.5.1	Functions	568
B.5.2	Detailed Description	570
B.5.3	Functions Documentation	570
B.6	ロギング	580
B.6.1	Functions	580
B.6.2	Functions Documentation	580
B.7	整数演算 API	581
B.7.1	Functions	581
B.7.2	Functions Documentation	581
B.8	乱数生成	582
B.8.1	Functions	582
B.8.2	Functions Documentation	583
B.9	署名 API	589
B.9.1	Functions	589
B.9.2	Functions Documentation	590
B.10	wolfCrypt の初期化とクリーンアップ	592
B.10.1	Functions	592
B.10.2	Functions Documentation	593
B.11	アルゴリズム - 3DES	595
B.11.1	Functions	595
B.11.2	Functions Documentation	597
B.12	アルゴリズム - AES	607
B.12.1	Functions	607
B.12.2	Functions Documentation	614
B.13	アルゴリズム - ARC4	653
B.13.1	Functions	653
B.13.2	Functions Documentation	653
B.14	アルゴリズム - BLAKE2	654
B.14.1	Functions	654
B.14.2	Functions Documentation	655
B.15	アルゴリズム - Camellia	657
B.15.1	Functions	657
B.15.2	Functions Documentation	657
B.16	アルゴリズム - ChaCha	661
B.16.1	Functions	661
B.16.2	Functions Documentation	661
B.17	アルゴリズム - ChaCha20_Poly1305	663
B.17.1	Functions	663
B.17.2	Functions Documentation	664
B.18	Callbacks コールバック - CryptoCb	666
B.19	アルゴリズム - Curve25519	666
B.19.1	Functions	666
B.19.2	Functions Documentation	668
B.20	アルゴリズム - Curve448	682
B.20.1	Functions	682
B.20.2	Functions Documentation	684
B.21	アルゴリズム - DSA	698
B.21.1	Functions	698
B.21.2	Functions Documentation	699

B.22	アルゴリズム - Diffie-Hellman	705
B.22.1	Functions	705
B.22.2	Functions Documentation	706
B.23	アルゴリズム - ECC	714
B.23.1	Functions	714
B.23.2	Functions Documentation	719
B.24	アルゴリズム - ED25519	755
B.24.1	Functions	755
B.24.2	Functions Documentation	758
B.25	アルゴリズム - ED448	778
B.25.1	Functions	778
B.25.2	Functions Documentation	781
B.26	プラットフォームセキュリティアーキテクチャ (PSA)API	799
B.26.1	Functions	799
B.26.2	Functions Documentation	799
B.27	アルゴリズム - SipHash	801
B.27.1	Functions	801
B.27.2	Functions Documentation	801
C	API ヘッダーファイル	805
C.1	dox_comments/header_files-ja/aes.h	805
C.1.1	Functions	805
C.1.2	Functions Documentation	812
C.1.3	Source code	849
C.2	dox_comments/header_files-ja/arc4.h	853
C.2.1	Functions	853
C.2.2	Functions Documentation	853
C.2.3	Source code	854
C.3	dox_comments/header_files-ja/asn.h	854
C.4	dox_comments/header_files-ja/asn_public.h	854
C.4.1	Functions	854
C.4.2	Functions Documentation	862
C.4.3	Source code	907
C.5	dox_comments/header_files-ja/blake2.h	910
C.5.1	Functions	910
C.5.2	Functions Documentation	911
C.5.3	Source code	912
C.6	dox_comments/header_files-ja/bn.h	913
C.6.1	Functions	913
C.6.2	Functions Documentation	913
C.6.3	Source code	913
C.7	dox_comments/header_files-ja/camellia.h	914
C.7.1	Functions	914
C.7.2	Functions Documentation	914
C.7.3	Source code	918
C.8	dox_comments/header_files-ja/chacha20_poly1305.h	918
C.8.1	Functions	918
C.8.2	Functions Documentation	919
C.8.3	Source code	921
C.9	dox_comments/header_files-ja/chacha.h	921
C.9.1	Functions	921
C.9.2	Functions Documentation	922
C.9.3	Source code	924
C.10	dox_comments/header_files-ja/cmac.h	924

C.10.1 Functions	924
C.10.2 Functions Documentation	925
C.10.3 Source code	930
C.11 dox_comments/header_files-ja/coding.h	930
C.11.1 Functions	930
C.11.2 Functions Documentation	931
C.11.3 Source code	935
C.12 dox_comments/header_files-ja/compress.h	936
C.12.1 Functions	936
C.12.2 Functions Documentation	936
C.12.3 Source code	937
C.13 dox_comments/header_files-ja/cryptocb.h	937
C.13.1 Functions	937
C.13.2 Functions Documentation	938
C.13.3 Source code	940
C.14 dox_comments/header_files-ja/curve25519.h	940
C.14.1 Functions	940
C.14.2 Functions Documentation	942
C.14.3 Source code	956
C.15 dox_comments/header_files-ja/curve448.h	957
C.15.1 Functions	957
C.15.2 Functions Documentation	959
C.15.3 Source code	973
C.16 dox_comments/header_files-ja/des3.h	974
C.16.1 Functions	974
C.16.2 Functions Documentation	976
C.16.3 Source code	982
C.17 dox_comments/header_files-ja/dh.h	982
C.17.1 Functions	982
C.17.2 Functions Documentation	984
C.17.3 Source code	992
C.18 dox_comments/header_files-ja/doxygen_groups.h	993
C.19 dox_comments/header_files-ja/doxygen_pages.h	993
C.20 dox_comments/header_files-ja/dsa.h	993
C.20.1 Functions	993
C.20.2 Functions Documentation	994
C.20.3 Source code	1000
C.21 dox_comments/header_files-ja/ecc.h	1000
C.21.1 Functions	1000
C.21.2 Functions Documentation	1006
C.21.3 Source code	1041
C.22 dox_comments/header_files-ja/eccsi.h	1043
C.22.1 Functions	1043
C.22.2 Functions Documentation	1045
C.22.3 Source code	1049
C.23 dox_comments/header_files-ja/ed25519.h	1050
C.23.1 Functions	1050
C.23.2 Functions Documentation	1054
C.23.3 Source code	1073
C.24 dox_comments/header_files-ja/ed448.h	1075
C.24.1 Functions	1075
C.24.2 Functions Documentation	1078
C.24.3 Source code	1095
C.25 dox_comments/header_files-ja/error-crypt.h	1097

C.25.1 Functions	1097
C.25.2 Functions Documentation	1097
C.25.3 Source code	1098
C.26 dox_comments/header_files-ja/evp.h	1098
C.26.1 Functions	1098
C.26.2 Functions Documentation	1100
C.26.3 Source code	1108
C.27 dox_comments/header_files-ja/hash.h	1109
C.27.1 Functions	1109
C.27.2 Functions Documentation	1110
C.27.3 Source code	1118
C.28 dox_comments/header_files-ja/hmac.h	1119
C.28.1 Functions	1119
C.28.2 Functions Documentation	1121
C.28.3 Source code	1133
C.29 dox_comments/header_files-ja/iotsafe.h	1134
C.29.1 Functions	1134
C.29.2 Functions Documentation	1136
C.29.3 Source code	1146
C.30 dox_comments/header_files-ja/logging.h	1147
C.30.1 Functions	1147
C.30.2 Functions Documentation	1147
C.30.3 Source code	1149
C.31 dox_comments/header_files-ja/md2.h	1149
C.31.1 Functions	1149
C.31.2 Functions Documentation	1149
C.31.3 Source code	1152
C.32 dox_comments/header_files-ja/md4.h	1152
C.32.1 Functions	1152
C.32.2 Functions Documentation	1152
C.32.3 Source code	1154
C.33 dox_comments/header_files-ja/md5.h	1154
C.33.1 Functions	1154
C.33.2 Functions Documentation	1155
C.33.3 Source code	1158
C.34 dox_comments/header_files-ja/memory.h	1158
C.34.1 Functions	1158
C.34.2 Functions Documentation	1161
C.34.3 Source code	1173
C.35 dox_comments/header_files-ja/pem.h	1174
C.35.1 Functions	1174
C.35.2 Functions Documentation	1174
C.35.3 Source code	1175
C.36 dox_comments/header_files-ja/pkcs11.h	1175
C.36.1 Functions	1175
C.36.2 Functions Documentation	1176
C.36.3 Source code	1177
C.37 dox_comments/header_files-ja/pkcs7.h	1177
C.37.1 Types	1177
C.37.2 Functions	1177
C.37.3 Types Documentation	1180
C.37.4 Functions Documentation	1180
C.37.5 Source code	1195
C.38 dox_comments/header_files-ja/poly1305.h	1196

C.38.1 Functions	1196
C.38.2 Functions Documentation	1196
C.38.3 Source code	1199
C.39 dox_comments/header_files-ja/psa.h	1200
C.39.1 Functions	1200
C.39.2 Functions Documentation	1200
C.39.3 Source code	1202
C.40 dox_comments/header_files-ja/pwdbased.h	1202
C.40.1 Functions	1202
C.40.2 Functions Documentation	1203
C.40.3 Source code	1206
C.41 dox_comments/header_files-ja/quic.h	1206
C.41.1 Functions	1206
C.41.2 Attributes	1209
C.41.3 Functions Documentation	1210
C.41.4 Attributes Documentation	1221
C.41.5 Source code	1222
C.42 dox_comments/header_files-ja/random.h	1224
C.42.1 Functions	1224
C.42.2 Attributes	1224
C.42.3 Functions Documentation	1225
C.42.4 Attributes Documentation	1231
C.42.5 Source code	1231
C.43 dox_comments/header_files-ja/ripemd.h	1231
C.43.1 Functions	1231
C.43.2 Functions Documentation	1232
C.43.3 Source code	1234
C.44 dox_comments/header_files-ja/rsa.h	1234
C.44.1 Functions	1234
C.44.2 Functions Documentation	1238
C.44.3 Source code	1269
C.45 dox_comments/header_files-ja/sakke.h	1271
C.45.1 Functions	1271
C.45.2 Functions Documentation	1273
C.45.3 Source code	1277
C.46 dox_comments/header_files-ja/sha256.h	1279
C.46.1 Functions	1279
C.46.2 Functions Documentation	1279
C.46.3 Source code	1284
C.47 dox_comments/header_files-ja/sha512.h	1284
C.47.1 Functions	1284
C.47.2 Functions Documentation	1285
C.47.3 Source code	1288
C.48 dox_comments/header_files-ja/sha.h	1289
C.48.1 Functions	1289
C.48.2 Functions Documentation	1289
C.48.3 Source code	1292
C.49 dox_comments/header_files-ja/signature.h	1292
C.49.1 Functions	1292
C.49.2 Functions Documentation	1293
C.49.3 Source code	1295
C.50 dox_comments/header_files-ja/siphash.h	1296
C.50.1 Functions	1296
C.50.2 Functions Documentation	1296

C.50.3 Source code	1299
C.51 dox_comments/header_files-ja/srp.h	1299
C.51.1 Functions	1299
C.51.2 Functions Documentation	1301
C.51.3 Source code	1310
C.52 dox_comments/header_files-ja/ssl.h	1311
C.52.1 Functions	1311
C.52.2 Functions Documentation	1383
C.52.3 Source code	1688
C.53 dox_comments/header_files-ja/tfm.h	1710
C.53.1 Functions	1710
C.53.2 Functions Documentation	1710
C.53.3 Source code	1711
C.54 dox_comments/header_files-ja/types.h	1711
C.54.1 Functions	1711
C.54.2 Functions Documentation	1713
C.54.3 Source code	1716
C.55 dox_comments/header_files-ja/wc_encrypt.h	1716
C.55.1 Functions	1716
C.55.2 Functions Documentation	1717
C.55.3 Source code	1721
C.56 dox_comments/header_files-ja/wc_port.h	1722
C.56.1 Functions	1722
C.56.2 Functions Documentation	1722
C.56.3 Source code	1723
C.57 dox_comments/header_files-ja/wolfio.h	1723
C.57.1 Functions	1723
C.57.2 Functions Documentation	1727
C.57.3 Source code	1739
D SSL/TLS の概要	1741
D.1 アーキテクチャ	1741
D.2 SSL ハンドシェイク	1741
D.3 SSL/TLS における、バージョンごとの違い	1741
D.3.1 SSL 3.0	1743
D.3.2 TLS 1.0	1743
D.3.3 TLS 1.1	1743
D.3.4 TLS 1.2	1743
D.3.5 TLS 1.3	1744
E RFC・仕様・リファレンス	1745
E.1 プロトコル	1745
E.2 ストリーム暗号	1745
E.3 ブロック暗号	1745
E.4 ハッシュ	1745
E.5 公開鍵暗号	1745
E.6 その他	1745
F エラーコード	1746
F.1 wolfSSL エラーコード	1746
F.2 wolfCrypt エラーコード	1749
F.3 一般的なエラーコードとその解決策	1752
F.3.1 ASN_NO_SIGNER_E (-188)	1752
F.3.2 WANT_READ (-323)	1752

G	wolfSSL 対応暗号スイート・鍵交換グループ・署名アルゴリズム一覧	1753
G.1	A. 暗号スイート	1753
G.1.1	A-1. TLS 1.3	1753
G.1.2	A-2. TLS 1.2 ECDHE-ECDSA	1753
G.1.3	A-3. TLS 1.2 ECDHE-RSA	1754
G.1.4	A-4. TLS 1.2 DHE-RSA	1754
G.1.5	A-5. TLS 1.2 RSA (静的鍵交換)	1754
G.1.6	A-6. TLS 1.2 静的 ECDH	1755
G.1.7	A-7. TLS 1.2 PSK	1755
G.1.8	A-8. TLS 1.2 DHE-PSK	1756
G.1.9	A-9. TLS 1.2 ECDHE-PSK	1756
G.1.10	A-10. TLS 1.2 DH_anon	1756
G.2	B. 鍵交換グループ	1756
G.2.1	B-1. NIST 楕円曲線	1756
G.2.2	B-2. Koblitz 曲線	1757
G.2.3	B-3. Brainpool 曲線	1757
G.2.4	B-4. X25519 / X448	1757
G.2.5	B-5. FFDHE	1757
G.2.6	B-6. SM2	1757
G.2.7	B-7. ML-KEM スタンドアロン	1758
G.2.8	B-8. ML-KEM ハイブリッド (標準)	1758
G.2.9	B-9. ML-KEM ハイブリッド (実験的)	1758
G.2.10	B-10. Kyber Round 3 レガシー	1758
G.3	C. 署名アルゴリズム	1759
G.3.1	C-1. ECDSA (NIST 曲線)	1759
G.3.2	C-2. ECDSA (Brainpool, TLS 1.3 専用)	1759
G.3.3	C-3. EdDSA	1759
G.3.4	C-4. RSA-PSS RSAE	1759
G.3.5	C-5. RSA-PSS PSS (TLS 1.3 専用)	1759
G.3.6	C-6. RSA PKCS#1 v1.5	1760
G.3.7	C-7. SM2	1760
G.3.8	C-8. ML-DSA / Dilithium	1760
G.4	D. configure オプション □ #define 対応表	1760
H	wolfSSL ハードウェア最適化サポート対応表	1762
H.1	A. アーキテクチャ別 ISA/SIMD 最適化	1762
H.2	B. SP 数学ライブラリ (公開鍵暗号)	1763
H.2.1	sp_int.c インラインアセンブリ高速化	1763
H.2.2	SP 鍵サイズ設定	1763
H.3	C. マイコン/SoC チップ別ハードウェアアクセラレータ	1763
H.4	D. 外部アクセラレータ・コプロセッサ	1765
H.5	E. OS カーネル経由ハードウェアオフロード	1765
I	ポスト量子暗号の実験	1766
I.1	ポスト量子暗号をわかりやすく紹介	1766
I.1.1	なぜポスト量子暗号?	1766
I.1.2	私たちは自分自身をどのように守るのですか?	1766
I.2	wolfSSL のでポスト量子暗号を始めましょう	1767
I.2.1	ビルド手順	1767
I.2.2	量子安全な TLS 接続を確立する	1768
I.3	暗号アーティファクトのサイズ	1768
I.3.1	ポスト量子署名スキームのアーティファクトサイズ	1768
I.3.2	ポスト量子 KEM のアーティファクトサイズ	1768
I.4	統計データ	1768

I.4.1	実行時バイナリサイズ	1769
I.4.2	TLS 1.3 データ送信サイズ	1769
I.4.3	ヒープとスタックの使用量	1769
I.4.4	ベンチマーク	1774
I.5	ドキュメンテーション	1782
I.6	ポスト量子ステートフルハッシュベース署名	1783
I.6.1	動機づけ	1783
I.6.2	LMS/HSS 署名	1783
I.6.3	XMSS/XMSS ^{MT} 署名	1786
I.6.4	開発者ノート	1791
J	wolfSSL 移植ガイド	1792
J.1	目的	1792
J.2	対象読者	1792
J.3	序章	1792
J.4	wolfSSL の移植	1792
J.4.1	データ型	1792
J.4.2	エンディアン	1793
J.4.3	writev	1794
J.4.4	ネットワーク I/O	1794
J.4.5	ファイルシステム	1794
J.4.6	スレッド化	1795
J.4.7	ランダムシード	1795
J.4.8	メモリー	1796
J.4.9	時間	1796
J.4.10	C 標準ライブラリ	1797
J.4.11	ロギング	1797
J.4.12	公開鍵操作	1797
J.4.13	アトミックレコードレイヤーの処理	1797
J.4.14	機能	1798
J.5	次のステップ	1798
J.5.1	wolfCrypt テストアプリケーション	1798
J.6	サポート	1798

1 序章

このマニュアルは、wolfSSL 組み込み SSL/TLS ライブラリの技術ガイドとして書かれており、wolfssl のビルド、実行方法、ビルドオプションの概要、提供機能、移植性の向上の方法、サポートなどに関する情報を提供します。

このドキュメントの PDF バージョンは[こちら](#)を参照下さい。

1.1 なぜ wolfSSL を選ぶのですか？

組み込み機器向けの SSL ソリューションとして wolfSSL を選択する理由はたくさんあります。最大の理由としてはそのサイズ (典型的なフットプリントサイズが 20~100 KB の範囲) と、サポートであるといえるでしょう。加えて、最新の標準 (SSL 3.0、TLS 1.0、TLS 1.1、TLS 1.2、TLS 1.3、DTLS 1.0、DTLS 1.2 および DTLS1.3) に準拠している点、先進的な暗号アルゴリズムのサポート (ストリーム暗号を含む)、マルチプラットフォーム、ロイヤリティフリー、および OpenSSL パッケージを使用した既存のアプリケーションの移植を容易にするための OpenSSL 互換性 API の提供などが理由として挙げられます。第 4 章にて、より詳しくご紹介しています。

2 wolfSSL のビルド

wolfSSL はポータビリティを念頭に置いて書かれており、一般的にほとんどのシステムで容易にビルドできます。wolfSSL をビルドするのが難しい場合は、サポートフォーラム (<https://www.wolfssl.com/forums>) を通じてサポートを求めるか、support@wolfssl.com に遠慮なく直接お問い合わせください。

この章では、Unix と Windows で wolfSSL をビルドする方法を説明し、非標準環境で wolfSSL をビルドする場合のガイダンスも提供します。第 3 章に入門ガイド、第 11 章に SSL チュートリアルをご用意しています。

autoconf/automake システムを使用して wolfSSL をビルドする場合、単一の MakeFile を使用して wolfSSL ライブラリの全てのパーツとサンプルプログラムを一度にビルドします。

2.1 wolfSSL ソースコードの取得

最新バージョンの wolfSSL は、wolfSSL の Web サイトから ZIP ファイルとしてダウンロードできます。

<https://wolfssl.jp/download/>

zip ファイルをダウンロードしたら、unzip コマンドを使用してファイルを解凍します。ネイティブの行末文字を使用するには、unzip を使用する際に -a オプションを有効にします。[unzip man page] で、-a オプションが説明されています。

[...] -a オプションでは、zip によってテキストファイル (ZipInfo リストで「b」ではなく「t」ラベルがついているファイル) と識別されると行末文字、ファイル終端文字を変換します。必要に応じて指定してください。[...]

注：wolfSSL 2.0.0RC3 のリリースから、wolfSSL のディレクトリ構造と標準のインストールフォルダーが変更されました。これらの変更は、オープンソースプロジェクトが wolfSSL を統合しやすくするために行われました。ヘッダーと構造の変更の詳細については、第 9 章の「ライブラリヘッダー」および「構造の使用」節を参照してください。

2.2 *nix 上でビルド

Linux、*BSD、OS X、Solaris、またはその他の *nix のようなシステムに wolfSSL をビルドする場合、autoconf システムを使用します。wolfSSL をビルドするには、wolfSSL ルートディレクトリから 2 つのコマンド ./configure および make を実行するだけです。

./configure スクリプトはビルド環境をセットアップし、任意の数のビルドオプションを ./configure に追加します。利用可能なビルドオプションのリストについては、この章のビルドオプション節を参照するか、次のコマンドラインを実行して、./configure に渡すことができるオプションのリストを表示させてください。

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

./configure が正常に実行されたら、wolfSSL をビルドするために make を実行してください。

```
make
```

wolfSSL をインストールするには以下を実行します。

```
make install
```

インストールするには特権を必要とする場合もあるかもしれません。その場合は次のように sudo をコマンドの前に加えてください。

```
sudo make install
```

ビルドをテストするには、wolfSSL のルートディレクトリから testsuite プログラムを実行します。

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

あるいは、autoconf を使用して testsuite を実行し、標準の wolfSSL API および暗号テストを実行します。

```
make test
```

testsuite プログラムの予想される出力に関する詳細は、第 3 章のテストスイート節にあります。

もし、wolfSSL ライブラリのみでのビルドが必要で、そのほかのコンポーネント（サンプルプログラム、テストスイート、ベンチマークアプリケーション等）はビルドしたくない場合には、wolfSSL のルートディレクトリで下記コマンドを実行してください、

```
make src/libwolfSSL.la
```

2.3 Windows の上でのビルド

以下の説明に加えて、Visual Studio で wolfSSL をビルドするための手順とヒントは[ここ](#)でも提供しています。

2.3.1 Visual Studio 2008

インストールのルートディレクトリに Visual Studio 2008 用のソリューションが含まれています。Visual Studio 2010 以降で使用する場合は、既存のプロジェクトファイルはインポートプロセス中に変換できるはずです。

注意: 新しいバージョンの Visual Studio にインポートする場合、「プロジェクトとインポートされたプロパティシートを上書きしますか?」と尋ねられます。「いいえ」を選択すると、以下の問題を回避できます。「はい」を選択すると、SAFESEH 指定のために EDITANDCONTINUE が無視されるという警告が表示されます。testsuite、sslSniffer、server、echoserver、echoclient、client をそれぞれ右クリックして、プロパティ-> 構成プロパティ-> リンカー->Advanced (Advanced ウィンドウの一番下までスクロール) を変更する必要があります。「安全な例外ハンドラーを含むイメージ」を見つけ、右端の下矢印をクリックします。これを前述の各プロジェクトに対して「いいえ」(/SAFESEH:NO)に変更します。もう一つの選択肢は EDITANDCONTINUE を無効にすることですが、デバッグ目的では有用であるため推奨されません。

2.3.2 Visual Studio 2010

wolfSSL ソリューションをビルドするには、Service Pack 1 をダウンロードする必要があります。Visual Studio がリンクエラーを報告する場合は、プロジェクトをクリーンしてリビルドすると解消するはずです。

2.3.3 Visual Studio 2013 (64 ビットソリューション)

wolfSSL ソリューションをビルドするには、Service Pack 4 をダウンロードする必要があります。Visual Studio がリンクエラーを報告する場合は、プロジェクトをクリーンしてリビルドすると解消するはずです。

各ビルドをテストするには、Visual Studio メニューから「Build All」を選択し、testsuite プログラムを実行します。Visual Studio プロジェクトのビルドオプションを編集するには、目的のプロジェクト (wolfSSL、echoclient、echoserver など) を選択し、「プロパティ」パネルを参照してください。

注意: wolfSSL v3.8.0 リリース以降、ビルドプリプロセッサマクロは IDE/WIN/user_settings.h にある中央ファイルに移動しました。このファイルはプロジェクト内でも見つけることができます。ECC や ChaCha20/Poly1305 などの機能を追加するには、HAVE_ECC や HAVE_CHACHA / HAVE_POLY1305 などの #defines をここに追加します。

2.3.4 Cygwin

Windows 開発マシン上で Windows 用の wolfSSL をビルドする場合は、リポジトリに含まれる Visual Studio プロジェクトファイルを使用してビルドすることをお勧めします。しかし Cygwin が必要な場合は、当社チームがビルドに成功した際のガイダンスがありますから以下にご紹介します。

1. <https://www.cygwin.com/install.html> にアクセスして `setup-x86_64.exe` をダウンロードします、
2. `setup-x86_64.exe` を実行し、希望する方法でインストールします。「Select Packages」段階に到達するまでインストールメニューをクリックします。
3. 「+」アイコンをクリックして「All」を展開します、
4. 「Archive」セクションに移動し、「unzip」ドロップダウンを選択して、「Skip」を 6.0-15（または他のバージョン）に変更します。
5. 「Devel」の下で「autoconf」ドロップダウンをクリックし、「Skip」を「10-1」（または他のバージョン）に変更します、
6. 「Devel」の下で「automake」ドロップダウンをクリックし、「Skip」を「10-1」（または他のバージョン）に変更します。
7. 「Devel」の下で「gcc-core」ドロップダウンをクリックし、「Skip」を 7.4.0-1 に変更します。（注：wolfSSL は GCC 9 または 10 をテストしておらず、比較的新しいものであるため、開発用に微調整する時間が多少経つまで使用することをお勧めしません）
8. 「Devel」の下で「git」ドロップダウンをクリックし、「Skip」を 2.29.0-1（または他のバージョン）に変更します。
9. 「Devel」の下で「libtool」ドロップダウンをクリックし、「Skip」を「2.4.6-5」（または他のバージョン）に変更します。
10. 「Devel」の下で「make」ドロップダウンをクリックし、「Skip」を 4.2.1-1（または他のバージョン）に変更します。
11. 「Next」をクリックして、インストールの残りの部分を進めます。

追加のパッケージとして、以下が必要です。

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

2.3.4.1 インストール後の作業 Cygwin ターミナルを開き、wolfSSL をクローンします。

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

2.4 各社 IDE によるビルド

wolfSSL サイトから wolfSSL ソースコード一式をダウンロード、解凍して以下の設定をします。

2.4.1 1. wolfSSL ライブラリー生成プロジェクト

- コンパイル対象のソースファイルの登録 `/src` の下の `.c` ファイルと `./wolfcrypt/src` の `.c` ファイル一式を登録します。ただし、以下のファイルを除きます。

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

- コンフィグレーションオプションの定義 ファイル名 `"user_settings.h"` のヘッダーファイルにコンフィグレーション定義を格納します。

wolfssl/examples/configs/README.md, user_settings_template.h を参照してください。

- プリディファインドマクロ名の登録 WOLFSSL_USER_SETTINGS を定義すると、ビルド時に上記の user_settings.h ファイルがインクルードされます。
- インクルードパスの登録 wolfSSL ソースファイルルートのパスと上記 user_settings.h のパスを登録します。

2.4.2 2. アプリケーション実行ファイル生成プロジェクト

- 1. で生成した wolfSSL ライブラリーをリンク対象として登録します。wolfSSL ソースファイルルートのパスと
- 1. で定義したコンフィグレーションオプション (user_settings.h) のパスを インクルードパスとして登録します。
- プリディファインドマクロ名として "WOLFSSL_USER_SETTINGS" を指定します。
- アプリケーションのプログラム先頭付近で "wolfssl/wolfcrypt/settings.h" をインクルードします。

2.5 非標準環境でのビルド

公式にはサポートしていませんが、特に組み込み向けやクロスコンパイルシステムを使用して、非標準環境で wolfSSL をビルドしたいユーザーを支援するよう努めています。以下にいくつかの注意点を示します。

1. ソースファイルとヘッダーファイルは、wolfSSL ダウンロードパッケージ内にあるのと同じディレクトリ構造を維持する必要があります。
2. 一部のビルドシステムでは、wolfSSL ヘッダーファイルの場所を明示的に知りたい場合があるため、それを指定する必要があるかもしれません。それらは <wolfSSL_root>/wolfSSL ディレクトリにあります。通常は、インクルードパスに <wolfSSL_root> ディレクトリを追加してください。
3. wolfSSL は、configure プロセスがビッグエンディアンを検出しない限り、デフォルトでリトルエンディアンシステムとなります。非標準環境でビルドするユーザーは configure プロセスを使用していないため、ビッグエンディアンシステムを使用している場合は BIG_ENDIAN_ORDER を定義する必要があります。
4. wolfSSL は 64 ビット型を利用できると速度が向上します。configure プロセスは long または long long が 64 ビットかどうかを判断し、define を設定します。したがって、システム上で sizeof(long) が 8 バイトの場合は、SIZEOF_LONG 8 を定義してください。sizeof(long long) が 8 バイトの場合は、SIZEOF_LONG_LONG 8 を定義します。
5. ライブラリのビルドで問題が発生した場合は、お知らせください。サポートが必要な場合は、support@wolfssl.com までご連絡ください。
6. ビルドを変更できる定義は、本章の後の節に示しています。多くのオプションの詳細な説明については、ビルドオプション節を参照してください。

2.5.1 Yocto Linux へのビルド

wolfSSL は Yocto Linux と OpenEmbedded で wolfSSL をビルドするためのレシピも含んでいます。これらのレシピは meta-wolfSSL レイヤーとして GitHub リポジトリで管理しています。

<https://github.com/wolfSSL/meta-wolfSSL>

Yocto Linux で wolfSSL をビルドするには Git と bitbake が必要です。以下に、Yocto Linux 上で wolfSSL 製品（レシピが存在するもの）をビルドする方法を説明します。

1. wolfSSL meta をクローン

```
git clone https://github.com/wolfSSL/meta-wolfSSL
```

2. 「meta-wolfSSL」 レイヤーをビルドの bblayers.conf に挿入

BBLAYERS セクション内で、meta-wolfSSL がクローン作成された場所へのパスを追加します。

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

3. wolfSSL 製品レシピをビルド

bitbake を使用して、wolfSSL、wolfssh、wolfmqtt をビルドします。これらのレシピのいずれかを bitbake コマンドに渡すだけです。(例: bitbake wolfSSL) これにより、ユーザーはコンパイルが問題なく成功することを確認できます。

4. local.conf を編集

最後のステップは、ビルドの local.conf ファイルを編集して、ビルド中のイメージに必要なライブラリを含めることです。希望するレシピの名前を含むように IMAGE_INSTALL_append 行を編集します。以下にその例を示します。

```
IMAGE_INSTALL_append="wolfSSL wolfssh wolfmqtt"
```

イメージがビルドされると、wolfSSL のデフォルトの場所 (またはレシピからの関連製品) は /usr/lib/ ディレクトリになります。

さらに、wolfSSL は後のビルドオプション節にリストされている有効化および無効化オプションを使用して、Yocto にビルドする際にカスタマイズできます。これには、.bbappend ファイルを作成し、wolfSSL アプリケーション/レシピレイヤー内に配置する必要があります。このファイルの内容には、EXTRA_OECONF 変数に連結する内容を指定する行を含める必要があります。以下に、TLS 1.3 有効化オプションを通じて TLS 1.3 サポートを有効にする例を示します。

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

Yocto へのビルドに関する詳細なドキュメントは、meta-wolfSSL README に掲載しています。

<https://github.com/wolfSSL/meta-wolfSSL/blob/master/readme.md>

2.5.2 Atollic TrueSTUDIO でのビルド

3.15.5 以降の wolfSSL バージョンには、ARM M4-Cortex デバイス上で wolfSSL をビルドするために使用される TrueSTUDIO プロジェクトファイルが含まれています。TrueSTUDIO プロジェクトファイルは、ST Microelectronics の一部である Atollic によって作成されたもので、無料でダウンロードできます。これを使用することで、STM32 デバイス上でのビルドプロセスを簡素化できます。TrueSTUDIO で wolfSSL 静的ライブラリプロジェクトファイルをビルドするには、TrueSTUDIO を開いた後、ユーザーが以下の手順を実行する必要があります。

1. プロジェクトをワークスペースにインポートする (ファイル > インポート)
2. プロジェクトをビルドする (プロジェクト > プロジェクトのビルド)

ビルド時に user_settings.h 内にある設定がインクルードされます。user_settings.h ファイルのデフォルトの内容は最小限であり、多くの機能は含まれていません。ユーザーはこのファイルを変更し、後の章に示すオプションで機能を追加または削除できます。

2.5.3 IAR でのビルド

<wolfSSL_root>/IDE/IAR-EWARM ディレクトリには、以下のファイルが含まれています。

1. ワークスペース: wolfSSL.eww ワークスペースには、wolfSSL-Lib ライブラリと wolfCrypt-test、wolfCrypt-benchmark 実行可能プロジェクトが含まれています。
2. wolfSSL-Lib プロジェクト: lib/wolfSSL-lib.ewp wolfCrypt と wolfSSL 関数の完全なセットライブラリを生成します。

3. テストスイートプロジェクト：test/wolfCrypt-test.ewp test.out テストスイート実行可能ファイルを作成します。
4. ベンチマークプロジェクト：benchmark/wolfCrypt-benchmark.ewp benchmark.out ベンチマーク実行可能ファイルを作成します。

これらのプロジェクトは一般的な ARM Cortex-M MPU 向けに設定されています。特定のターゲット MPU 用のプロジェクトを作成するには、以下の手順を実行してください。

1. デフォルト設定：プロジェクトのデフォルトターゲットは Cortex-M3 シミュレータに設定されています。user_settings.h にはプロジェクトのデフォルトオプションが含まれています。シミュレータに対してビルドしてダウンロードできます。「view」->「Terminal I/O」でターミナル I/O ウィンドウを開き、実行を開始します。
2. プロジェクトオプション設定：各プロジェクトに適切な「ターゲット」オプションを選択します。
3. 実行可能プロジェクトの場合：MPU 用の「SystemInit」と「startup」を追加し、デバッグ用の「ドライバ」を選択します。
4. ベンチマークプロジェクトの場合：current_time 関数のオプションを選択するか、WOLFSSL_USER_CURRTIME オプションで独自の「current_time」ベンチマークタイマーを作成します。
5. ビルドとダウンロード：EWARM ビルドとダウンロードのために、メニューバーの「Project->Make」と「Download and Debug」に進みます。

2.5.4 macOS や iOS でのビルド

2.5.4.1 Xcode <wolfSSL_root>/IDE/XCODE ディレクトリには、以下のファイルが含まれています：

1. wolfSSL.xcworkspace - ライブラリとテストスイートクライアントを含むワークスペース
2. wolfSSL_testsuite.xcodeproj - テストスイートを実行するプロジェクト
3. wolfSSL.xcodeproj - wolfSSL および/または wolfCrypt 用の OS/x と iOS ライブラリをビルドするプロジェクト
4. wolfSSL-FIPS.xcodeproj - 利用可能な場合、wolfSSL と wolfCrypt-FIPS をビルドするプロジェクト
5. user_settings.h - プロジェクト間で共有されるカスタムライブラリ設定

ライブラリは、ターゲットに応じて libwolfSSL_osx.a または libwolfSSL_ios.a として出力されます。また、wolfSSL/wolfCrypt (および CyaSSL/CtaoCrypt 互換性) ヘッダーを Build/Products/Debug または Build/Products/Release にある include ディレクトリにコピーします。

ライブラリとテストスイートを適切にリンクするには、ビルドの場所をワークスペースに対して相対的に設定する必要があります。

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

これらの Xcode プロジェクトは、複数のプロジェクトでマクロを設定するための user_settings.h ファイルを有効にするために、WOLFSSL_USER_SETTINGS プリプロセッサを定義しています。

必要に応じて、Xcode プリプロセッサは以下の手順で変更できます。

1. 「Project Navigator」でプロジェクトをクリックします。
2. 「Build Settings」タブをクリックします。
3. 「Apple LLVM 6.0 - Preprocessing」セクションまでスクロールします。
4. 「Preprocessor Macros」の開示を開き、「+」と「-」ボタンを使用して変更します。これを Debug と Release の両方に対して行ってください。

このプロジェクトは、デフォルト設定を使用して wolfSSL と wolfCrypt をビルドする必要があります。

2.5.5 GCC ARM でのビルド

<wolfSSL_root>/IDE/GCC-ARM ディレクトリには、Cortex M シリーズ用の wolfSSL プロジェクト例がありますが、他のアーキテクチャに適用することもできます。

1. gcc-arm-none-eabi がインストールされていることを確認してください。
2. Makefile.common を変更します：
 - 正しいツールチェーンパス TOOLCHAIN を使用します。
 - 正しいアーキテクチャ「ARCHFLAGS」を使用します。[GCC ARM Options](#) -mcpu=name を参照してください。
 - linker.ld のメモリマップがフラッシュ/RAM と一致することを確認するか、Makefile.common の SRC_LD = -T./linker.ld をコメントアウトします。
3. make を使用して、静的ライブラリ (libwolfSSL.a)、wolfCrypt テスト/ベンチマーク、および wolfSSL TLS クライアントターゲットを/Build 内の .elf と .hex としてビルドします。

2.5.5.1 一般的な Makefile を使ったクロスコンパイルによるビルド

Cortex-A53 を搭載した Raspberry Pi 用の Makefile.common 変更例を示します。

1. Makefile.common 内の ARCHFLAGS を -mcpu=cortex-a53 -mthumb に変更します。
2. カスタムメモリマップが適用されないため、SRC_LD をコメントアウトします。
3. TOOLCHAIN をクリアしてデフォルト gcc を使用します。“TOOLCHAIN =” とします。
4. LDFLAGS += --specs=nano.specs と LDFLAGS += --specs=nosys.specs をコメントアウトしてください。

2.5.5.2 configure でのクロスコンパイルによるビルド

メインプロジェクトディレクトリの構成スクリプトは、gcc-arm-none-eabi ツールでビルドするようにクロスコンパイルを実行できます。以下の例ではツールへのパスが適切に設定されていると仮定します。

```
./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_SOCKET -DNO_WRITEV" \
  --disable-filesystem --enable-crypttests \
  --disable-shared
make
make install
```

32 ビットアーキテクチャをビルドしている場合は、-DTIME_T_NOT_64BIT を CFLAGS に追加してください。

2.5.6 Keil MDK-ARM でのビルド

Keil MDK-ARM で wolfSSL をビルドするための詳細な手順とヒントは[こちら](#)で見つけることができます。

注意：MDK-ARM がデフォルトではない場所にインストールされている場合、プロジェクトファイルの参照パス定義をすべて変更する必要があります。

2.6 C プリプロセッサマクロとして定義される機能

2.6.1 機能の削除

以下の定義は、wolfSSL から機能を削除するために使用できます。これは、ライブラリ全体のフットプリントサイズを削減しようとしている場合に役立ちます。NO_< 機能名 > 定義を定義することに加えて、ビルドから対応するソースファイルも削除できます。ヘッダーファイルは対象外です。

2.6.1.1 NO_WOLFSSL_CLIENT サーバーのみのビルド用です。クライアントに固有の呼び出しを削除します。サイズのためにいくつかの呼び出しを削除する場合にのみ、これを使用します。

2.6.1.2 NO_WOLFSSL_SERVER クライアントのみのビルド用です。サーバーに固有の呼び出しを削除します。サイズのためにいくつかの呼び出しを削除する場合にのみ、これを使用します。

2.6.1.3 NO_DES3 DES3 暗号化の使用を削除します。一部の古いサーバーがそれを使用しているため、DES3 はデフォルトで組み込まれており、SSL 3.0 では必要です。NO_DH および NO_AES は上記 2 つと同じで、広く使用されています。

2.6.1.4 NO_DSA DSA の使用を削除します。DSA は、一般的な使用が段階的に廃止されています。

2.6.1.5 NO_ERROR_STRINGS エラー文字列を無効にします。エラー文字列は、wolfSSL の場合は `src/internal.c` または wolfCrypt の `wolfcrypt/src/asn.c` にあります。

2.6.1.6 NO_HMAC ビルドから HMAC を削除します。

注意：SSL/TLS は HMAC に依存します。wolfCrypt のみを使用している場合、（ビルドオプション WOLFCRYPT_ONLY）HMAC を無効にできます。

2.6.1.7 NO_MD4 ビルドから MD4 を削除します。MD4 は解読されているため、使用してはなりません。

2.6.1.8 NO_MD5 ビルドから MD5 を削除します。

2.6.1.9 NO_SHA ビルドから SHA-1 を削除します。

2.6.1.10 NO_SHA256 ビルドから SHA-256 を削除します。

2.6.1.11 NO_PSK 事前共有キー拡張機能の使用をオフにします。デフォルトでは組み込まれています。

2.6.1.12 NO_PWDBASED PKCS # 12 から PBKDF1、PBKDF2、PBKDF などのパスワードベースの鍵導出関数を無効にします。

2.6.1.13 NO_RC4 ビルドから ARC4 ストリーム暗号の使用を削除します。ARC4 はまだ人気があり広く使用されているため、デフォルトで組み込まれています。

2.6.1.14 NO_SESSION_CACHE セッションキャッシュが不要なときに定義できます。これにより、メモリ使用量を約 3KB 減らすことができます。

2.6.1.15 NO_TLS TLS をオフにします。TLS をオフに設定することはお勧めしません。

2.6.1.16 SMALL_SESSION_CACHE wolfSSL が使用する SSL セッションキャッシュのサイズを制限するように定義できます。これにより、デフォルトのセッションキャッシュが 33 セッションから 6 セッションに短縮され、約 2.5KB 削減できます。

2.6.1.17 NO_RSA RSA アルゴリズムのサポートを削除します。

2.6.1.18 WC_NO_RSA_OAEP OAEP パディングのコードを削除します。

2.6.1.19 NO_AES_CBC AES-CBC アルゴリズムサポートをオフにします。

2.6.1.20 NO_AES_DECRYPT コードサイズを削減するために設定できます。復号を無効にし、暗号化のみをサポートするように設定します。

2.6.1.21 WOLFCRYPT_ONLY TLS を無効にして wolfCrypt のみを有効にします。

2.6.1.22 NO_CAMELLIA_CBC Camellia CBC サポートを無効にします。TLS 暗号スイートにのみ適用されます。

2.6.1.23 NO_AES AES アルゴリズムサポートを無効にします。

2.6.1.24 NO_AES_128 コンパイル時の AES キーサイズ選択に使用されます。

2.6.1.25 NO_AES_192 コンパイル時の AES キーサイズ選択に使用されます。

2.6.1.26 NO_AES_256 コンパイル時の AES キーサイズ選択に使用されます。

2.6.1.27 NO_AESGCM_AEAD AES GCM を使用する TLS 暗号スイートを無効にします。AES GCM 暗号スイートが有効になっていない場合に内部的に使用するものですが、暗号スイートを制限するためにも使用できます。

2.6.1.28 NO_ASN_TIME ASN の時間チェックを無効にします。

注意：すべての証明書の開始/終了日チェックがスキップされるため、これは注意して使用する必要があります。

2.6.1.29 NO_CHECK_PRIVATE_KEY このマクロは、デフォルトでオンになっている追加の秘密鍵チェックを無効にします。これにより、秘密鍵が公開鍵のペアであることを検証するチェックが有効になります。RSA、ECDSA、ED25519、ED448、Falcon、Dilithium および Sphincs でサポートされています。

2.6.1.30 NO_DH Diffie-Hellman (DH) サポートを無効にします。

2.6.1.31 NO_ED25519_CLIENT_AUTH ED25519 の TLS クライアント認証サポートを無効にします。メッセージのキャッシュが必要なため、ED25519 が使用されていない場合、TLS 中のメモリ使用量を削減するために使用されます。

2.6.1.32 NO_ED448_CLIENT_AUTH ED448 のクライアント認証を無効にします。

2.6.1.33 NO_FORCE_SCR_SAME_SUITE デフォルトでは、セキュアな再ネゴシエーションでは同じ暗号スイートを使用する必要があります。これはその要件を無効にします。

2.6.1.34 NO_MULTIBYTE_PRINT 組み込みデバイスが問題を抱える可能性のある、マルチバイト文字をコンパイルアウトするために使用されます。

2.6.1.35 NO_OLD_SSL_NAMES wolfSSL と OpenSSL を一緒に使用するための、古い OpenSSL 互換性マクロの一部を無効にします。

2.6.1.36 NO_OLD_WC_NAMES 不要な名前空間を削除します。

2.6.1.37 NO_OLD_POLY1305 相互運用性のために使用される、古い ChaCha20/Poly1305 TLS 1.2 暗号スイートのサポートを無効にします。

2.6.1.38 NO_HANDSHAKE_DONE_CB wolfSSL_SetHsDoneCb で設定されるハンドシェイクコールバックのサポートを無効にします。このオプションはコードサイズを削減するのに役立ちます。

2.6.1.39 NO_STDIO_FILESYSTEM これは stdio.h のインクルードを無効にします。移植性を保つために使用されます。

2.6.1.40 NO_TLS_DH TLS DH を除外します。一時的な有限体 Diffie-Hellman 鍵合意に基づく暗号スイートをネゴシエートすべきではありません。

2.6.1.41 NO_WOLFSSL_CM_VERIFY 証明書マネージャの検証コールバックを無効にします。検証コールバックはエラーをインターセプトして上書きすることを可能にします。このオプションはコードサイズを削減するのに役立ちます。

2.6.1.42 NO_WOLFSSL_DIR ディレクトリサポートを無効にします。

2.6.1.43 NO_WOLFSSL_RENESAS_TSIP_TLS_SESSION TSIP TLS リンク共通鍵暗号化方式のみを無効にします。

注意：これはルネサス RX TSIP 固有の定義です。

2.6.1.44 NO_WOLFSSL_SHA256 これは TLS 1.3 にのみ適用されます。SHA2-256 を wolfCrypt から有効にして使用できるようにしますが、TLS 1.3 からは除外します。

2.6.1.45 WOLFSSL_BLIND_PRIVATE_KEY 秘密鍵をブラインドするためのマスクとして使用されます。ブラインド処理はロウハンマー攻撃から保護するために使用されます。

2.6.1.46 WOLFSSL_DTLS13_NO_HRR_ON_RESUME これが定義されている場合、DTLS サーバーはクライアントの再開が成功した場合にクッキー交換を行いません。

再開はより速く（1 RTT 少なく）なり、帯域幅の消費が少なくなります。（1 つの ClientHello と、1 つの HelloVerifyRequest/HelloRetryRequest が少なくなります）一方、有効な SessionID/チケット/psk が収集された場合、偽造された clientHello メッセージはサーバー上のリソースを消費します。

DTLS 1.3 の場合、このオプションを使用すると、サーバーが Early Data/0-RTT Data を処理することもできます。これがなければ、サーバーはクッキーを持つ検証済み ClientHello を受信するまでステートフル処理に入らないため、Early Data はドロップされます。

クッキー交換なしで DTLS 1.3 再開を許可するには、次のようにします。

1. WOLFSSL_DTLS13_NO_HRR_ON_RESUME を定義して wolfSSL をコンパイルする
2. wolfSSL_dtls13_no_hrr_on_resume(ssl, 1) を WOLFSSL オブジェクトに呼び出して、再開時のクッキー交換を無効にする
3. 通常の接続と同様に続行する

2.6.1.47 WOLFSSL_NO_CLIENT_AUTH Ed25519 と Ed448 を使用するために必要な、キャッシングコードを無効にします。

2.6.1.48 WOLFSSL_NO_CURRDIR wolfSSL/test.h のテストパスに ./ をサポートしないプラットフォーム用の移植性マクロです。テストツールにのみ適用されます。

2.6.1.49 WOLFSSL_NO_DEF_TICKET_ENC_CB デフォルトのチケット暗号化コールバックを定義しません。サーバーのみに適用されます。アプリケーションはセッションチケットを使用するために独自のコールバックを設定する必要があります。

2.6.1.50 WOLFSSL_NO SOCK 移植のため、組み込みソケットサポートを無効にするためのマクロです。ソケットなしで TLS を使用する場合、通常は WOLFSSL_USER_IO を定義し、送信/受信にコールバックを使用します。

2.6.1.51 WOLFSSL_NO_TLS12 TLS 1.2 を除外するために定義します。

2.6.1.52 WOLFSSL_PEM_TO_DER PEM から DER への変換を無効にします。

2.6.1.53 NO_DEV_URANDOM /dev/urandom の使用を無効にします。

2.6.1.54 WOLFSSL_NO_SIGALG 署名アルゴリズムの拡張子を無効にします。

2.6.1.55 NO_RESUME_SUITE_CHECK TLS 接続を再開するときに暗号スイートのチェックを無効にします。

2.6.1.56 NO_ASN ASN フォーマットの証明書処理のサポートをオフにします。

2.6.1.57 NO_OLD_TLS SSLV3、TLSV1.0、TLSV1.1 のサポートを削除します。

2.6.1.58 WOLFSSL_AEAD_ONLY 非 AEAD アルゴリズムのサポートを削除します。AEAD は、「認証された暗号化」の略であり、これらのアルゴリズム (AES-GCM など) がデータを暗号化および復号化するだけでなく、そのデータの機密性と信頼性を保証するアルゴリズムです。

2.6.1.59 WOLFSSL_SP_NO_2048 RSA/DH 2048 ビット単精度 (SP) 最適化を削除します。

2.6.1.60 WOLFSSL_SP_NO_3072 RSA/DH 3072 ビット単精度 (SP) 最適化を削除します。

2.6.1.61 WOLFSSL_SP_NO_256 SECP256R1 の ECC 単精度 (SP) 最適化を削除します。WOLFSSL_SP_MATH にのみ適用されます。

2.6.2 機能の有効化 (デフォルトで有効)

2.6.2.1 HAVE_TLS_EXTENSIONS ほとんどの TLS ビルドに必要な TLS 拡張機能のサポートを有効にします。

./configure を使用する場合はデフォルトでオンになっていますが、WOLFSSL_USER_SETTINGS でビルドする場合は手動で定義する必要があります。

2.6.2.2 HAVE_SUPPORTED_CURVES TLS でサポートされている曲線と、TLS で使用されるキー共有拡張機能を有効にします。ECC、Curve25519、および Curve448 が必要です。

./configure を使用する場合はデフォルトでオンになっていますが、WOLFSSL_USER_SETTINGS でビルドする場合は手動で定義する必要があります。

2.6.2.3 HAVE_EXTENDED_MASTER TLS v1.2 以前で使用するセッションキーの計算用に拡張マスターシークレット PRF を有効にします。PRF 方式はデフォルトでオンになっており、より安全であると考えられています。

./configure を使用する場合はデフォルトでオンになっていますが、WOLFSSL_USER_SETTINGS でビルドする場合は手動で定義する必要があります。

2.6.2.4 HAVE_ENCRYPT_THEN_MAC ブロック暗号による暗号化後に MAC を実行するための encrypt-then-mac サポートを有効にします。これがデフォルトで、セキュリティが向上します。

./configure を使用する場合はデフォルトでオンになっていますが、WOLFSSL_USER_SETTINGS でビルドする場合は手動で定義する必要があります。

2.6.2.5 HAVE_ONE_TIME_AUTH Poly 認証を設定するため、TLS v1.2 で Chacha20/Poly1305 を使用する場合に必要です。

./configure を使用する場合は、これは ChaCha20/Poly1305 でデフォルトで有効になりますが、WOLFSSL_USER_SETTINGS でビルドする場合は手動で定義する必要があります。

2.6.2.6 WOLFSSL_ASN_TEMPLATE テンプレートベースの ASN.1 処理を使用する新しいバージョンの ASN 解析コードを有効にします。この解析は標準的な ASN.1 ルールに準拠し、テンプレート構造を使用してエンコードとデコードを指示することで、パーサーコードがテンプレート間で汎用化できるようにします。./configure を使用する場合はデフォルトでオンになっていますが、WOLFSSL_USER_SETTINGS でビルドする場合は手動で定義する必要があります。

2.6.2.7 WOLFSSL_DEBUG_ASN_TEMPLATE ASN.1 テンプレートを使用する際のデバッグ出力を有効にします。WOLFSSL_ASN_TEMPLATE と併用する場合にのみ関連します。

2.6.2.8 WOLFSSL_ASN_TEMPLATE_TYPE_CHECK テスト用にコンパイラの型の問題をより適切にテストするために ASN 関数を使用します。WOLFSSL_ASN_TEMPLATE と併用する場合にのみ関連します。

2.6.3 機能の有効化 (デフォルトで無効)

2.6.3.1 WOLFSSL_CERT_GEN wolfSSL の証明書生成機能をオンにします。詳細については、第 7 章 キーと証明書を参照してください。

2.6.3.2 WOLFSSL_DER_LOAD wolfSSL_CTX_der_load_verify_locations() 関数を使用して、wolfSSL コンテキスト (WOLFSSL_CTX) への DER フォーマットされた CA 証明書をロードできます。

2.6.3.3 WOLFSSL_DTLS DTLS(Datagram TLS) の使用をオンにします。これは広くサポートされていないか、使用されていません。

2.6.3.4 WOLFSSL_KEY_GEN wolfSSL の RSA 鍵生成機能をオンにします。詳細については、第 7 章 キーと証明書を参照してください。

2.6.3.5 WOLF_PRIVATE_KEY_ID これは PKCS11 で使用され、キー ID とラベル API のサポートを有効にします。FIPS v5 以前は暗号コールバックで WOLF_PRIVATE_KEY_ID をサポートしていません。

2.6.3.6 WOLFSSL_WOLFSENTRY_HOOKS wolfSSL_CTX_set_AcceptFilter() と wolfSSL_CTX_set_ConnectFilter() を使用した一般的なネットワーク accept および connect フィルターフックのサポートを TLS レイヤーに追加します。また、サンプルクライアントおよびサーバーアプリケーションでの wolfSentry 統合も有効化します。

2.6.3.7 WOLFSSL_CERT_EXT 証明書拡張、鍵および証明書生成機能です。

2.6.3.8 WOLFSSL_CERT_REQ 証明書要求、鍵および証明書生成機能です。

2.6.3.9 WOLFSSL_SSLKEYLOGFILE これは Wireshark で使用されるキーロギングを有効にします。マスターシークレットとクライアントランダムがファイルに書き込まれるため、コンパイラが警告を発するようになります。これはテストに役立ちますが、本番環境ではお勧めしません。

2.6.3.10 WOLFSSL_SSLKEYLOGFILE_OUTPUT このマクロはキーロギングのファイル名を定義します。WOLFSSL_SSLKEYLOGFILE と共に使用してください。

2.6.3.11 WOLFSSL_HAVE_WOLFSCPEP wolfSCEP が利用可能かどうかを確認するために autoconf で使用される機能を有効にします。

2.6.3.12 WOLFSSL_HAVE_MIN このマクロはライブラリの移植性を高めるためのもので、MIN/MAX がすでにプラットフォームによって定義されているかどうかを示します。重複定義を防ぎます。

2.6.3.13 WOLFSSL_HAVE_TLS_UNIQUE 「tls-unique」チャンネルバインディングとして使用するために、TLS ハンドシェイク後に「Finished」メッセージを保持します。

libest port で追加：アプリケーションが「tls-unique」チャンネルバインディングタイプを取得できるようにします。

- [RFC 5929 Section 3](#)

これは EST プロトコルで使用され、「所有証明」を通じて TLS セッションに登録をバインドします。

- [RFC 7030 Section 3.4](#)
- [RFC 7030 Section 3.5](#)

2.6.3.14 WOLFSSL_ENCRYPTED_KEYS 暗号化キー PKCS8 サポートを有効にします。このマクロは PKCS8 パスワードベースのキー暗号化を有効にします。

- [RFC 5208](#)

2.6.3.15 WOLFSSL_CUSTOM_OID サブジェクトおよび要求拡張用のカスタム OID サポートを有効にする証明書機能です。これはカスタム OID を持つ証明書の解析にも適用されます。

2.6.3.16 WOLFSSL_RIPEMD RIPEMD-160 サポートを有効にします。

2.6.3.17 WOLFSSL_SHA384 SHA-384 サポートを有効にします。

2.6.3.18 WOLFSSL_SHA512 SHA-512 サポートを有効にします。

2.6.3.19 WOLFSSL_AES_DIRECT AES ECB モードのダイレクトサポートを有効にします。それ自体では ECB モードは安全とは見なされません。この機能は PKCS7 に必要です。

警告： ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

2.6.3.20 DEBUG_WOLFSSL デバッグ機能を含めてビルドします。詳細は第 8 章 デバッグを参照してください。

2.6.3.21 HAVE_AESCCM AES-CCM サポートを有効にします。

2.6.3.22 HAVE_AESGCM AES-GCM サポートを有効にします。

2.6.3.23 WOLFSSL_AES_XTS AES-XTS サポートを有効にします。

2.6.3.24 HAVE_CAMELLIA Camellia サポートを有効にします。

2.6.3.25 HAVE_CHACHA Chacha20 サポートを有効にします。

2.6.3.26 HAVE_POLY1305 Poly1305 サポートを有効にします。

2.6.3.27 HAVE_CRL 証明書失効リスト (CRL) サポートを有効にします。

2.6.3.28 HAVE_CRL_IO CRL URL でインライン HTTP リクエストをブロックできるようにします。CRL を WOLFSSL_CTX にロードし、作成したすべての wolfSSL オブジェクトに適用します。

2.6.3.29 HAVE_ECC 楕円曲線暗号化 (ECC) サポートを有効にします。

2.6.3.30 HAVE_LIBZ 接続上のデータの圧縮を可能にする拡張機能です。通常、使用すべきではありません。「ビルドオプション」節にある `--with-libz=PATH` の下のメモを参照してください。

2.6.3.31 OPENSSL_EXTRA ライブラリにさらに多くの OpenSSL 互換性を組み込み、wolfSSL OpenSSL 互換性レイヤーを有効にします。これにより、OpenSSL で動作するように設計された既存のアプリケーションへの移植を容易にします。デフォルトでオフになっています。

2.6.3.32 HAVE_EXT_CACHE 内部キャッシュの代わりに外部セッションキャッシュを使用するためのサポートを有効にします。

2.6.3.33 WOLFSSL_WPAS_SMALL WPA サプリカントサポート用の互換性レイヤーの小さなサブセットを有効にします。

2.6.3.34 OPENSSL_ALL 統合テスト用のすべての互換性機能のサポートを有効にします。

2.6.3.35 OPENSSL_COEXIST OpenSSL 互換レイヤーです。古い名前を無効にする必要があります。wolfSSL と OpenSSL が共存できるようにするモードです。

2.6.3.36 OPENSSL_VERSION_NUMBER OpenSSL 互換性を実装するバージョン番号を指定します。

2.6.3.37 WOLFSSL_NGINX OpenSSL 互換性をもつアプリケーション固有のマクロです。(--enable-nginx) WOLFSSL_NGINX を使用します。

2.6.3.38 WOLFSSL_ERROR_CODE_OPENSSL OpenSSL 互換性 API wolfSSL_EVP_PKEY_cmp は成功時に 0、失敗時に -1 を返します。この動作は OpenSSL とは異なります。EVP_PKEY_cmp は以下を返します。

- 1: 2 つのキーが一致する
- 0: 一致しない
- -1: キータイプが異なる
- -2: 操作がサポートされていない

この関数が OpenSSL と同じ動作をするようにしたい場合は、WS_RETURN_CODE が OpenSSL と同等の動作に戻り値を変換するように、WOLFSSL_ERROR_CODE_OPENSSL を定義してください。

2.6.3.39 WOLFSSL_HARDEN_TLS RFC 9325 で指定された推奨事項を実装します。このマクロは、望ましいセキュリティビット数に定義する必要があります。現在実装されている値は 112 ビットと 128 ビットです。以下のマクロは特定のチェックを無効にします。

- 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

2.6.3.40 WOLFSSL_ASIO OpenSSL 互換性をもつアプリケーション固有のマクロです。

2.6.3.41 WOLFSSL_QT OpenSSL 互換性をもつアプリケーション固有のマクロです。QT 用の DH Extra、OpenSSL all、OpenSSH、および static ephemeral を有効にします。

2.6.3.42 WOLFSSL_HAPROXY OpenSSL 互換性をもつアプリケーション固有のマクロです。

2.6.3.43 WOLFSSL_ASN_TEMPLATE デュアルアルゴリズム証明書機能です。デフォルトで新しい ASN テンプレート asn.c コードを使用するため、ASN.1 テンプレート機能が必要です。

2.6.3.44 WOLFSSL_ASYNC_IO 非同期クリーンアップで使用されます。

2.6.3.45 WOLFSSL_ATMEL atmel_get_random_number 関数を使用してランダムデータをシードする ASF フックを有効にします。

2.6.3.46 WOLFSSL_CMAC 追加の CMAC アルゴリズムを有効にします。

注意：WOLFSSL_AES_DIRECT が必要です。

2.6.3.47 WOLFSSL_ESPIDF_ERROR_PAUSE test.c でのみ使用され、テストエラー時にデバッグ目的で遅延を追加します。

2.6.3.48 TEST_IPV6 テストアプリケーションでの IPv6 のテストをオンにします。wolfSSL は IP ニュートラルですが、テストアプリケーションはデフォルトで IPv4 を使用しています。

2.6.3.49 TEST_NONBLOCK_CERTS 非ブロッキング OCSP レスポンスのテストにのみ使用されます。WOLFSSL_NONBLOCK_OCSP と OCSP_WANT_READ で有効になります。

2.6.3.50 TEST_OPENSSL_COEXIST 以下のビルドオプションを有効にする場合に使用します。

```
./configure --enable-opensslcoexist
```

2.6.3.51 TEST_PK_PRIVKEY PK コールバックのテストにのみ使用されます。wolfSSL/test.h では PK コールバックでロード・使用される実際の秘密鍵を渡すために、コンテキストを使用します。

2.6.3.52 TEST_BUFFER_SIZE サンプルのクライアント/サーバー-B オプションで使用される、TLS ベンチマークテストバッファサイズのオーバーライドを可能にします。

2.6.3.53 FORCE_BUFFER_TEST ファイルシステムを使用する代わりに、test_certs.h バッファの使用を強制します。wolfSSL/test.h の内部テストにのみ使用されます。

2.6.3.54 WOLFSSL_FORCE_MALLOC_FAIL_TEST ランダムな malloc の失敗を誘発するための内部テスト用に定義します。

2.6.3.55 WOLFSSL_POST_HANDSHAKE_AUTH TLS 拡張機能、ポストハンドシェイク認証に使用されます。

2.6.3.56 WOLFSSL_PSK_MULTI_ID_PER_CS TLS 1.3 PSK において、暗号スイートごとに複数の ID を処理します。

2.6.3.57 WOLFSSL_PUBLIC_ASN 内部で使用される ASN.1 API を公開します。これは、内部の asn.h API を使用して解析したいお客様に役立ちます。

2.6.3.58 WOLFSSL_QUIC QUIC プロトコルのサポートを有効にします。詳細については[こちら](#)を参照してください。

2.6.3.59 WOLFSSL_QUIC_MAX_RECORD_CAPACITY 最大 quic キャパシティを 1024*1024 - 1 MB として定義します。

2.6.3.60 WOLFSSL_RENESAS_FSPSM_TLS まだサポートしていない、TLS 関連機能です。

2.6.3.61 WOLFSSL_RENESAS_TSIP_TLS TSIP TLS リンク共通鍵暗号化方式のみを無効にするためのものです。

2.6.3.62 WOLFSSL_SM2 SM 暗号を使用するために定義します。

2.6.3.63 WOLFSSL_SM3 SM 暗号を使用するために定義します。

- 2.6.3.64 WOLFSSL_SM4** SM 暗号を使用するために定義します。
- 2.6.3.65 WOLFSSL_SM4_CBC** SM4 CBC の SM 設定です。
- 2.6.3.66 WOLFSSL_SM4_CCM** SM4 CCM の SM 設定です。
- 2.6.3.67 WOLFSSL_SM4_GCM** SM4 GCM の SM 設定です。
- 2.6.3.68 WOLFSSL_SNIFFER_CHAIN_INPUT** Chain Input オプションでは、スニファークが入力を生のパケットへのポインタではなく、struct iovec リストとして受け取ることができます。
- 2.6.3.69 XSLEEP_MS** テストのみに使用されます。カスタム遅延を定義できます。
- 2.6.3.70 XSNPRINTF** snprintf 関数をオーバーライドできます。
- 2.6.3.71 DEFAULT_TIMEOUT_SEC** HAVE_IO_TIMEOUT と一緒に使用して、wolfio.c ソケットタイムアウトを秒単位で指定します。これは OCSP および CRL HTTP の内部ソケットコードで使用されます。
- 2.6.3.72 HAVE_IO_TIMEOUT** 証明書の失効に関するものです。IO オプションは接続タイムアウトのサポートを有効にしますが、デフォルトではオフです。
- 2.6.3.73 HAVE_OCSP** オンライン証明書ステータスプロトコル (OCSP) サポートを有効にします。
- 2.6.3.74 HAVE_CSHARP** C # ラッパーに必要な構成オプションをオンにします。
- 2.6.3.75 HAVE_CURVE25519** Curve25519 アルゴリズムの使用をオンにします。
- 2.6.3.76 HAVE_ED25519** ED25519 アルゴリズムの使用をオンにします。
- 2.6.3.77 WOLFSSL_DH_CONST** Diffie Hellman Operations を実行するときにフローティングポイント値の使用をオフにし、XPOW() および XLOG() のテーブルを使用します。外部数学ライブラリへの依存関係を削除します。
- 2.6.3.78 WOLFSSL_TRUST_PEER_CERT** 信頼できるピア証明書の使用をオンにします。これにより、CA 証明書を使用するのではなく、ピア証明書に接続することができます。信頼できるピア証明書が Peer Cert チェーンよりも一致している場合にオンになっていると、ピアが検証されたと見なされます。CA 証明書を使用することが望ましいです。
- 2.6.3.79 WOLFSSL_STATIC_MEMORY** 静的メモリバッファと機能の使用をオンにします。これにより、動的ではなく静的メモリを使用できます。
- 2.6.3.80 WOLFSSL_STATIC_MEMORY_LEAN** 合わせて、WOLFSSL_STATIC_MEMORY の定義が必要です。65k 未満のメモリプールサイズを必要とする構造体に対してより小さい型サイズを使用し、IO バッファなどの使用可能な機能を制限してフットプリントサイズを削減します。
- 2.6.3.81 WOLFSSL_SESSION_EXPORT** DTLS セッションのエクスポートとインポートの使用をオンにします。これにより、DTLS セッションの現在の状態をシリアル化および送受信することができます。

2.6.3.82 WOLFSSL_ARMASM ARMv8 ハードウェアアクセラレーションの使用をオンにします。

2.6.3.83 WC_RSA_NONBLOCK Fast Math RSA ノンブロッキングサポートをオンにして、RSA 操作をより小さな仕事の塊に分割して処理します。この機能は、wc_RsaSetNonBlock() を呼び出し、FP_WOULDBLOCK 戻りコードをチェックすることにより有効になります。

2.6.3.84 WC_RSA_BLINDING タイミング耐性を有効にするために使用されます。

2.6.3.85 WC_RSA_PSS RSA PSS パディングを有効にします。TLS 1.3 でサポートされる唯一の RSA パディングスキームは PSS です（仕様によるものです）。PSS パディングはランダムパディングを使用します。

2.6.3.86 WOLFSSL_RSA_VERIFY_ONLY RSA 用の小さなビルドをオンにします。**WOLFSSL_RSA_PUBLIC_ONLY**、**WOLFSSL_RSA_VERIFY_INLINE**、**NO_SIG_WRAPPER**、**WOLFCRYPT_ONLY**を定義してください。

2.6.3.87 WOLFSSL_RSA_PUBLIC_ONLY RSA の公開キーのみの小さなビルドをオンにします。**WOLFCRYPT_ONLY**を定義してください。

2.6.3.88 WOLFSSL_SHA3 SHA3 使用のビルドをオンにします。これは、SHA3-224、SHA3-256、SHA3-384、SHA3-512 の SHA3 Keccak のサポートです。さらに、WOLFSSL_SHA3_SMALL を使用してパフォーマンスとリソース使用のトレードオフを行うことができます。

2.6.3.89 USE_ECDSA_KEYSZ_HASH_ALGO エフェメラル ECDHE 鍵サイズまたは次に使用可能な次の最高値と一致するハッシュアルゴリズムを選択します。この回避策は、SHA512 でハッシュされた P-256 鍵などのシナリオを正しくサポートしていないいくつかのピアに関する問題を解決します。

2.6.3.90 WOLFSSL_ALT_CERT_CHAINS この定義によって CA 証明書が通信相手から提示されることを許可しますが、有効なチェーンの一部としては使用しません。デフォルトの wolfSSL の動作は、提示されたすべてのピア証明書の検証を要求することです。これにより、中間 CA 証明書を信頼できるものとして扱い、Root CA 証明書まで至る CA の署名エラーは無視されません。代替の証明書チェーンモードでは、ピア証明書が信頼できる CA に検証する必要があります。

2.6.3.91 WOLFSSL_SYS_CA_CERTS wolfSSL_CTX_load_system_CA_certs() が呼び出されたとき、wolfSSL 証明書マネージャーにそれらをロードするか、システム認証 API を呼び出すことで、wolfSSL が検証のためにシステムの CA 証明書を使用できるようにします。詳細は wolfSSL_CTX_load_system_CA_certs() を参照してください。このプリプロセッサマクロは、--enable-sys-ca-certs 構成オプションによって自動的に設定されます。

2.6.3.92 WOLFSSL_APPLE_NATIVE_CERT_VERIFICATION TLS ピア証明書を認証する際に Apple のネイティブトラスト API の使用を有効にします。**WOLFSSL_SYS_CA_CERTS**が定義されている必要があります。iOS や他の Apple デバイスで configure または CMake でビルドする場合、このマクロをユーザーが設定する必要はありませんが、macOS でネイティブ検証方法を使用したい場合は明示的に設定する必要があります。

2.6.3.93 WOLFSSL_CUSTOM_CURVES 標準以外の曲線を許可します。計算では、曲線“a”変数が含まれています。**HAVE_ECC_SECP2**、**HAVE_ECC_SECP3**、**HAVE_ECC_BRAINPOOL** および **HAVE_ECC_KOBLITZ** を使用して、追加の曲線タイプを有効にできます。

2.6.3.94 HAVE_COMP_KEY ECC 圧縮鍵サポートを有効にします。

2.6.3.95 WOLFSSL_EXTRA_ALERTS TLS 接続中に追加のアラートを送信できるようにします。この機能は、`--enable-opensslextra`を使用すると自動的に有効になります。

2.6.3.96 WOLFSSL_DEBUG_TLS TLS 接続中に追加のデバッグプリントアウトを有効にします。

2.6.3.97 HAVE_BLAKE2 Blake2S アルゴリズムのサポートを有効にします。

2.6.3.98 HAVE_FALLBACK_SCSV サーバー側での Signaling Cipher Suite Value(SCSV) サポートを有効にします。これはクライアントから送信される暗号スイート `0x56 0x00` を処理し、TLS バージョンのダウングレードを許可しないことを示します。

2.6.3.99 HAVE_AEAD TLS 1.3 に必要な AEAD を有効にします。

2.6.3.100 HAVE_AES_CBC AES CBC のオプションを有効にします。

2.6.3.101 HAVE_ALPN ALPN を有効にします。

2.6.3.102 HAVE_CAVIUM_OCTEON_SYNC Marvell Cavium/Octeon ハードウェアのブロッキング（同期）バージョンを有効にします。

2.6.3.103 HAVE_CERTIFICATE_STATUS_REQUEST 証明書ステータス要求機能としての証明書失効に使用されます。

2.6.3.104 HAVE_CERTIFICATE_STATUS_REQUEST_V2 証明書ステータス要求機能としての証明書失効に使用されます。

2.6.3.105 HAVE_IO_TIMEOUT 証明書失効に関するものです。IO オプションは接続タイムアウトのサポートを有効にしますが、デフォルトではオフです。

2.6.3.106 HAVE_CURL cURL とリンクする際、wolfSSL ライブラリのサブセットをビルドするために使用されます。

2.6.3.107 HAVE_CURVE448 Curve448 サポート用に定義します。追加のマクロ設定を変更できます。デフォルトでは共有シークレット、キーのエクスポート、およびインポートが有効になっています。

2.6.3.108 HAVE_DANE このオプションは HAVE_RPK（生の公開鍵）でのみサポートされ、将来追加される可能性がある場合のプレースホルダーです。

2.6.3.109 HAVE_DILITHIUM DILITHIUM 量子暗号化/署名アルゴリズムを含めるために有効にします。

2.6.3.110 HAVE_ED25519_KEY_IMPORT ED25519 の設定です。署名、検証、共有シークレット、インポート、およびエクスポートの詳細な制御のために Ed25519 および Curve25519 オプションを有効にします。

2.6.3.111 HAVE_EX_DATA CTX/WOLFSSL のユーザー情報用の「追加」EX データ API を有効にします。

2.6.3.112 HAVE_EX_DATA_CLEANUP_HOOKS インデックスで RSA キーに対して追加データとクリーンアップコールバックを設定します。

2.6.3.113 HAVE_FALCON OpenQuantumSafe からの量子後暗号 FALCON を有効にします。

2.6.3.114 HAVE_FIPS さまざまな FIPS バージョンを実装する際に使用されます。

2.6.3.115 HAVE_KEYING_MATERIAL [RFC 8446 Section 7.5](#)に基づいてキーイング材料のエクスポートを有効にします。

2.6.3.116 HAVE_OID_DECODING ASN テンプレートコードに含まれています。一部のケースでデコードに使用されます。

2.6.3.117 HAVE_MAX_FRAGMENT 最大フラグメントサイズを設定します。TLS 拡張機能です。

2.6.3.118 WOLFSSL_PSK_ONE_ID TLS 1.3 を持つ PSK ID を 1 つだけサポートできます。

2.6.3.119 SHA256_MANY_REGISTERS すべてのデータをレジスタに保持し、部分的にループを展開する SHA256 の処理を指定します。

2.6.3.120 WOLFCRYPT_HAVE_SRP wolfCrypt セキュアリモートパスワードサポートを有効にします。

2.6.3.121 WOLFSSL_MAX_STRENGTH 最強のセキュリティ機能のみを有効にし、弱いまたは廃止予定の機能を無効にします。タイミングベースのサイドチャネル攻撃から保護するためのコンスタント実行により性能が劣化します。

2.6.3.122 MAX_RECORD_SIZE 最大レコードサイズを決定します。標準では 2^{14} が最大サイズです。

2.6.3.123 MAX_CERTIFICATE_SZ 証明書メッセージペイロードの最大サイズを定義します。証明書あたり 2KB、MAX_CHAIN_DEPTH 個の証明書を想定しています。

2.6.3.124 MAX_CHAIN_DEPTH 最大チェーン深度を定義します。

2.6.3.125 MAX_CIPHER_NAME 最大暗号名を定義します。

2.6.3.126 MAX_DATE_SIZE バイト lastdate またはバイト nextdate として使用される日付の最大サイズを定義します。

2.6.3.127 MAX_EARLY_DATA_SZ 最大早期データサイズを定義するために使用されます。

2.6.3.128 WOLFSSL_MAX_SEND_SZ 最大送信サイズを指定するために定義します。

2.6.3.129 WOLFSSL_MAX_SUITE_SZ 最大スイートサイズを指定するために定義します。小さすぎるとエラーが発生します。

2.6.3.130 MAX_WOLFSSL_FILE_SIZE 4 MB の割り当てサイズ制限があります。

2.6.3.131 WOLFSSL_MAXQ10XX_TLS maxq10xx に使用している TLS バージョンを知らせます。

2.6.3.132 WOLFSSL_MAX_SIGALGO 最大署名アルゴリズムをオーバーライドする機能を有効にします。

2.6.3.133 WOLFSSL_MEM_GUARD 指定されたメモリガードを割り当てることができます。

2.6.3.134 WOLFSSL_STATIC_EPHEMERAL TLS スニファーサポートです

2.6.3.135 SSL_SNIFFER_EXPORTS WIN32 スニファーエクスポートです。

2.6.3.136 WOLFSSL_SNIFFER_KEYLOGFILE SSL キーログファイルオプションは、スニファーが [NSS キーログファイル](#) から取得したマスターシークレットを使用して TLS トラフィックを復号化できるようにします。これにより、スニファーは一時的な暗号スイートを使用する TLS 接続でも、すべての TLS トラフィックを復号化できます。キーログファイルスニффイングは TLS バージョン 1.2 および 1.3 でサポートされています。

wolfSSL は、スニファー機能とは別に、`--enable-keylog-export` 構成オプションを使用してキーログファイルをエクスポートするように構成できます。(注意：これは本質的に安全ではないため、本番環境では絶対に行わないでください)

キーログファイルのスニファーサポートを有効にするには、以下の構成コマンドラインを使用してビルドします。

```
./configure --enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_KEYLOGFILE
```

2.6.3.137 WOLFSSL_SNIFFER_STORE_DATA_CB Store Data Callback オプションを使用すると、スニファーは再割り当てされたデータポインタではなくカスタムバッファにアプリケーションデータを保存する際に呼び出されるコールバックを取ることができます。コールバックはすべてのデータが消費されるまでループで呼び出されます。このオプションを有効にするには、以下の構成コマンドラインを使用してビルドします。

```
./configure --enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_STORE_DATA_CB
```

2.6.3.138 WOLFSSL_SNIFFER_WATCH Session Watching オプションを使用すると、スニファーは初期設定なしで提供された任意のパケットを監視できます。すべての TLS セッションのデコードを開始し、サーバーの証明書が検出されると、その証明書はユーザーが提供したコールバック関数に渡され、適切な秘密鍵を提供する必要があります。このオプションを有効にするには、以下の構成コマンドラインを使用してビルドします。

```
./configure --enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_WATCH
```

2.6.3.139 STATIC_BUFFER_LEN レコードヘッダーからメモリをフラグメント化しないでください。RECORD_HEADER_SZ に展開します。

2.6.3.140 STATIC_CHUNKS_ONLY ユーザーは、小さな静的バッファを使用している場合 (デフォルト) に 16K 出力オプションをオフにするオプションがあり、SSL_write が持っているレコードよりも大きなデータを書き込もうとすると、ユーザーが静的バッファチャンクでのみ書き込むように指示しない限り、動的に取得します。

2.6.3.141 WOLFSSL_DEF_PSK_CIPHER ユーザー定義の PSK 暗号を有効にします。

2.6.3.142 WOLFSSL_OLD_PRIME_CHECK より高速な DH と RSA の素数チェックを使用する機能を有効にします。

2.6.3.143 WOLFSSL_STATIC_RSA 静的暗号スイートのみをサポートするレガシーシステム向けに残されています。静的な鍵を使用する暗号は強く非推奨とされており、避けられない場合以外は決して使用しないでください。[WOLFSSL_STATIC_PSK](#)および[WOLFSSL_STATIC_DH](#)も参照してください。

2.6.3.144 WOLFSSL_STATIC_PSK 静的な鍵を使用する暗号は非推奨とされています。[WOLFSSL_STATIC_RSA](#)を参照してください。

2.6.3.145 WOLFSSL_STATIC_DH 静的な鍵を使用する暗号は非推奨とされています。[WOLFSSL_STATIC_RSA](#)を参照してください。

2.6.3.146 HAVE_NULL_CIPHER NULL 暗号のサポートをオンにします。このオプションはセキュリティの観点から強く非推奨ですが、稀に暗号化/復号化操作を実行するには小さすぎるシステムがあります。そのようなシステムでも、このマクロを有効化して少なくともメッセージやピアを認証し、メッセージの改ざんを防ぐことができます。

2.6.3.147 HAVE_ANON 匿名の暗号スイートのサポートをオンにします。推奨されませんが、インターネットから切り離された、またはプライベートネットワークに関連するいくつかの有効なユースケースがあります。

2.6.3.148 HAVE_LIBOQS OpenQuantumSafe チームの LiboQS 統合のサポートをオンにします。詳細は付録 G ポスト量子暗号の実験を参照してください。

2.6.3.149 WOLFSSL_SP_4096 RSA/DH 4096 ビット単精度 (SP) サポートを有効にします。

2.6.3.150 WOLFSSL_SP_384 ECC SECP384R1 単精度 (SP) サポートを有効にします。WOLFSSL_SP_MATH にのみ適用されます。

2.6.3.151 WOLFSSL_SP_1024 Sakke ペアリングベースの単精度 (SP) サポートを有効にします。

2.6.3.152 ATOMIC_USER アトミックレコードレイヤーコールバックを有効にします。

2.6.3.153 BIG_ENDIAN_ORDER デフォルトはリトルエンディアンです。このマクロを定義することで、ビッグエンディアン環境で動作できます。

2.6.3.154 WOLFSSL_32BIT_MILLI_TIME 関数 `TimeNowInMilliseconds()` は符号なし 32 ビット値を返します。デフォルトの動作は符号付き 64 ビット値を返すことです。

2.6.3.155 WOLFSSL_MAX_DHKEY_BITS DH 最大ビットサイズは 8 の倍数でなければなりません。DH 最大ビットサイズは 16384 を超えたり、WOLFSSL_MIN_DHKEY_BITS より大きくすることはできません。

2.6.3.156 WOLFSSL_MIN_DHKEY_BITS DH 最小ビットサイズは 8 の倍数でなければなりません。112 ビットのセキュリティでは、DH は少なくとも 2048 ビットのキーが必要であり、最小ビットサイズは 16000 を超えてはなりません。

2.6.3.157 WOLFSSL_MAX_MTU 最大予想 MTU です。1500 - 100 バイトで、UDP と IP ヘッダーを考慮しています。

2.6.3.158 IGNORE_NETSCAPE_CERT_TYPE ネットスケープ証明書タイプを入力する場所を確保するために定義します。

2.6.3.159 SESSION_CERTS 証明書用の TLS セッションキャッシュです。

2.6.3.160 WOLFSSL_DUAL_ALG_CERTS デュアルアルゴリズム証明書の必須機能です。

2.6.3.161 CRL_MAX_REVOKED_CERTS RevokedCerts を保持するバッファの数を指定します。デフォルト値は 4 に設定されています。

2.6.3.162 CRL_STATIC_REVOKED_LIST バイナリ検索を可能にする RevokedCerts の固定静的リストを有効にします。

2.6.3.163 SESSION_INDEX キャッシュ内のセッションの場所を識別します。インデックスセッション/行シフトを指定します。

2.6.3.164 SESSION_TICKET_HINT_DEFAULT チケットヒントのデフォルトは、デフォルトのヒント値を設定するために使用されます。チケットキーの寿命はチケットのライフヒントよりも長くなければなりません。

2.6.3.165 WOLFSSL_DTLS13 wolfSSL DTLS 1.3 を有効にします。

2.6.3.166 WOLFSSL_TLS13 TLS 1.3 プロトコル実装を有効にします。

2.6.3.167 WOLFSSL_TLS13_IGNORE_AEAD_LIMITS [RFC 9147 Section 4.5.3](#) に示された制限です。鍵更新が必要な制限を、ハードな復号化失敗制限の中間地点として指定しています。

2.6.3.168 WOLFSSL_TLS13_MIDDLEBOX_COMPAT TLS 1.3 ハンドシェイクでミドルボックス互換性を有効にします。これには、暗号化されたメッセージの前に ChangeCipherSpec を送信することとセッション ID を含めることが含まれます。

2.6.3.169 WOLFSSL_TLS13_SHA512 ハンドシェイクでの SHA-512 ダイジェストの生成を許可します。ただし、現時点ではどの暗号スイートも SHA-512 を必要としません。これにより、TLS v1.3 ではまだ使用されていませんが、ハンドシェイクメッセージの SHA2-512 ハッシュの計算が可能になります。

2.6.3.170 WOLFSSL_UIP CONTIKI が定義されている場合、これは UIP の実装です。

2.6.3.171 TLS13_MAX_TICKET_AGE 最大チケット期間を指定します。TLS 1.3 の場合、これは 7 日間です。

2.6.3.172 TLS13_TICKET_NONCE_STATIC_SZ TLS13_TICKET_NONCE_STATIC_SZ はこの FIPS_VERSION_GE ではサポートされていません。

2.6.3.173 TLS13_TICKET_NONCE_MAX_SZ チケットナンス用のバージョン最大サイズを定義します。最大サイズは 255 バイトとして定義されています。

2.6.3.174 WOLFSSL_TICKET_ENC_AES128_GCM デフォルトコールバックでセッションチケットの暗号化/復号化に AES128-GCM を使用します。これはサーバーのみで適用されます。ChaCha20/Poly1305 がコンパイルされていない場合、これがデフォルトのアルゴリズムです。

2.6.3.175 WOLFSSL_TICKET_ENC_AES256_GCM デフォルトコールバックでセッションチケットの暗号化/復号化に AES256-GCM を使用します。これはサーバーのみで適用されます。

2.6.3.176 WOLFSSL_TICKET_ENC_CHACHA20_POLY1305 デフォルトコールバックでセッションチケットの暗号化/復号化に ChaCha20-Poly1305 を使用します。何も定義されていない場合、デフォルトのアルゴリズムが使用され、アルゴリズムがコンパイルされます。これはサーバーのみで適用されます。

2.6.3.177 WOLFSSL_TICKET_EXTRA_PADDING_SZ チケットの追加パディングサイズを 32 として定義します。

2.6.3.178 WOLFSSL_TICKET_HAVE_ID チケットに ID があることを確認するために使用します。サポートが組み込まれていて、チケットに ID が含まれている場合にのみキャッシュに追加します。そうでなければ、キャッシュからチケットを取得する方法がありません。

2.6.3.179 WOLFSSL_TICKET_KEY_LIFETIME デフォルトの寿命は、キーを使用した最初のチケットの発行から 1 時間です。これはヒントよりも長くなければなりません。

2.6.3.180 WOLFSSL_TICKET_NONCE_MALLOC チケットナンスの動的割り当てを有効にします。HKDF 展開コールバックを無効にする必要があります。

2.6.3.181 SHOW_CERTS 定義されている場合、証明書を表示します。組み込みデバッグに使用します。

2.6.3.182 SHOW_SECRETS デバッグに使用されます。適用可能なシークレットを表示します。

2.6.3.183 DEBUG_UNIT_TEST_CERTS 名前制約テストをデバッグする際に使用されます。複雑な定義ガードを持つ複数の場所で使用できるように静的ではありません。

2.6.3.184 DEBUG_WOLFSSL_VERBOSE OPENSSL_EXTRA または DEBUG_WOLFSSL_VERBOSE マクロを使用する場合、WOLFSSL_ERROR は新しい関数 WOLFSSL_ERROR_LINE にマップされ、WOLFSSL_ERROR が呼び出される行番号と関数名を取得します。

2.6.3.185 SOCKET_INVALID 無効なソケットを定義するために使用され、-1 として定義されています。

2.6.3.186 WOLFSSL_SOCKET_INVALID テストに使用され、無効なソケットを示すために使用される値のオーバーライドのみを許可します。通常は -1 です。

2.6.3.187 WOLFSSL_SOCKET_IS_INVALID ソケット処理で使用されます。

2.6.3.188 WOLFSSL_SRTP SRTP をアクティブにするために使用されます。

2.6.3.189 WOLFSSL_CIPHER_CHECK_SZ 暗号化操作が機能したことを確認するために 64 ビットが必要な暗号チェックサイズとして定義されています。

2.6.3.190 DTLS_CID_MAX_SIZE DTLS 1.3 パーシングコードは、レコードを復号化するために静的バッファにレコードヘッダーをコピーします。CID 最大サイズを増やすと、このバッファも増加し、セッションごとの実行時メモリフットプリントに影響します。DTLS CID の最大サイズは 255 バイトです。

2.6.3.191 DTLS13_EPOCH_SIZE DTLS 1.3 エポックを使用した、移植性を高めるためのマクロです。DTLS エポックにバインドされた鍵を保存し、必要に応じて適切な鍵/エポックを設定する方法を実装します。

2.6.3.192 DTLS13_RETRANS_RN_SIZE DTLS 1.3 における移植性を高めるためのマクロです。DTLS 1.3 で再送信前のサイズを識別するために使用されます。

2.6.3.193 WOLFSSL_DTLS_FRAG_POOL_SZ 指定された時間あたりに許可されるフラグメントの数を定義します。

2.6.3.194 WOLFSSL_CLIENT_SESSION_DEFINED API が使用する不透明な構造体を宣言します。

2.6.3.195 WOLFSSL_COND このシステムが COND_TYPE シグナリングをサポートしている場合に定義されます。シグナリング API に渡されるべきタイプです。

2.6.3.196 WOLFSSL_DTLS_CH_FRAG サーバーがフラグメント化された 2 番目/検証済み（有効なクッキー応答を含む）ClientHello メッセージを処理できるようにします。1 番目/未検証（クッキー拡張を含まない）ClientHello はフラグメント化されてはならず、DTLS サーバーがステートレスに処理できるようにする必要があります。これは DTLS 1.3 でのみ実装されています。ユーザーは、実行時にこれを明示的に有効にするために、サーバーで `wolfSSL_dtls13_allow_ch_frag()` を呼び出す必要があります。

注意：DTLS 1.3 + pqc を WOLFSSL_DTLS_CH_FRAG なしで使用すると、おそらく失敗します。この場合、`--enable-dtls-frag-ch` を使用して有効にします。

2.6.3.197 WOLFSSL_DTLS_MTU_ADDITIONAL_READ_BUFFER 私たちとは少し異なる MTU を持つピアと連携できるように、追加のバイトを読み取る必要がある場合に使用します。

2.6.3.198 WOLFSSL_DTLS_WINDOW_WORDS ウィンドウのストレージサイズをチェックするか、インデックスがウィンドウに対して有効かどうかを確認するために使用されます。

2.6.3.199 WOLFSSL_EXPORT_SPC_SZ CipherSpecs から使用されるバイト数を指定するために定義します。

2.6.3.200 WOLFSSL_MIN_DOWNGRADE 最小ダウングレードバージョンを指定します。

2.6.3.201 WOLFSSL_MIN_DTLS_DOWNGRADE 最小 DTLS ダウングレードバージョンを指定します。

2.6.3.202 WOLFSSL_MIN_ECC_BITS 許可される ECC キーサイズの最小値を設定できます。

2.6.3.203 WOLFSSL_MIN_RSA_BITS デフォルトでは、wolfSSL は RSA キーサイズを最小 1024 ビットに制限しています。512 ビットキーなどの小さく安全性の低い RSA キーのデコードを許可するには、コンパイラフラグ-DWOLFSSL_MIN_RSA_BITS=512 を CFLAGS または CPPFLAGS に追加するか、ユーザー設定ヘッダーで定義する必要があります。

2.6.3.204 WOLFSSL_MODE_AUTO_RETRY_ATTEMPTS 無限リトライループの可能性を制限するために使用されます。

2.6.3.205 WOLFSSL_MULTICAST DTLS マルチキャスト機能です。

2.6.3.206 WOLFSSL_MULTICAST_PEERS 最大許容 100 ピアとして定義されたマルチキャスト機能です。

2.6.3.207 WOLFSSL_NAMES_STATIC Position Independent Code (PIC) のための静的 ECC 構造体を使用します。

2.6.3.208 WOLFSSL_SEND_HRR_COOKIE DTLS 1.3 で使用される TLS 拡張機能です。

2.6.3.209 WOLFSSL_SEP 機能証明書ポリシーセット拡張です。

2.6.3.210 WOLFSSL_SESSION_ID_CTX アプリケーションセッションコンテキスト ID をコピーするために使用されます。

2.6.3.211 WOLFSSL_SESSION_TIMEOUT 秒単位のデフォルトセッション再開キャッシュタイムアウトは、タイムアウトを手動で定義するために使用されます。

2.6.3.212 KEEP_OUR_CERT SSL 証明書を返す能力を確保するために使用されます。

2.6.3.213 KEEP_PEER_CERT ピア証明書を保持します。OpenSSL 互換性レイヤーの一部はピア証明書が必要とします。

2.6.3.214 WOLFSSL_SIGNER_DER_CERT これは署名に使用される DER/ASN.1 の保持を可能にします。これは wolfSSL_X509_STORE_get1_certs など、互換性レイヤーで使用されます。

2.6.3.215 CA_TABLE_SIZE wolfSSL 証明書マネージャー署名者テーブルで使用されます。デフォルトの CA_TABLE_SIZE は 11 ですが、実際のニーズに基づいて調整できます。各 WOLFSSL_CTX には独自の証明書マネージャー (CM) があります。

2.6.3.216 ECDHE_SIZE コンパイル時にこれをオーバーライドできるようにするために定義します。ECDHE サーバーサイズはデフォルトで 256 ビットであり、これは事前に決められた ECDHE 曲線サイズを設定できます。デフォルトは 32 バイトです。

2.6.3.217 CIPHER_NONCE 認証に実装される暗号化番号として使用されます。これは擬似乱数であり、整合性のみの暗号スイートです。

2.6.3.218 WOLFSSL_USE_POPEN_HOST wolfio.c ソケットオープンコードでホストとポストでソケットを作成するために popen を使用します。CRL と OCSP で使用されます。

2.6.3.219 CloseSocket ソケットを閉じるために使用される関数をオーバーライドする方法です。CRL、OCSP、BIO で使用されます。

2.6.3.220 CONFIG_POSIX_API ネットワーキングシステムコール用の POSIX 名を有効にします。

2.6.3.221 WOLFSSL_USER_CURRTIME マクロ WOLFSSL_USER_CURRTIME を使用して gettimeday なしで test.h で使用するオプションを追加します。

2.6.3.222 WOLFSSL_USER_MUTEX ユーザー定義ミューテックスのオプションです。

2.6.3.223 DEFAULT_MIN_ECCKEY_BITS ECCKey の最小ビット数を識別します。

2.6.3.224 DEFAULT_MIN_RSAKEY_BITS RSA キーの最小ビット数を識別します。

2.6.3.225 EXTERNAL_SERIAL_SIZE バッファに符号なしバイナリで X509 シリアル番号を書き込む生のシリアル番号バイトです。すべての場合において、バッファは少なくとも EXTERNAL_SERIAL_SIZE (32) である必要があります。成功した場合、WOLFSSL_SUCCESS を返します。

注意：これはユーザーが定義できない内部マクロです。

2.6.3.226 LARGE_STATIC_BUFFERS 組み込みコールバックには大きな静的バッファが必要です。16K までの大きなバッファを有効にするオプションを提供してください。

2.6.3.227 LIBWOLFSSL_VERSION_STRING これは wolfSSL バージョン文字列であり、リリースバンドルまたは ./configure が実行されたときに入力されます。また、LIBWOLFSSL_VERSION_HEX にはこの 32 ビット HEX バージョンもあります。これらは wolfSSL/version.h から来ています。

2.6.4 wolfSSL のカスタマイズ及び移植

2.6.4.1 WOLFSSL_USER_SETTINGS 定義されている場合、ユーザー固有の設定ファイルを使用できます。ファイルには user_settings.h と名前が付けられ、インクルートパスに存在する必要があります。これは、標準の settings.h ファイルの前に含まれるため、デフォルト設定をオーバーライドできます。

2.6.4.2 WOLFSSL_CALLBACKS デバッガがない環境でシグナルを使用するデバック用のコールバックの利用を可能にする拡張機能です。デフォルトではオフです。ブロッキングソケットを使用してタイマーを設定するためにも使用できます。詳しくは第 6 章 コールバックをご覧ください。

2.6.4.3 WOLF_CRYPT_CB 暗号コールバックサポートを有効にします。この機能は、**--enable-cryptocb** を使用すると自動的に有効になります。

2.6.4.4 WOLFSSL_DYN_CERT WOLFSSL_NO_MALLOC が設定されていても、証明書を解析するときに subjectCN および publicKey フィールドの割り当てを許可します。RSA 証明書で WOLFSSL_NO_MALLOC オプションを使用する場合、ピアの証明書で証明書を検証するために、CA の公開鍵を保持する必要があります。ca->publicKey が NULL であるため、これは ConfirmSignature エラー -173 BAD_FUNC_ARG として表示されます。

2.6.4.5 WOLFSSL_USER_IO ユーザーがデフォルトの I/O 関数 **EmbedSend()** および **EmbedReceive()** の自動設定を削除できます。カスタム I/O 抽象化レイヤに使用されます (詳細については第 5 章「抽象化レイヤ」節を参照)。

2.6.4.6 NO_FILESYSTEM `stdio`(標準入出力関数) が使用できないために、証明書とキーファイルをロードできない場合に使用されます。これにより、ファイルの代わりにバッファを使用できます。

2.6.4.7 NO_INLINE 頻繁に使用されるライン数の少ない関数の自動インライン化を無効にします。この定義により、wolfSSL が遅くなり、実際にはこれらが小さな関数であるため、通常は関数呼び出し/ Return よりもはるかに小さくなります。autoconf を使用していない場合は、コンパイル済みファイルのリストに `wolfcrypt/src/misc.c` を追加する必要があります。

2.6.4.8 NO_DEV_RANDOM デフォルトの `/dev/random` 乱数ジェネレーターの使用を無効にします。定義されている場合、ユーザーは OS 固有の `GenerateSeed()` 関数 (`wolfcrypt/src/random.c` で見つけた) を記述する必要があります。

2.6.4.9 NO_MAIN_DRIVER 通常のビルド環境で使用されて、テストアプリケーションが独自に呼び出されるか、テストスイートドライバアプリケーションを介して呼び出されます。`test.c`、`client.c`、`server.c`、`echoclient.c`、`echoserver.c`、および `testsuite.c` でテストファイルで使用する必要があります。

2.6.4.10 NO_WRITEV `writev()` セマンティクスのシミュレーションを無効にします。

2.6.4.11 SINGLE_THREADED ミューテックスの使用をオフにします。wolfSSL は現在、セッションキャッシュの保護にのみ使用しています。wolfSSL の使用が常に単スレッドからに限定されている場合は、この機能をオンにすることができます。

2.6.4.12 USER_TICKS `time(0)` の使用が可能でない場合、ユーザーは自分のクロックチェック関数を定義できます。カスタム機能には秒単位の正確さが必要ですが、エポックと相関がある必要はありません。`wolfSSL_int.c` の機能を参照してください。

2.6.4.13 USER_TIME ユーザーが自身で定義した構造体を使用する (または必要とする) 場合の `time.h` 構造体の使用を無効にします。実装の詳細については `wolfcrypt/src/asn.c` を参照してください。ユーザーは `XTIME()`、`XGMTIME()`、および `XVALIDATE_DATE()` を定義および/または実装する必要があります。

2.6.4.14 USE_CERT_BUFFERS_256 `<wolfSSL_root>/wolfSSL/certs_test.h` にある 256 ビットのテスト用証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.6.4.15 USE_CERT_BUFFERS_1024 `<wolfSSL_root>/wolfSSL/certs_test.h` にある 1024 ビットのテスト用証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.6.4.16 USE_CERT_BUFFERS_2048 `<wolfSSL_root>/wolfSSL/certs_test.h` にある 2048 ビットテスト証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.6.4.17 USE_CERT_BUFFERS_3072 `<wolfSSL_root>/wolfSSL/certs_test.h` にある 3072 ビットのテスト用証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.6.4.18 USE_CERT_BUFFERS_4096 <wolfSSL_root>/wolfSSL/certs_test.h にある 4096 ビットのテスト用証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.6.4.19 USE_CERT_BUFFERS_25519 <wolfSSL_root>/wolfSSL/certs_test.h にある Ed25519 のテスト用証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.6.4.20 USE_WOLFSSL_IO このマクロは send/recv コールバックを有効にします。使用例は[こちら](#)で見ることができます。

2.6.4.21 CUSTOM_RAND_GENERATE_SEED ユーザーが wc_GenerateSeed(byte* output, word32 sz) に相当するカスタム機能を定義できるようにします。

2.6.4.22 CUSTOM_RAND_GENERATE_BLOCK ユーザーがカスタム乱数生成機能を定義できるようにします。使用例は以下の通りです。

```
./configure --disable-hashdrbg
CFLAGS="-DCUSTOM_RAND_GENERATE_BLOCK= custom_rand_generate_block"
/* RNG */
/* #define HAVE_HASHDRBG */
extern int custom_rand_generate_block(unsigned char* output, unsigned int sz);
```

2.6.4.23 NO_PUBLIC_GCM_SET_IV 独自のカスタムハードウェアポートを作成していて、wc_AesGcmSetIV() の公開実装が提供されていない場合は、これを使用してください。

2.6.4.24 NO_PUBLIC_CCM_SET_NONCE 独自のカスタムハードウェアポートを作成していて、wc_AesGcmSetNonce() の公開実装が提供されていない場合は、これを使用してください。

2.6.4.25 NO_GCM_ENCRYPT_EXTRA 独自のカスタムハードウェアポートを行っていて、wc_AesGcmEncrypt_ex() の実装が提供されていない場合は、これを使用してください。

2.6.4.26 WOLFSSL_STM32[F1 | F2 | F4 | F7 | L4] 適切な STM32 デバイス用にビルドするときにこれらの定義のいずれかを使用します。必要に応じて、[wolfSSL ポーティングガイド](#)を確認し wolfSSL-root/wolfSSL/wolfcrypt/settings.h を更新します。

2.6.4.27 WOLFSSL_STM32_CUBEMX Cubemx ツールを使用してハードウェア抽象化レイヤー (HAL) API を生成する場合、この設定を使用して wolfSSL に適切なサポートを追加します。

2.6.4.28 WOLFSSL_CUBEMX_USE_LL Cubemx ツールを使用して API を生成する場合、HAL (ハードウェア抽象化層) または低層 (LL) の 2 つのオプションがあります。この定義を使用して、WOLFSSL_STM32[F1/F2/F4/F7/L4] の wolfSSL-root/wolfSSL/wolfcrypt/settings.h に含まれるヘッダーを制御します。

2.6.4.29 NO_STM32_CRYPT STM32 のハードウェア暗号サポートを提供しない場合に定義します。

2.6.4.30 NO_STM32_HASH STM32 のハードウェアハッシュサポートを提供しない場合に定義します。

2.6.4.31 NO_STM32_RNG STM32 のハードウェア RNG サポートを提供しない場合のために定義します。

2.6.4.32 XTIME_MS TLS 1.3 を使用するとき使用するための関数をミリ秒単位でマッピングするためのマクロです。

使用例

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

2.6.4.33 WOLFSSL_CIPHER_TEXT_CHECK TLS 接続中の AES 暗号化操作に対する可能なグリッチ攻撃をチェックするためにこれを定義します。

2.6.4.34 RTTHREAD RT-THREAD マクロは、rtthread IoT を wolfSSL にポーティングする際に使用されます。

2.6.4.35 SO_REUSEPORT ローカルアドレスとポートの再利用を許可します。

2.6.4.36 INTIME_RTOS INtime RTOS 用のポート設定です。

2.6.4.37 WOLFSSL_SGX SGX へのポーティング時に使用します。

2.6.5 メモリまたはコードの使用量の削減

2.6.5.1 TFM_TIMING_RESISTANT スタックサイズが小さいシステムで Fast Math(**USE_FAST_MATH**)を使用するとき定義できます。これにより大きな静的配列が削除されます。

2.6.5.2 ECC_TIMING_RESISTANT サイドチャネルと差分電力分析 (DPA) 攻撃を防ぐために `ecc.c` でコードを有効にするタイミング耐性機能として使用されます。

2.6.5.3 FUSION_RTOS Fusion RTOS 実装は、チケットが最初に見られた時と送信された時の違いを表すために使用されます。32 ビット値としてミリ秒単位の時間を返します。

2.6.5.4 WOLFSSL_SMALL_STACK スタックサイズが小さいデバイスに使用できます。これにより、`wolfcrypt/src/integer.c` の動的メモリの使用が増加しますが、パフォーマンスが遅くなる可能性があります。

2.6.5.5 ALT_ECC_SIZE Fast Math と RSA/DH を使用する場合は、ECC メモリ消費量を削減するためにこれを定義できます。ECC ポイントにスタックを使用する代わりに、ヒープから割り当てます。

2.6.5.6 ECC_SHAMIR ECC Math のバリエーションは、わずかに高速ですが、ヒープの使用量を 2 倍にします。

2.6.5.7 RSA_LOW_MEM 定義された場合、CRT は使用されていないため、一部のメモリを保存しますが、RSA 操作を遅くします。デフォルトではオフになっています。

2.6.5.8 WOLFSSL_SHA3_SMALL SHA3 を有効の場合、このマクロはビルドサイズを縮小します。

2.6.5.9 WOLFSSL_SMALL_CERT_VERIFY DecodedCert を使用せずに証明書署名を確認します。一部のコードでは2倍になりますが、ピークヒープメモリの使用が小さくなります。**WOLFSSL_NONBLOCK_OCSP**では使用できません。

2.6.5.10 GCM_SMALL テーブルを使用する代わりに実行時に計算することで、AES GCM コードサイズを減らすオプションです。可能なオプションは、GCM_SMALL,GCM_WORD32,GCM_TABLE です。

2.6.5.11 CURVED25519_SMALL **CURVE25519_SMALL**および**ED25519_SMALL**を定義します。

2.6.5.12 CURVE25519_SMALL CURVE25519 のスモールメモリオプションです。これはより少ないメモリを使用しますが、遅いです。

2.6.5.13 ED25519_SMALL ED25519 のスモールメモリオプションです。これはより少ないメモリを使用しますが、遅いです。

2.6.5.14 USE_SLOW_SHA ローリンググループを使用しないことでコードサイズを縮小します。これにより、SHA のパフォーマンスが低下します。

2.6.5.15 USE_SLOW_SHA256 ローリンググループを使用しないことでコードサイズを縮小します。これにより、SHA のパフォーマンスが低下します。約 2k 小さくできますが、約 25%遅くなります。

2.6.5.16 USE_SLOW_SHA512 ローリンググループを使用しないことでコードサイズを縮小します。これにより、SHA のパフォーマンスが低下します。2 倍以上小さくできますが、50%が遅くなります。

2.6.5.17 ECC_USER_CURVES ユーザーが有効になっている ECC カーブサイズを選択できるようにします。デフォルトでは 256 ビットカーブのみが有効になっています。他の曲線を使用できるようにするには、HAVE_ECC192,HAVE_ECC224 などを使用します。

2.6.5.18 WOLFSSL_SP_NO_MALLOC SP コードでは常に Stack を使用して、Heap `xmalloc()`/`xrealloc()`/`xfree()` 呼び出しは行われません。

2.6.5.19 WOLFSSL_SP_NO_DYN_STACK 動的スタックアイテムの使用を無効にします。コードサイズが小さく、小さなスタックではなく使用されます。

2.6.5.20 WOLFSSL_SP_FAST_MODEXP コードサイズを犠牲にして、より高速な `mod_exp` 実装でコンパイルします。

2.6.5.21 WC_DISABLE_RADIX_ZERO_PAD 16 進文字列出力で先頭ゼロの印刷を無効にします。

例えば、値 8 は通常「0x08」と表示されますが、このマクロが定義されている場合、「0x8」と表示されます。このマクロを定義すると、コードサイズを削減できます。

2.6.5.22 WC_ASN_NAME_MAX X.509 証明書フィールドの最大名前サポートのオーバーライドを許可します。

2.6.5.23 OPENSSL_EXTRA_X509_SMALL 証明書の特別な小さな OpenSSL 互換レイヤーです。

2.6.6 パフォーマンスを向上させる

2.6.6.1 USE_INTEL_SPEEDUP AES、Chacha20、Poly1305、SHA256、SHA512、Ed25519 および Curve25519 の加速に Intel の AVX/AVX2 命令を使用できます。

2.6.6.2 WOLFSSL_AESNI Intel と AMD チップセットに組み込まれている AES アクセラレーション操作を使用できます。この定義を使用する場合、aes_asm.asm(AT & T 構文を備えた Windows 用) または aes_asm.S ファイルは、Intel AES の新しい命令セット (AESNI) を介して最適化するために使用されます。

2.6.6.3 HAVE_INTEL_RDSEED DRBG シードソース用の Intel の RDSEED を有効にします。

2.6.6.4 HAVE_INTEL_RDRAND wolfSSL のランダムソースの Intel の RDRAND 命令を有効にします。

2.6.6.5 FP_ECC ECC 固定点キャッシュを有効にします。これにより、同じ秘密鍵に対する繰り返し操作が高速化されます。FP_ENTRIES および FP_LUT を使用してエントリと LUT ビットの数と定義して、デフォルトの静的メモリ使用量を削減することもできます。

2.6.6.6 WOLFSSL_ASYNC_CRYPT これにより、Intel QuickAssist や Marvell(Cavium)Nitrox V などのハードウェアベースのアダプターを使用した非同期暗号化のサポートが可能になります。非同期コードは公開コードに含まれていません。評価のために必要でしたら、info@wolfssl.jp までお問い合わせください。

2.6.6.7 WOLFSSL_NO_ASYNC_IO これは非同期 I/O ネットワーキングを無効にします。非同期 I/O はデフォルトでオンになっており、ハンドシェイクプロセス中に最大約 140 バイトを占めることがあります。ネットワークインターフェースが書き込み時に SOCKET_EWOULDBLOCK または SOCKET_EAGAIN (またはカスタム I/O コールバックの場合は WOLFSSL_CBIO_ERR_WANT_WRITE) を返さない場合、WOLFSSL_NO_ASYNC_IO を定義して、wolfSSL がハンドシェイクメッセージを構築している間に状態を保存しないようにできます。

2.6.7 GCM パフォーマンスチューニング

GCM パフォーマンスには 4 つのバリエーションがあります。

- GCM_SMALL - 最小のフットプリント、最も遅い (FIPS 検証済み)
- GCM_WORD32 - 中程度 (FIPS 検証済み)
- GCM_TABLE - 高速 (FIPS 検証済み)
- GCM_TABLE_4BIT - 最速 (FIPS 検証済み)

2.6.8 wolfSSL の数学ライブラリオプション

wolfSSL には 3 つの数学ライブラリがあります。

- Big Integer
- Fast Math
- Single Precision Math

wolfSSL をビルドするときは、これらの 1 つだけを使用する必要があります。

Big Integer Library は最も移植性の高いオプションで、アセンブリなしで C 言語で書かれています。そのため、特定のアーキテクチャに最適化されていません。すべての数学変数はヒープ上でインスタンス化されます。スタック使用量は最小限です。残念ながら、Big Integer Library はタイミング耐性がありません。

Fast Math Library は良いオプションです。C とアセンブリの両方を使用して実装されています。そのため、特定のアーキテクチャに最適化されています。すべての数学変数はスタック上でインスタンス化されます。

ヒープ使用量は最小限です。TFM_TIMING_RESISTANT マクロが定義されている場合、タイミング耐性を持たせることができます。このライブラリは FIPS 140-2 および 140-3 認証を取得しています。

Single Precision (SP) Math Library は推奨ライブラリです。C とアセンブリの両方を使用して実装されています。そのため、特定のアーキテクチャに最適化されています。すべての数学変数はスタック上でインスタンス化されます。ヒープ使用量は最小限です。常にタイミング耐性があります。一般的にはコードサイズを犠牲にして速度が最適化されていますが、不要なコードをコンパイルしないように高度に構成可能です。このライブラリは DO-178C 認証を取得しています。

2.6.8.1 Big Integer ライブラリ (廃止予定) このライブラリは 2023 年末までに wolfSSL/wolfCrypt ライブラリから廃止および削除される予定です。必要に応じて、`--enable-heapmath` または `CFLAGS=-DUSE_INTEGER_HEAP_MATH` で有効にすることができます。

パブリックドメインの LibTomMath ライブラリからフォークされました。LibTomMath の詳細については、<https://www.libtom.net/LibTomMath/> を参照してください。私たちのフォークは元のパブリックドメインコードよりもかなり活発でセキュアであることに注意してください。

これは一般的に最も移植性が高く、使い始めるのが最も簡単です。通常の big integer ライブラリのマイナス点は、より遅く、すべてのメモリがヒープから割り当てられるため多くのヒープメモリを使用し、`XREALLOC()` 実装を必要とし、タイミング耐性がないことです。実装は `integer.c` にあります。

2.6.8.2 Fast Math

2.6.8.2.1 USE_FAST_MATH パブリックドメインの LibTomFastMath ライブラリからフォークされました。LibTomFastMath の詳細については、<https://www.libtom.net/TomsFastMath> を参照してください。私たちのフォークは元の LibTomFastMath のパブリックドメインコードよりもかなり活発でセキュアであることに注意してください。私たちはパフォーマンス、セキュリティ、コード品質を向上させました。また、FastMath コードは FIPS 140-2 および 140-3 認証を取得しています。

FastMath ライブラリは可能であればアセンブリを使用し、RSA、DH、DSA などの非対称秘密/公開鍵操作を高速化します。アセンブリの組み込みはコンパイラとプロセッサの組み合わせに依存します。一部の組み合わせでは追加の構成フラグが必要になり、一部は不可能な場合もあります。新しいアセンブリルーチンで FastMath を最適化するための支援はコンサルティングベースで利用可能です。アーキテクチャ固有の最適化を参照してください。

FastMath では、すべてのメモリはスタック上に割り当てられます。FastMath を使用する場合、スタックメモリの使用量が大きくなる可能性があるため、このオプションを使用する場合は `TFM_TIMING_RESISTANT` も定義することをお勧めします。TFM_TIMING_RESISTANT が定義されている場合、FastMath コードはタイミング耐性があります。これにより、一定時間のための大きな数学ウィンドウが減少し、メモリ使用量が少なくなります。秘密鍵操作中にショートカットが少なく、したがって分岐が少ないため、スタックの使用量が少なくなります。これにより、タイミング攻撃が実際の脅威であり、悪意のある第三者に秘密鍵を複製するのに十分な情報を与える可能性があるため、実装がより安全になります。

例えば、ia32 では、すべてのレジスタが利用可能である必要があるため、高い最適化とフレームポインタの省略に注意する必要があります。wolfSSL はデバッグビルド以外では GCC に `-O3 -fomit-frame-pointer` を追加します。異なるコンパイラを使用している場合は、これらを構成中に手動で `CFLAGS` に追加する必要があるかもしれません。

macOS では、`CFLAGS` に `-mdynamic-no-pic` も追加する必要があります。さらに、OS X で ia32 の共有モードでビルドしている場合は、`LDFLAGS` にもオプションを渡す必要があります。

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

これにより、一部のシンボルに対してエラーではなく警告が表示されます。

FastMath は動的およびスタックメモリの使用方法も変更します。通常の数学ライブラリは大きな整数に動的メモリを使用します。FastMath はデフォルトで 4096 ビットの整数を保持する固定サイズのバッファを使用し、2048 ビットで 2048 ビットの乗算を可能にします。4096 ビットで 4096 ビットの乗算が必要

な場合は、wolfSSL/wolfcrypt/tfm.h の FP_MAX_BITS を変更してください。FP_MAX_BITS が増加すると、公開鍵操作で使用するバッファがより大きくなるため、実行時のスタック使用量も増加します。FP_MAX_BITS は最大キーサイズの 2 倍である必要があります。例えば、最大のキーが 2048 ビットの場合、FP_MAX_BITS は 4096 であるべきで、4096 ビットの場合は FP_MAX_BITS は 8192 であるべきです。ECC のみを使用する場合、これは最大 ECC キーサイズの 2 倍に減らすことができます。ライブラリのいくつかの関数は複数の一時的な大きな整数を使用するため、スタックは比較的大きくなる可能性があります。これは、スタックサイズが低い値に設定されている組み込みシステムやスレッド環境でのみ問題になるはずですが、FastMath を使用している環境で公開鍵操作中にスタック破壊が発生した場合は、スタック使用量に対応するためにスタックサイズを増やしてください。

autoconf システムを使用せずに FastMath を有効にする場合は、USE_FAST_MATH を定義し、tfm.c を wolfSSL ビルドに追加し、integer.c を削除する必要があります。ALT_ECC_SIZE を定義すると、ECC ポイントはスタックではなくヒープからのみ割り当てられます。

2.6.8.2.2 アーキテクチャ固有の最適化 USE_FAST_MATH でアセンブリの最適化のために以下のマクロを定義できます。

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

これらのいずれも定義されていないか、TFM_NO_ASM が定義されている場合、TFM_ISO が定義され、ISO C ポータブルコードが使用されます。

2.6.8.2.3 アルゴリズム固有の最適化 有効にすると、それぞれの ECC 曲線に対して乗算と二乗計算の最適化された実装が使用されます。

- TFM_ECC192
- TFM_ECC224
- TFM_ECC256
- TFM_ECC384
- TFM_ECC521

2.6.8.2.4 TFM_SMALL_SET 小さい数の乗算のためのスピード最適化を行います。1-16 ワードのコンバ (Comba) 乗算と二乗の実装を含みます。ECC 操作の性能を向上させるのに役立ちます。

2.6.8.2.5 TFM_HUGE_SET より大きな数の乗算のための速度最適化を行います。20、24、28、32、48、64 ワードコンバ (Comba) 乗算と、ビットサイズが許す場所での二乗の実装が含まれています。RSA/DH/DSA 操作のパフォーマンスの向上に役立ちます。

2.6.8.2.6 TFM_SMALL_MONT_SET Intel アーキテクチャ上の小さな数値のモンゴメリーリダクションのための速度最適化を行います。1~16 ワードモンゴメリーリダクションの実装が含まれています。ECC 操作の性能を向上させるのに役立ちます。

2.6.8.3 独自の単精度 (SP) 数学ライブラリのサポート SP 数学ライブラリは推奨されるデフォルトのオプションであり、DO-178C 認定を受けています。このライブラリを使用すると、特定の鍵サイズと一般的な曲線に対する公開鍵の操作が高速化されます。次のような正しいコードファイルが含まれていることをご確認ください。

- `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 単精度演算ライブラリのサポートを有効にします。

2.6.8.4 WOLFSSL_SP_MATH SP 数学とアルゴリズムのみを有効にします。通常 (`integer.c`) または `FAST(tfm.c)` などの大きな整数演算コードを排除します。鍵サイズと曲線を SP でサポートされているものだけに制限します。

2.6.8.5 WOLFSSL_SP_MATH_ALL SP 数学とアルゴリズムを有効にします。SP ではサポートされていない鍵サイズと曲線のために、通常 (`integer.c`) または `FAST(tfm.c)` などの大きな整数数学コードを実装しています。

2.6.8.6 WOLFSSL_SP_SMALL SP Math を使用する場合、これはコードの小さなバージョンを使用し、大きなスタック変数を回避します。

2.6.8.6.1 SP_WORD_SIZE 1 ワードを 1 ワードに保存するための 32 ビットまたは 64 ビットのデータ型。

2.6.8.6.2 WOLFSSL_SP_NONBLOCK 単一の精度数学の「非ブロッキング」モードを有効にします。これにより、長い操作のために `FP_WouldBlock` を返し、機能が完了するまで再度呼び出す必要があります。現在、これは ECC でのみサポートされており、`WC_ECC_NONBLOCK` と組み合わせて使用されています。

2.6.8.6.3 WOLFSSL_SP_FAST_NCT_EXPTMOD より速い非一定の時間モジュール式指数の実装を可能にします。公開鍵操作にのみ使用されます。秘密鍵操作ではありません。

2.6.8.6.4 WOLFSSL_SP_INT_NEGATIVE マルチプレシジョン数値を負にすることができます。(暗号化操作には必要ありません。)

2.6.8.6.5 WOLFSSL_SP_INT_DIGIT_ALIGN `sp_int_digit` ポインタの非整列アクセスが許可されていない場合に有効にします。

2.6.8.6.6 WOLFSSL_HAVE_SP_RSA 2048、3072、4096 ビットのための単精度 RSA。

2.6.8.6.7 WOLFSSL_HAVE_SP_DH 2048、3072、4096 ビットの単精度 DH。

2.6.8.6.8 WOLFSSL_HAVE_SP_ECC SECP256R1 および SECP384R1 用の単精度 ECC。

2.6.8.6.9 WOLFSSL_SP_LARGE_CODE 大きなバイナリーサイズとなりますが、単精度 (SP) スピードアップを許可します。一部の組み込みプラットフォームには適していない可能性があります。

2.6.8.6.10 WOLFSSL_SP_DIV_WORD_HALF 2 倍の長さのワードを使用して除算が利用できないことを示します。たとえば、32 ビット CPU で、ライブラリから 64 ビット除算でコンパイルしたくない場合は、このマクロを定義すると、半分のワードサイズを使用して除算が行われる実装を有効化します。

2.6.8.6.11 WOLFSSL_SP_DIV_32 32 ビット除算が利用できず、wolfSSL が独自の単精度 (SP) 実装を使用する必要があることを示します。

2.6.8.6.12 WOLFSSL_SP_DIV_64 64 ビット除算が利用できず、WOLFSSL は独自の単精度 (SP) 実装を使用する必要があることを示します。

2.6.8.6.13 WOLFSSL_SP_ASM より高速な単一精度 (SP) プラットフォーム固有のアセンブリコードの実装を有効にします。プラットフォームが検出されます。

2.6.8.6.14 WOLFSSL_SP_X86_64_ASM 単精度 (SP) Intel x64 アセンブリの実装を有効にします。

2.6.8.6.15 WOLFSSL_SP_ARM32_ASM 単精度 (SP) Aarch32 アセンブリの実装を有効にします。

2.6.8.6.16 WOLFSSL_SP_ARM64_ASM 単精度 (SP) Aarch64 アセンブリの実装を有効にします。

2.6.8.6.17 WOLFSSL_SP_ARM_CORTEX_M_ASM 単精度 (SP) Cortex-M ファミリー (Cortex-M4 を含む) アセンブリの実装を有効にします。

2.6.8.6.18 WOLFSSL_SP_ARM_THUMB_ASM 単精度 (SP) ARM Thumb アセンブリの実装を有効にします (-mthumb と一緒に使用)。

2.6.8.6.19 WOLFSSL_SP_X86_64 単精度 (SP) Intel X86 64 ビットアセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.6.8.6.20 WOLFSSL_SP_X86 単精度 (SP) Intel X86 アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.6.8.6.21 WOLFSSL_SP_PPC64 単精度 (SP) PPC64 アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.6.8.6.22 WOLFSSL_SP_PPC 単精度 (SP) PPC アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.6.8.6.23 WOLFSSL_SP_MIPS64 単精度 (SP) MIPS64 アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.6.8.6.24 WOLFSSL_SP_MIPS 単精度 (SP) MIPS アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.6.8.6.25 WOLFSSL_SP_RISCV64 単精度 (SP) RISCV64 アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.6.8.6.26 WOLFSSL_SP_RISCV32 単精度 (SP) RISCv32 アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.6.8.6.27 WOLFSSL_SP_S390X 単精度 (SP) S390X アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.6.9 スタックまたはチップ固有の定義

wolfSSL は、さまざまなプラットフォームと TCP/IP スタック用にビルドできます。次の定義のほとんどは wolfSSL/wolfcrypt/settings.h にあり、デフォルトでコメントアウトされています。以下を参照する特定のチップまたはスタックのサポートを有効にするために、それぞれのコメントを解除できます。

2.6.9.1 IPHONE iOS で使用するためにビルドする場合に定義します。

2.6.9.2 THREADX ThreadX RTOS(<https://www.rtos.com>) で使用するためにビルドするときに定義します。

2.6.9.3 MICRIUM Micrium の μ C/OS-III RTOS(<https://www.micrium.com>) で使用するためにビルドするときに定義します。

2.6.9.4 MBED mbed プロトタイピングプラットフォーム (<https://www.mbed.org>) で使用するためにビルドするときに定義します。

2.6.9.5 MICROCHIP_PIC32 マイクロチップの PIC32 プラットフォーム (<https://www.microchip.com>) で使用するためにビルドするときに定義します。

2.6.9.6 MICROCHIP_TCPIP_V5 マイクロチップ TCP/IP スタックのバージョン 5 を定義できます。

2.6.9.7 MICROCHIP_TCPIP マイクロチップ TCP/IP スタックバージョン 6 以降に定義できます。

2.6.9.8 WOLFSSL_MICROCHIP_PIC32MZ PIC32MZ ハードウェア暗号化エンジン用に定義できます。

2.6.9.9 FREERTOS Freertos(<https://www.freertos.org>) で使用するためにビルドするときに定義できます。LwIP を使用している場合は、WOLFSSL_LWIP も定義します。

2.6.9.10 FREERTOS_WINSIM Freertos Windows Simulator(<https://www.freertos.org>) で使用するためにビルドするときに定義できます。

2.6.9.11 WOLFSSL_CHIBIOS ChibiOS RTOS で使用するためにビルドする場合に定義します。

2.6.9.12 WOLFSSL_CMSIS_RTOS Mbed CMIS-RTOS で使用するためにビルドする場合に定義します。

2.6.9.13 WOLFSSL_CMSIS_RTOSv2 Mbed CMIS-RTOSv2 で使用するためにビルドする場合に定義します。

2.6.9.14 WOLFSSL_LWIP_NATIVE LWIP ネイティブのプラットフォームで使します。

2.6.9.15 WOLFSSL_DEOS この定義を使用して、[ここ](#)で利用可能な Deos RTOS の wolfSSL サポートを有効にできます。

2.6.9.16 WOLFSSL_ESPIDF ESP-IDF でビルドする場合に使用します。

2.6.9.17 WOLFSSL_LINUXKM Linux カーネルモジュール用にビルドする場合に使用します。

2.6.9.18 WORD64_AVAILABLE 64 ビット型がサポートされていることを示す移植用マクロです。通常は `sizeof_long_long` 8 を使用する方が良いです。

2.6.9.19 WOLFSSL_NUCLEUS_1_2 Nucleus 1.2 でビルドする場合に使用します。

2.6.9.20 WOLFSSL_PICOTCP PicoTCP でビルドする場合に使用します。

2.6.9.21 WOLFSSL_RENESAS_RA6M3G RENESAS RA6M3G でビルドする場合に使用します。

2.6.9.22 WOLFSSL_RENESAS_RA6M4 RENESAS RA6M4 でビルドする場合に使用します。

2.6.9.23 WOLFSSL_RIOT_OS RIOT-OS でビルドする場合に使用します。

2.6.9.24 WOLFSSL_uITRON4 uITRON4 用にビルドする場合に使用します。

2.6.9.25 WOLFSSL_uTKERNEL2 uT-Kernel でビルドする場合に使用します。

2.6.9.26 WOLFSSL_VXWORKS VxWorks でビルドする場合に使用します。

2.6.9.27 DEVKITPRO devkitPro 用にビルドする場合に使用します。

2.6.9.28 WOLFSSL_VXWORKS_6_x VxWorks 6.x 専用の実装でのみ使用されます。

2.6.9.29 WOLFSSL_WICED WICED Studio 用にビルドする場合に使用されます。

2.6.9.30 FREESCALE_KSDK_FREERTOS このマクロの古い名前は `FREESCALE_FREE_RTOS` です。Freescale KSDK FreeRTOS 用にビルドする場合に使用されます。

2.6.9.31 FREESCALE_KSDK_MQX Freescale KSDK MQX/RTCS/MFS 用にビルドする場合に使用されま

す。

2.6.9.32 FREESCALE_MQX_5_0 Freescale Classic MQX バージョン 5.0 用にビルドする場合に使用され

ます。

2.6.9.33 WOLFSSL_KEIL_TCP_NET TCP スタック (`MDK_CONF_NETWORK`) を構成します。デフォルトでは Keil TCP `WOLFSSL_KEIL_TCP_NET` を使用します。なしの場合は 0、ユーザー IO コールバックの場合は 2 を使用します。

2.6.9.34 INTEL_GALILEO Arduino と wolfSSL を構成する際に使用されます。Intel Galileo プラットフォーム用にビルドする場合は `#define INTEL_GALILEO` を追加します。

2.6.9.35 HAVE_KEIL_RTX MDK-RTX-TCP-FS 構成用の wolfSSL。

2.6.9.36 EBSNET EBSNET 製品と RTIP を使用するとき定義できます。

2.6.9.37 WOLFSSL_EMBOS SEGGER embOS (<https://www.segger.com/products/rtos/embos/>) のビルド時に定義できます。emNET を使用する場合は、**WOLFSSL_EMNET** も定義します。

2.6.9.38 WOLFSSL_EMNET SEGGER emNET TCP/IP スタック (<https://www.segger.com/products/connectivity/emnet/>) のビルド時に定義できます。

2.6.9.39 WOLFSSL_LWIP LWIP TCP/IP スタック (<https://savannah.nongnu.org/projects/lwip/>) で wolfSSL を使用するとき定義できます。

2.6.9.40 WOLFSSL_ISOTP 通常、CAN バスに使用される ISO-TP トランスポートプロトコルで wolfSSL を使用する場合、定義できます。使用例は [wolfssl-examples リポジトリ](#) にあります。

2.6.9.41 WOLFSSL_GAME_BUILD ゲームコンソールのために wolfSSL をビルドするとき定義できます。

2.6.9.42 WOLFSSL_LSR LSR 用にビルドする場合は定義できます。

2.6.9.43 FREESCALE_MQX Freescale MQX/RTCS/MFS(<https://www.freescale.com>) 用にビルドするときに定義できます。これにより、FREESCALE_K70_RNGA が定義され、Kinetis H/W 乱数ジェネレーターアクセラレータのサポートが可能になります。

2.6.9.44 WOLFSSL_STM32F2 STM32F2(<https://www.st.com/internet/mcu/subclass/1520.jsp>) 用のビルド時に定義できます。これは、wolfSSL で STM32F2 ハードウェア暗号化およびハードウェア RNG サポートを可能にします。

2.6.9.45 COMVERGE Comverge 設定を使用する場合は定義できます。

2.6.9.46 WOLFSSL_QL QL SEP 設定を使用している場合は定義できます。

2.6.9.47 WOLFSSL_EROAD eroad のためにビルドを定義することができます。

2.6.9.48 WOLFSSL_IAR_ARM IAR Ewarm 用にビルドする場合は定義できます。

2.6.9.49 WOLFSSL_TIRTOS TI-RTOS 用のビルド時に定義できます。

2.6.9.50 WOLFSSL_ROWLEY_ARM Rowley CrossWorks でビルドするときに定義できます。

2.6.9.51 WOLFSSL_NRF51 Nordic NRF51 に移植するときに定義できます。

2.6.9.52 WOLFSSL_NRF51_AES NORDIC NRF51 に移植するときに ENCRYPT を AES 128 ECB Encrypt に内蔵 AES ハードウェアを使用するように定義できます。

2.6.9.53 WOLFSSL_CONTIKI Contiki オペレーティングシステムのサポートを有効にするために定義できます。

2.6.9.54 WOLFSSL_APACHE_MYNEWT Apache MyNewt ポート層を有効にするように定義できます。

2.6.9.55 WOLFSSL_APACHE_HTTPD Apache HTTPD Web サーバーのサポートを有効にするために定義できます。

2.6.9.56 ASIO_USE_WOLFSSL wolfSSL を ASIO 互換バージョンとしてビルドするように定義できます。その後、ASIO は B00ST_ASIO_USE_WOLFSSL プリプロセッサ定義に依存します。

2.6.9.57 WOLFSSL_CRYPTOCCELL ARM Cryptocell を使用できるように定義できます。

2.6.9.58 WOLFSSL_SIFIVE_RISC_V RISC-V SiFive/HiFive ポートの使用を有効にするために定義できます。

2.6.9.59 WOLFSSL_MDK_ARM MDK ARM のサポートを追加します。

2.6.9.60 WOLFSSL_MDK5 MDK5 ARM のサポートを追加します。

2.6.10 OS 特有の定義

2.6.10.1 USE_WINDOWS_API UNIX/Linux API に対して、Windows ライブラリ API の使用を指定します。

2.6.10.2 WIN32_LEAN_AND_MEAN Microsoft Win32 Lean と Mean Build のサポートを追加します。

2.6.10.3 FREERTOS_TCP FreeRTOS TCP スタックのサポートを追加します。

2.6.10.4 WOLFSSL_SAFERTOS SafeRTOS のサポートを追加します。

2.7 ビルドオプション

以下は、wolfSSL ライブラリのビルド方法をカスタマイズするために ./configure スクリプトに追加される可能性のあるオプションです。デフォルトでは共有ライブラリとしてのみビルドされ、スタティックライブラリとしてのビルドが無効になっています。これによりビルド時間が半分に短縮されます。必要に応じて、どちらのモードも明示的に無効化または有効化できます。

2.7.1 --enable-debug

wolfSSL デバッグサポートを有効にし、デバッグ情報を含めてコンパイルします。デバッグログを出力するには、このビルドオプションを加えた上でメッセージを stderr に出力するマクロ `DEBUG_WOLFSSL` を定義します。その後 `wolfSSL_Debugging_ON()` を実行するとデバッグログ出力が有効化されます。同様に `wolfSSL_Debugging_OFF()` を実行することで、デバッグログ出力を停止できます。詳細は第 8 章 デバッグを参照してください。

2.7.2 --enable-distro

wolfSSL Distro Build を有効にします。

2.7.3 --enable-singlethread

シングルスレッドモードを有効にし、マルチスレッド保護を無効にします。

シングルスレッドモードを有効にすると、セッションキャッシュのマルチスレッド保護がオフになります。アプリケーションがシングルスレッドであること、またはアプリケーションがマルチスレッドであっても一度に 1 つのスレッドだけがライブラリにアクセスする場合にのみ、シングルスレッドモードを有効にしてください。

2.7.4 --enable-dtls

wolfSSL DTLS のサポートを有効にします。

DTLS サポートを有効にすると、ライブラリのユーザーは TLS および SSL に加えて DTLS プロトコルも使用できるようになります。詳細については、第 4 章「DTLS」節を参照してください。

2.7.5 --disable-rng

RNG のコンパイルと使用を無効にします。

2.7.6 --enable-sctp

wolfSSL DTLS-SCTP サポートを有効にします。

2.7.7 --enable-openssh

OpenSSH 互換ビルドを有効にします。

2.7.8 --enable-apachehttpd

Apache httpd 互換ビルドを有効にします。

2.7.9 --enable-openvpn

OpenVPN 互換ビルドを有効にします。

2.7.10 --enable-opensslextra

追加の OpenSSL API 互換性を有効にし、サイズを増加させます。

OpenSSL Extra を有効にすると、より多くの OpenSSL 互換関数が含まれます。これを指定しないベーシックビルドで、ほとんどの TLS/SSL ニーズ対応できる関数を使用できます。しかし 10~100 の OpenSSL 関数を使用するアプリケーションを移植する場合には、これを有効にすることでよりスムーズに移植できるようになります。wolfSSL OpenSSL 互換レイヤーは現在も継続して開発しています。必要な関数が欠落している場合はお問い合わせください。OpenSSL 互換レイヤーの詳細については、第 13 章 OpenSSL 互換性を参照してください。

2.7.11 --enable-opensslall

wolfSSL がサポートする openssl 互換性レイヤーの OpenSSL API をすべて有効にします。

2.7.12 --enable-maxstrength

最大強度ビルドを有効にし、TLSv1.2-AEAD-PFS 暗号のみを許可します。また、グリッチ検出も有効になります。相互運用性の問題が発生する可能性があるため、これはデフォルトで無効になっています。

2.7.13 --disable-harden

ハードニング、タイミング耐性と RSA ブラインドを無効にします。この機能を無効にするとパフォーマンスが向上する可能性があります。

注意 ハードニングはサイドチャネル攻撃に対する緩和策を提供します。慎重に検討した後にのみこの機能を無効にしてください。

user_settings.h で無効にするには、以下のようにします。

- #define WC_NO_CACHE_RESISTANT
- #define WC_NO_HARDEN
- WC_RSA_BLINDING が定義されている場合、削除する
- ECC_TIMING_RESISTANT が定義されている場合、削除する
- TFM_TIMING_RESISTANT が定義されている場合、削除する

2.7.14 --enable-ipv6

IPv6 のテストを有効にします。wolfSSL 自体は IP 中立です

IPv6 を有効にすると、テストアプリケーションが IPv4 の代わりに IPv6 を使用するように変更されます。wolfSSL 自体は IP 中立で、どちらのバージョンも使用できますが、現在のテストアプリケーションは IP 依存です。

2.7.15 --enable-bump

SSL バンプビルドを有効にします。

2.7.16 --enable-leanpsk

LEAN PSK ビルドを有効にします。

PSK を使用し、ライブラリから多くの機能を削除して非常に小さなビルドを構成します。これを有効にすると、ビルドサイズは約 21kB になります。

2.7.17 --enable-leantls

LEAN TLS 1.2 クライアントのみ（クライアント認証なし）、ECC256、AES128 および SHA256（Shamir なし）をサポートするよう構成します。現時点では、他のビルドオプションと組み合わせて使用することを想定していません。

2.7.18 --enable-bigcache

ビッグセッションキャッシュを有効にします。

ビッグセッションキャッシュを有効にすると、セッションキャッシュが 33 セッションから 20,027 セッションまで増加します。デフォルトのセッションキャッシュサイズ 33 は、一般的な TLS クライアントと組み込みサーバーに適しています。ビッグセッションキャッシュは、1 分あたり約 200 の新しいセッションを処理するような、中程度の負荷がかかるサーバーに適しています。

2.7.19 --enable-hugecache

巨大なセッションキャッシュを有効にします。

巨大なセッションキャッシュを有効にすると、セッションキャッシュサイズが 65,791 セッションに増加します。このオプションは、1 分あたり 13,000(1 秒あたり 200) を越える新規セッションを処理するような、重い負荷のかかるサーバー向けです。

2.7.20 --enable-smallcache

小さなセッションキャッシュを有効にします。

小さなセッションキャッシュを有効にすると、wolfSSL は 6 セッションのみを保存し、RAM 使用量を 500 バイト未満に収めます。これは、デフォルトの RAM 使用量 3kB が大きすぎると感じる埋め込みデバイスに適しています。

2.7.21 --enable-savesession

永続的なセッションキャッシュを有効にします。

このオプションを有効にすることで、アプリケーションが wolfSSL セッションキャッシュをメモリバッファに保持 (保存) して復元できるようになります。

2.7.22 --enable-savecert

永続的な証明書キャッシュを有効にします。

このオプションを有効にすることで、アプリケーションが wolfSSL 証明書キャッシュをメモリバッファに保持 (保存) して復元できるようになります。

2.7.23 --enable-atomicuser

アトミックユーザーレコードレイヤを有効にします。

このオプションを有効にすると、ユーザーアトミックレコードレイヤの処理コールバックがオンになります。これにより、アプリケーションは独自の MAC /暗号化および復号化/検証コールバックを登録することができます。

2.7.24 --enable-pkcallbacks

公開鍵コールバックを有効にします。

このオプションを有効にすると公開鍵コールバックが有効になり、アプリケーションが独自の ECC 署名/検証・RSA 署名/検証・暗号化/復号コールバックを登録できるようになります。

2.7.25 --enable-sniffer

wolfSSL Sniffer のサポートを有効にします。

スニファーマーク (SSL 検査) サポートを有効にすると、SSL トラフィックパケットの収集と、正しいキーファイルを使用したそれらのパケットの復号が可能になります。

現在、スニファーマークは以下の RSA 暗号をサポートしています。

CBC 暗号：

- AES-CBC
- Camellia-CBC
- 3DES-CBC

ストリーム暗号：

- RC4

2.7.26 --enable-aesgcm

AES-GCM サポートを有効にします。

このオプションを有効にすると公開鍵コールバックが有効になり、アプリケーションは独自の ECC 署名/検証と RSA 署名/検証を許可し、コールバックを識別して暗号化/復号できます。

AES-GCM を有効にすると、これらの暗号スイートが wolfSSL に追加されます。wolfSSL は、速度とメモリ消費のバランスを考慮した 4 つの異なる AES-GCM 実装を提供しています。利用可能な場合、wolfSSL は 64 ビットまたは 32 ビット演算を使用します。組み込みアプリケーション向けには、RAM ベースのルックアップテーブル (セッションごとに 8KB) を使用する高速な 8 ビット版があり、これは 64 ビット版と同等の速度を持ちます。また、追加の RAM を消費しない低速な 8 ビット版もあります。--enable-aesgcm configure オプションは、=word32、=table、=small のオプションで変更できます。例: --enable-aesgcm=table。

2.7.27 --enable-aesccm

AES-CCM サポートを有効にします。

2.7.28 --disable-aescbc

AES-CBC をコンパイルアウトするために--disable-aescbcで使用されていました。

AES-GCM は、AES の 8 バイト認証 (CCM-8) を備えた CBC-MAC モードでカウンターを有効にします。

2.7.29 --enable-aescfb

AES-CFB モードサポートをオンにします。

2.7.30 --enable-aesctr

wolfSSL AES-CTR サポートを有効にします。

AES-CTR を有効にすると、カウンタモードが有効になります。

2.7.31 --enable-aesni

wolfSSL Intel AES-NI サポートを有効にします。

AES-NI サポートを有効にすると、AES-NI をサポートするチップを使用する際にチップから直接 AES 命令を呼び出せるようになります。これにより、AES 関数の速度が向上します。AES-NI に関する詳細については、第 4 章 機能を参照してください。

2.7.32 --enable-intelasm

Intel および AMD プロセッサ向けの ASM 高速化を有効にします。

wolfSSL 用に intelasm オプションを有効にすると、プロセッサの拡張機能を活用して AES のパフォーマンスを劇的に向上させます。この設定オプションが有効な場合に活用される命令セットには、AVX1、AVX2、BMI2、RDRAND、RDSEED、AESNI、および ADX が含まれます。これらは Intel プロセッサに最初に導入され、AMD プロセッサも近年採用し始めています。有効にすると、wolfSSL はプロセッサをチェックし、プロセッサがサポートする命令セットを活用します。

2.7.33 --enable-camellia

Camellia サポートを有効にします。

2.7.34 --enable-md2

MD2 サポートを有効にします。

2.7.35 --enable-nullcipher

wolfSSL NULL 暗号サポート（暗号化なし）を有効にします。

2.7.36 --enable-ripemd

wolfSSL ripemd-160 サポートを有効にします。

2.7.37 --enable-blake2

wolfSSL Blake2 サポートを有効にします。

2.7.38 --enable-blake2s

wolfSSL Blake2S サポートを有効にします。

2.7.39 --enable-sha3

x86_64 および Aarch64 では、デフォルトで有効になっています。wolfSSL SHA3 のサポートを有効にします。小型ビルド用に、=small を設定できます。

2.7.40 --enable-sha512

x86_64 ではデフォルトで有効になっています。wolfSSL SHA-512 サポートを有効にします。

2.7.41 --enable-sessioncerts

セッション証明書保存を有効にします。

2.7.42 --enable-keygen

鍵生成機能を有効にします。RSA 鍵生成にのみ適用されます。

2.7.43 --enable-certgen

証明書生成機能を有効にします。

2.7.44 --enable-cert

証明書の拡張機能を有効にします。サポートしている拡張機能については、第 7 章を参照してください。

2.7.45 --enable-certreq

証明書リクエスト生成を有効にします。

2.7.46 --enable-sep

SEP 拡張機能を有効にします。

2.7.47 --enable-hkdf

HKDF (HMAC-KDF) を有効にします。

2.7.48 --enable-x963kdf

X9.63 KDF サポートを有効にします。

2.7.49 --enable-dsa

デジタル署名アルゴリズム (DSA) を有効にします。FIPS 186-4 で定義されている NIST 承認のデジタル署名アルゴリズムと RSA および ECDSA は、Secure Hash Standard (FIPS 180-4) で定義されている承認されたハッシュ関数と組み合わせて使用する場合、デジタル署名の生成と検証に使用されます。

2.7.50 --enable-eccshamir

x86_64 ではデフォルトで有効になっています。ECC Shamir を有効にします。

2.7.51 --enable-ecc

x86_64 ではデフォルトで有効になっています。ECC を有効にします。このオプションを有効にすると、ECC サポートと暗号スイートが wolfSSL に組み込まれます。

2.7.52 --enable-eccustcurves

ECC カスタムカーブを有効にします。すべてのカーブタイプを有効にするには、=all を指定します。

2.7.53 --enable-compkey

圧縮鍵のサポートを有効にします。

2.7.54 --enable-curve25519

Curve25519 を有効にします。Curve25519_SMALL の場合は --enable-curve25519=small と指定します。

楕円曲線は 128 ビットのセキュリティを提供し、ECDH 鍵合意と共に使用されます (後の「クロスコンパイル」節を参照)。Curve25519 を有効にすると、Curve25519 アルゴリズムを使用できます。

デフォルトの Curve25519 は、より多くのメモリを使用し実行時間が速くなるように設計されています。オプション --enable-curve25519=small を使用することで、速度は低下しますがメモリの使用量を抑えることができます。

2.7.55 --enable-ed25519

Ed25519 を有効にします。Ed25519_SMALL の場合は --enable-ed25519=small と指定します。

Ed25519 オプションを有効にすると、Ed25519 アルゴリズムを使用できます。

デフォルトの Ed25519 は、より多くのメモリを使用し実行時間が速くなるように設計されています。オプション --enable-ed25519=small を使用することで、速度は低下しますがメモリの使用量を抑えることができます。

2.7.56 --enable-fpecc

固定小数点キャッシュ ECC を有効にします。

2.7.57 --enable-eccencrypt

ECC 暗号化を有効にします。

2.7.58 --enable-psk

PSK（事前共有鍵）を有効にします。

2.7.59 --disable-errorstrings

エラー文字列テーブルを無効にします。

2.7.60 --disable-oldtls

古い TLS バージョン（1.2 未満）を無効にします。

2.7.61 --enable-ssl3

SSL バージョン 3.0 を有効にします。

2.7.62 --enable-stacksize

サンプルプログラムでのスタックサイズ情報を有効にします。

2.7.63 --disable-memory

メモリコールバックを無効にします。

2.7.64 --disable-rsa

RSA を無効にします。

2.7.65 --enable-rsapss

RSA-PSS を有効にします。

2.7.66 --disable-dh

DH を無効にします。

2.7.67 --enable-anon

匿名認証を有効にします。

2.7.68 --disable-asn

ASN を無効にします。

2.7.69 --disable-aes

AES を無効にします。

2.7.70 --disable-coding

Base16/64 コーディングを無効にします。

2.7.71 --enable-base64encode

x86_64 ではデフォルトで有効になっています。Base64 エンコーディングを有効にします。

2.7.72 --disable-des3

DES3 を無効にします。

2.7.73 --enable-arc4

ARC4 を有効にします。

2.7.74 --disable-md5

MD5 を無効にします。

2.7.75 --disable-sha

Sha を無効にします。

2.7.76 --enable-webserver

Web サーバーを有効にします。

これにより、wolfSSL 組み込み Web サーバーで構築する際に完全な機能を提供するための、標準ビルドに必要な機能がオンになります。

2.7.77 --enable-fips

FIPS 140-2 を有効にします。

注意: 別途、ライセンス契約が必要です。

2.7.78 --enable-sha224

x86_64 ではデフォルトで有効になっています。wolfSSL SHA-224 サポートを有効にします。

2.7.79 --disable-poly1305

wolfSSL Poly1305 サポートを無効にします。

2.7.80 --disable-chacha

chacha を無効にします。

2.7.81 --disable-hashdrbg

ハッシュ DRBG サポートを無効にします。

2.7.82 --disable-filesystem

ファイルシステムのサポートを無効にします。

これにより、ファイルシステムの使用を無効にできます。このオプションは `NO_FILESYSTEM` を定義します。

2.7.83 --disable-inline

インライン関数を無効にします。

このオプションを無効にすると、wolfSSL の関数インライン化が無効になります。関数インライン化が有効な場合、関数ブレースホルダはリンクされず、代わりに関数呼び出し時にコードブロックが挿入されます。

2.7.84 --enable-ocsp

オンライン証明書ステータスプロトコル (OCSP) を有効にします。

このオプションを有効にすると、OCSP(Online Certificate Status Protocol) サポートが wolfSSL に追加されます。[RFC 6960](#)に記載されているように、X.509 証明書の失効状態を取得するために使用されます。

2.7.85 --enable-ocspstapling

OCSP ステーリングを有効にします。TLS1.3 使用時、Certificate メッセージへの複数 OCSP Stapling を無効にするには `=no-multi` を指定します。

2.7.86 --enable-ocspstapling2

OCSP ステープリングバージョン 2 を有効にします。

2.7.87 --enable-crl

CRL(証明書失効リスト) を有効にします。

2.7.88 --enable-crl-monitor

CRL モニターを有効にします。

このオプションを有効にすると、特定の CRL(証明書の失効リスト) ディレクトリを積極的に監視する機能が追加されます。

2.7.89 --enable-sni

サーバー名表示を有効にします (SNI)。

このオプションを有効にすると、TLS サーバー名表示 (SNI) 拡張機能がオンになります。

2.7.90 --enable-maxfragment

最大フラグメント長を有効にします。

このオプションを有効にすると、TLS の最大フラグメント長拡張機能がオンになります。

2.7.91 --enable-alpn

アプリケーションレイヤープロトコルネゴシエーション (ALPN) を有効にします。

2.7.92 --enable-truncatedhmac

切り詰められたキー付きハッシュ MAC (HMAC) を有効にします。

このオプションを有効にすると、TLS 切り詰め HMAC 拡張機能がオンになります。

2.7.93 --enable-renegotiation-indication

再ネゴシエーションの表示を有効にします。

[RFC 5746](#)で説明されているように、この仕様は、再ネゴシエーションを実行する TLS 接続に結び付けることにより、再ネゴシエーションスプライシングを含む SSL/TLS 攻撃を防ぎます。

2.7.94 --enable-secure-renegotiation

安全な再ネゴシエーションを有効にします。

2.7.95 --enable-supportedcurves

サポートされている楕円曲線を有効にします。

このオプションを有効にすると、TLS サポート ECC 曲線拡張機能がオンになります。

2.7.96 --enable-session-ticket

セッションチケットを有効にします。

2.7.97 --enable-extended-master

拡張マスターシークレットを有効にします。

2.7.98 --enable-tlsx

すべての TLS 拡張機能を有効にします。

このオプションを有効にすると、現在 wolfSSL でサポートしているすべての TLS 拡張機能がオンになります。

2.7.99 --enable-pkcs7

PKCS # 7 サポートを有効にします。

2.7.100 --enable-pkcs11

PKCS # 11 アクセスを有効にします。

2.7.101 --enable-ssh

wolfSSH オプションを有効にします。

2.7.102 --enable-scep

wolfSCEP (Simple Certificate Enrollment Protocol) を有効にします。

IETF によって定義されているように、SCEP は HTTP 上で PKCS#7 と PKCS#10 を活用する PKI です。CERT は SCEP が証明書要求を強力的に認証しないことに注意しています。

2.7.103 --enable-srp

安全なリモートパスワードを有効にします。

2.7.104 --enable-smallstack

スタックの使用量を抑えます。

2.7.105 --enable-valgrind

ユニットテストで Valgrind を有効にします。

このオプションを有効にすると、wolfSSL ユニットテストを実行するときに Valgrind がオンになります。これは、開発サイクルの早い段階で問題を発見するのに役立ちます。

2.7.106 --enable-testcert

テスト証明書を有効にします。

このオプションが有効になると、通常は公開されない ASN 証明書 API の一部が公開されます。これは wolfCrypt テストアプリケーション (wolfcrypt/test/test.c) で使用されているように、テストに役立ちます。

2.7.107 --enable-iopool

I/O プールの例を有効にします。

2.7.108 --enable-certservice

証明書サービスを有効にします。(Windows Server)

2.7.109 --enable-jni

wolfSSL JNI を有効にします。

2.7.110 --enable-lighty

Lighttpd/Lighty を有効にします。

2.7.111 --enable-stunnel

stunnel を有効にします。

2.7.112 --enable-md4

MD4 を有効にします。

2.7.113 --enable-pwdbased

pwdbased を有効にします。

2.7.114 --enable-scrypt

scrypt を有効にします。

2.7.115 --enable-cryptonly

wolfCrypt のみを有効にします。

2.7.116 --disable-examples

サンプルプログラムのビルドを無効にします。

サンプルプログラムのビルドを有効にすると、wolfSSL のサンプルアプリケーションがビルドされます。詳細は、第 3 章 入門をご参照ください。

2.7.117 --disable-crypttests

暗号ベンチマーク/テストを無効にします。

2.7.118 --enable-fast-rsa

Intel IPP を使用した RSA を有効にします。

fast-rsa を有効にすると、IPP ライブラリを使用して RSA 操作が高速化されます。通常よりも多くのメモリを使用するようになります。IPP ライブラリが見つからない場合、設定中にエラーメッセージが表示されます。autoconf ははじめに wolfssl_root/IPP ディレクトリを探索し、次は Linux システムの /usr/lib/などのマシン上のライブラリの標準的な場所を確認します。

RSA 操作に使用されるライブラリは、wolfssl-X.X.X/IPP/ディレクトリにあります。(X.X.X は現在の wolfSSL バージョン番号) バンドルされたライブラリからのビルドはディレクトリの場所と IPP の名前に依存しているため、サブディレクトリ IPP のファイル構造は変更しないでください。

メモリを割り当てる際、fast-rsa 操作のメモリタグは DYNAMIC_TYPE_USER_CRYPT0 です。これにより、fast-rsa オプションを有効化した場合の、RSA 操作によるメモリ使用状況を確認できます。

2.7.119 --enable-staticmemory

静的メモリ使用を有効にします。

2.7.120 --enable-mcapi

マイクロチップ API を有効にします。

2.7.121 --enable-asynccrypt

非同期暗号を有効にします。

2.7.122 --enable-sessionexport

セッションのエクスポートとインポートを有効にします。

2.7.123 --enable-aeskeywrap

AES キーラップサポートを有効にします。

2.7.124 --enable-jobserver

値: yes(デフォルト)/ no/#

make を使用する場合、マルチスレッドビルドを使用して wolfSSL をビルドします。yes (デフォルト) は CPU コア の数を検出し、そのカウントに対して推奨されるジョブ数を使用してビルドします。これは make -j オプションと同様に機能します。手動で固定値を指定したい場合には、その値を使用します。

2.7.125 --enable-shared[=PKGS]

wolfSSL ライブラリを共有ライブラリとしてビルドします。デフォルトで有効です。

共有ライブラリビルドを無効にすると、wolfSSL 共有ライブラリがビルドされるのを除外します。デフォルトでは、時間とスペースを節約するために共有ライブラリのみがビルドされています。

2.7.126 --enable-static[=PKGS]

wolfSSL ライブラリをスタティックライブラリとしてビルドします。デフォルトでは無効です。使用する場合はこのオプションを指定してください。

2.7.127 --with-liboqs=PATH

OpenQuantumsafe インストールへのパスです。デフォルトでは /usr/local を参照します。

これにより、wolfSSL が実験的な TLS 1.3 Quantum-Safe KEM グループ、ハイブリッド量子セーフ KEM グループ、Liboqs との wolfSSL 統合を介して Falcon Signature Scheme を使用する機能がオンになります。詳細については、このドキュメントの「Quantum-Safe Cryptography の実験」を参照してください。

2.7.128 --with-libz=PATH

オプションで圧縮用に libz を含めます。

libz を有効にすると、libz ライブラリからの圧縮サポートが wolfSSL で可能になります。このオプションを含めて `wolfSSL_set_compression()` を呼び出すことについては、よくご検討ください。送信前にデータを圧縮すると、送受信されるメッセージの実際のサイズが減少します。しかし、圧縮されたデータはほとんどのネットワークにおいて、そのまま送信するよりも分析に時間がかかります。

2.7.129 --with-cavium

cavium/software ディレクトリへのパス。

2.7.130 --with-user-crypto

USER_CRYPTO インストールへのパス。デフォルトでは /usr/local を参照します。

2.7.131 --enable-rsavy

RSA 検証のみのサポートを有効にします。

注意: `--enable-cryptonly` が必要です。

2.7.132 --enable-rsapub

デフォルト値：RSA 公開鍵のみのサポートを有効にします。

注意: `--enable-cryptonly`が必要です。

2.7.133 --enable-armasm

ARMV8 ASM サポートを有効にします。

デフォルトの構成は、64 か 32 ビットシステムに基づいて MCU または MFPU を設定します。CPPFLAGS を使用して渡された MCU または MFPU 設定を上書きしません。一部のコンパイラでは、constraints のために `-mstrict-align` が必要になる場合があり、CPPFLAGS でユーザーが MCU/MFPU フラグを渡さない限り、`-mstrict-align` もデフォルトで設定されます。

2.7.134 --disable-tls12

TLS 1.2 のサポートを無効にします。

2.7.135 --enable-tls13

TLS 1.3 サポートを有効にします。

このビルドオプションを `--disable-tls12` および `--disable-openssl` と組み合わせて、TLS 1.3 のみの wolfSSL ビルドを生成できます。

2.7.136 --enable-all

SSL v3 を除くすべての wolfSSL 機能を有効にします。

2.7.137 --enable-xts

AES-XTS モードを有効にします。

2.7.138 --enable-asio

ASIO を有効にします。

一緒に `--enable-opensslextra` と `--enable-opensslall` を有効にする必要があります。これら 2 つのオプションが有効になっていない場合、autoconf ツールは自動的にこれらのオプションを有効にします。

2.7.139 --enable-qt

Qt 5.12 以降のサポートを有効にします。

wolfSSL QT ポートと互換性のある wolfSSL ビルド設定を有効にします。QT ソースファイルにパッチを当てるには、wolfSSL が提供するパッチファイルが必要です。

2.7.140 --enable-qt-test

Qt テスト互換ビルドを有効にします。

組み込みの QT テストの実行との互換性のために、wolfSSL をビルドするためのサポートを有効にします。

2.7.141 --enable-apache-httpd

Apache httpd サポートを有効にします。

2.7.142 --enable-afalg

ハードウェアアクセラレーション用の Linux モジュール AF_ALG の使用を有効にします。Xilinx での追加使用は `=xilinx`、`=xilinx-rsa`、`=xilinx-aes`、`=xilinx-sha3` で可能です。

`--enable-devcrypto`と同様に、暗号操作のオフロードのために Linux カーネルモジュール (AF_ALG) を活用します。一部のハードウェアでは、Linux の暗号ドライバを通じてパフォーマンス加速が可能です。Petalinux と Xilinx の場合、フラグ `--enable-afalg=xilinx` を使用して、AF_ALG の Xilinx インターフェースを使用するように wolfSSL に指示できます。

2.7.143 --enable-devcrypto

ハードウェアアクセラレーション用の Linux `/dev/crypto` の使用を有効にします。

aes (すべての aes サポート)、hash (すべてのハッシュアルゴリズム)、および cbc (aes-cbc のみ) の任意の組み合わせを引数で指定できます。オプションが指定されていない場合、デフォルトで `all` を使用します。

2.7.144 --enable-mcast

wolfSSL DTLS マルチキャストサポートを有効にします。

2.7.145 --disable-pkcs12

PKCS12 コードを無効にします。

2.7.146 --enable-fallback-scsv

シグナリング暗号スイート値 (SCSV) を有効にします。

2.7.147 --enable-psk-one-id

TLS 1.3 を使用した単一の PSK ID のサポートを有効にします。

2.7.148 --enable-cryptocb

暗号コールバックを有効にします。wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_CTX_SetDevid を使用して関連付けられている devid を設定します。--enable-cryptocb で次の 2 つの定義を使用して、RSA または ECC ソフトウェアフォールバックをコンパイルして、ソフトウェア RSA/ECC が不要な場合のフットプリント削減のために最適化できます。

- WOLF_CRYPTOCB_ONLY_RSA - RSA ソフトウェア暗号フォールバックをコンパイルアウトします。
- WOLF_CRYPTOCB_ONLY_ECC - ECC ソフトウェア暗号フォールバックをコンパイルアウトします。

WOLF_CRYPTOCB_ONLY_* オプションを使用するには、サンプルプログラムのビルドを無効にする必要があります。--disable-examplesを参照してください

2.7.149 --enable-reproducible-build

バイナリジッター (タイムスタンプやその他の非機能メタデータ) を抑制し、同一のハッシュを持つビット単位で同一のバイナリパッケージを生成できるようにします。

2.7.150 --enable-sys-ca-certs

wolfSSL が `wolfSSL_CTX_load_system_CA_certs()` が呼び出されたときに、検証のために信頼されたシステム CA 証明書を使用できるようにします。これは、wolfSSL 証明書マネージャーにそれらをロードするか、システム認証 API を呼び出すことによって行われます。詳細については、`wolfSSL_CTX_load_system_CA_certs()` を参照してください。

2.8 特別な数学最適化フラグ

2.8.1 --enable-fastmath

FastMath の実装を有効にします。単精度 (SP) 演算が有効な場合、FastMath と Big Integer ライブラリの両方が無効になります。

USE_FAST_MATH および Big Integer Math Library 節を参照してください。

2.8.2 --enable-fasthugemath

Fast Math + 巨大なコードを有効にします。

FastHugeMath の有効化には FastMath ライブラリのサポートが含まれます。公開鍵に関する実装に一般的な鍵サイズのループを展開することで、コードサイズが大幅に増加します。fasthugemath を使用する前後でベンチマークユーティリティを実行して、わずかな速度向上がコードサイズの増加に見合うかどうかをご確認ください。

2.8.3 --enable-sp-math

制限されたアルゴリズムスイートを使用した単精度 (SP) 数学実装を有効にします。サポートされていないアルゴリズムは無効になっています。--enable-sp、--enable-sp-math-all、--enable-fastmath、--enable-fasthugemath よりも優先されます。

- 整数ライブラリーの実装を `sp_int.c` の実装に置き換えます。
- 最小限の実装で、`sp_int.c` の一部を有効にしますが、すべてではありません。
- RSA/ECC/DH 操作を実行できるようにするには、--enable-sp と組み合わせて、`sp_x86_64.c` や `sp_arm.c` などのソリューション（ターゲットシステムに応じた以下のファイルリスト）をオンにする必要があります。
- --enable-sp-math-all (下記) と組み合わせてはいけません

ファイルリスト (プラットフォームに依存します。システムの仕様に基づいて構成によって選択されるか、Makefile/IDE ソリューションを使用する場合は手動で制御できます):

- `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

デフォルトで有効になっています。完全なアルゴリズムスイートを持つ単精度 (SP) 数学実装を有効にします。サポートされていないアルゴリズムが有効になっていますが、最適化されていません。--enable-sp、--enable-fastmath、--enable-fasthugemath よりも優先されます。

- 数学の実装を `sp_int.c` の実装に置き換えます。

- 完全な実装であり、`--enable-sp` の動作に依存しません
- `--enable-sp` と組み合わせて、可能であれば、32 ビットの `sp_c32.c` または 64 ビットの `sp_c64.c` で移植可能な C アセンブリ (ハードウェア固有でないアセンブリ) で記述された実装を使用できるようになります。それ以外 (不可能な場合) には、`sp_int.c` の実装が使用されます。移植可能な C アセンブリは、ハードウェアの最適化が利用できないターゲットでパフォーマンスを大幅に向上させます。
- `--enable-sp-math` (上記) と組み合わせてはいけません

注: 鍵の長さがビット [256, 384, 521, 1024, 2048, 3072, 4096] の非対称暗号を使用している場合は、最大のパフォーマンスを得るために `--enable-sp-math` オプションの使用をご検討ください。ただしフットプリントサイズは大きくなります。

2.8.5 `--enable-sp-asm`

単精度 (SP) アセンブリの実装を有効にします。

Intel x86_64 および ARM アーキテクチャを使用したアセンブリを通して、単精度性能の向上を可能にするために使用できます。

2.8.6 `--enable-sp=OPT`

RSA、DH、および ECC の単一精度 (SP) 数学を有効にして、パフォーマンスを改善します。

`+OPT` には多くの値を設定できます。以下は、`enable-sp` を呼び出す方法と、結果として定義される結果のマクロのリストです。これらはすべて、カンマ区切りのリストで組み合わせることができます。例えば、`--enable-sp=ec256,ec384` のようにします。定義されるマクロの意味は、上記の「wolfSSL 独自の単精度 (SP) 数学サポート」節に記載しています。

注: 1) `--enable-sp=small` `--enable-sp-math` を使用することで、より小さくできる可能性があります。2) `--enable-sp-math-all=small...`

(1) は特定のキーサイズのみ実装しているのに対し、(2) はすべてのキーサイズをサポートする実装を持っているためです。

注: これは x86_64 用で、他の構成フラグはありません。結果は、指定するアーキテクチャやその他の構成フラグによって異なる場合があります。たとえば、`WOLFSSL_SP_384` と `WOLFSSL_SP_4096` は、Intel X86_64 に対してのみ有効になります。

2.8.6.1 `--enable-sp=no` または `--disable-sp` 新しいマクロは定義されません。`--enable-sp` を使用しないのと同等です。

2.8.6.2 `--enable-sp` または `--enable-sp=yes`

- `WOLFSSL_HAVE_SP_RSA`
- `WOLFSSL_HAVE_SP_ECC`
- `WOLFSSL_HAVE_SP_DH`
- `WOLFSSL_SP_384`
- `WOLFSSL_SP_4096`
- `WOLFSSL_SP_LARGE_CODE`

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 または--enable-sp=p256 または--enable-sp=p256

- WOLFSSL_HAVE_SP_ECC

2.8.6.6 --enable-sp=smallec256 または--enable-sp=smallp256 または--enable-sp=small256

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_SMALL

2.8.6.7 --enable-sp=ec384 または--enable-sp=p384 または--enable-sp=384

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_384
- WOLFSSL_SP_NO_256

2.8.6.8 --enable-sp=smallec384 または--enable-sp=smallp384 または--enable-sp=small384

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_384
- WOLFSSL_SP_NO_256
- WOLFSSL_SP_SMALL

2.8.6.9 --enable-sp=ec1024 または--enable-sp=p1024 または--enable-sp=1024

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_1024
- WOLFSSL_SP_NO_256

2.8.6.10 --enable-sp=smallec1024 または--enable-sp=smallp1024 または--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 または --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 他のアルゴリズムオプションと組み合わせて、それらのオプションに対してアセンブリコードがオンになっていることを示します。例えば、`--enable-sp=rsa2048,asm` のように設定します。

2.9 クロスコンパイル

組み込み機器プラットフォーム上の多くのユーザーは、wolfSSL をクロスコンパイルしています。ライブラリをクロスコンパイルさせる最も簡単な方法は、`./configure` システムを使用することです。MakeFile を生成し、wolfSSL をビルドするために使用できます。

クロスコンパイルを行う場合、次のように ./configure にホストを指定する必要があります。

```
./configure --host=arm-linux
```

また、使用するコンパイラ、リンカーなどを指定する必要がある場合があります。

```
./configure --host=arm-linux CC=arm-linux-gcc AR=arm-linux-ar RANLIB=arm-linux  
malloc をオーバーライドしていることを検出し、rpl_malloc および/または rpl_realloc について  
configure システムがエラーを報告することがあります。
```

未定義参照が発生する場合は、./configure に以下を追加してください。

```
ac_cv_func_malloc_0_nonnull=yes ac_cv_func_realloc_0_nonnull=yes
```

クロスコンパイルのために wolfSSL を正しく設定した後は、ライブラリのビルドとインストールのための標準的な autoconf の手順に従うことができます。

```
make  
sudo make install
```

wolfSSL のクロスコンパイルに関する追加のヒントやフィードバックがある場合は、info@wolfssl.jpまでお知らせください。

2.9.1 サンプルツールチェーンを使ったクロスコンパイルのコンフィグオプション例

2.9.1.1 armebv7-eabi-hf-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-eabi-hf-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-eabi-hf-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./"
```

```
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 移植向けビルド

wolfSSL は多くの環境やデバイスに移植（ポート）されています。これらのポートの一部とそれらのマニュアルはディレクトリ `wolfSSL-X.X.X/IDE` にあります。ここで、`x.x.x` は現在の wolfSSL バージョン番号です。このディレクトリには、環境用の wolfSSL をビルドするために使用されている IDE の有用な情報とコードも含まれています。

ポートリスト：

- Arduino
- LPCXPRESSO
- Wiced Studio
- CSBench
- SGX Windows と Linux
 - これらのディレクトリ (`wolfSSL/IDE/WIN-SGX` および `wolfSSL/IDE/LINUX-SGX`) には、Intel SGX プロジェクトで使用されるライブラリとして wolfSSL をビルドするための Makefiles および Visual Studio ソリューションが含まれています。
- Hexagon
 - このディレクトリ (`wolfSSL/IDE/HEXAGON`) には、Hexagon ツールチェーンを備えたビルド用の MakeFile が含まれています。ECC 検証操作を DSP プロセッサにオフロードするための wolfSSL をビルドするために使用できます。ディレクトリには、ビルドに必要な手順を支援する ReadMe ファイルが含まれています。
- Hexiwear
- NetBurner M68K
 - ディレクトリ (`wolfSSL/IDE/M68K`) には、NetBurner RTO を使用して MCF5441X デバイス用の wolfSSL をビルドするための MakeFile があります。
- Renesas
 - このディレクトリ (`wolfSSL/IDE/Renesas`) は、ルネサスデバイスの異なる複数のビルドを含みます。ハードウェアアクセラレーションを使用して実証するビルド例もあります。
- XCode
- Eclipse
- Espressif
- IAR-EWARM
- Kinetis Design Studio(KDS)
- Rowley Crossworks ARM
- OpenSTM32
- RISCV
- Zephyr
- mynewt
- INTIME-RTOS

2.11 NXP CAAM 向けのビルド

2.11.1 i.MX8(Linux)

2.11.1.1 既知の問題

- HSM セッション (wc_SECO_CloseHSM および wolfSSL_Cleanup または wolfCrypt_Cleanup) を閉じる前に、開いている HSM キーストアセッションを終了すると、次回 NVM を起動したときにセグメンテーションフォールトが発生します。これを回避するには、電源を入れ直す必要があります。

2.11.1.2 制限事項

- 多量の入力（1 MByte）を扱う SECO での AES 操作は“共有メモリに十分なスペースがありません”とエラーになります。
- 2 つの鍵ストアを生成した後、3 つ目を生成しようとする失敗します。鍵ストアをリセットするために `rm -rf /etc/seco_hsm` を実行し電源を再投入してください。

2.11.1.3 イントロ i.MX8 デバイスには、高度なセキュリティのための SECO ハードウェアモジュールが用意されています。このモジュールは AES 操作とキーストレージ、限定的な ECC 操作とキーストレージを処理し、RNG を提供します。wolfSSL は可能な限り SECO を利用するように拡張されています。

一部のアルゴリズムは i.MX8 の CAAM でサポートされていますが、SECO モジュールではまだサポートされていません。これらのケースでは、wolfSSL は `/dev/crypto` を通じて Linux CAAM ドライバーを呼び出し、CAAM に直接ジョブを作成します。

NXP Linux CAAM ドライバーではデフォルトでサポートされているアルゴリズムもありますが、CAAM がサポートするすべてのアルゴリズムがサポートされているわけではありません。wolfSSL は Linux CAAM ドライバーを拡張して、追加のアルゴリズムのサポートを追加しています。

同じアプリケーションから両方の経路で CAAM にアクセスするには、異なるコードパスに関連付けられた `devId` を `WOLFSSL_CAAM_DEVID` または `WOLFSSL_SECO_DEVID` のいずれかに設定できます。これらの ID は、構造体を最初に初期化する際に使用され、構造体の存続期間中に使用されるコードパスを設定します。ソフトウェアのみを使用する場合は、デフォルトの `INVALID_DEVID` を設定する必要があります。この例外は、キーストアを使用しない SECO 項目です：TRNG およびハッシュ。

使用されたソフトウェアのバージョンは以下のとおりです。

- `imx-seco-libs` ブランチ `imx_5.4.24_2.1.0`
- NXP 「repo」 ツールと Yocto ビルド
- wolfSSL 5.2.0 以降（5.2.0 リリース後に開発したためです）

2.11.1.4 サポートされるアルゴリズム サポートされるアルゴリズム、モード、オペレーションとして、以下を含みます。

- AES (ECB, CBC, CCM, GCM)
- AES CMAC
- SHA256, SHA384
- ECC 256/384（鍵生成、署名、検証、`ecdh`）
- RSA 3072（鍵生成、署名、検証）
- HMAC
- Curve25519
- TRNG

2.11.1.5 イメージのビルド

2.11.1.5.1 “repo” セットアップ 以下に示す NXP 「repo」 コマンドツールのセットアップは、Ubuntu 18.04 LTS で実施しました。

```
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 libstdl1.2-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"
```

2022 年 1 月 11 日現在、GitHub は認証されていない Git 接続を許可しなくなりました。このドキュメントが作成された時点 (2022 年 3 月) では、NXP のリポジトリツールはこれをまだ考慮していません。この問題を回避するには、次のコマンドを使用して `git://github.com/` を `https://github.com/` にリダイレクトしてください。

```
git config --global url."https://github.com/".insteadOf git://github.com/
```

ビルド用のディレクトリを作成します。

```
mkdir imx-yocto-bsp
cd imx-yocto-bsp/
```

NXP の「repo」コマンドツールを設定した後、目的の Linux バージョンでディレクトリを初期化して同期します。この場合は 5.4.24_2.1.0 です。

```
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=imx8qpc0mek source imx-setup-release.sh -b
↪ build-xwayland
```

2.11.1.5.2 追加の Yocto CAAM レイヤー 次に、`cryptodev-module`、`cryptodev-linux`、および `linux-imx/drivers/crypto/caam/*` のファイルにパッチを適用する CAAM ドライバー拡張レイヤーをダウンロードします。ベース blob と ECDSA (署名/検証/鍵生成) は [こちら](https://source.codeaurora.org/external/imxsupport/imx_sec_apps/) にあります。RSA ブラックキー、ECDH、および Curve25519 サポートを持つ拡張レイヤーは `meta-imx-expand-caam` です。これらのディレクトリを他の既存の `meta-*` ディレクトリの横にある `sources` ディレクトリに配置します。

直前のコマンドの `build-xwayland` ディレクトリに居る想定

クローンするかあるいは zip ファイルを解凍

```
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/
```

あるいは

```
git clone https://source.codeaurora.org/external/imxsupport/imx_sec_apps.git
# "meta-imx-expand-caam" は wolfSSL が提供します
unzip meta-imx-expand-caam.zip
```

```
cp -r imx_sec_apps/meta-imx-ecdsa-sec ../sources/
mv meta-imx-expand-caam ../sources/
```

これらのレイヤーをビルドに追加します。最初に ecdsa を、次に CAAM 拡張を追加します。

```
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 "
```

このビルドではデバッグツールと SSH サーバーを追加しましたが、必ずしも必要ではありません。追加する重要な項目は cryptodev-module と cryptodev-linux です。eckey は、BLOB のカプセル化とカプセル化解除を行うための NXP のデモツールです。

オプション: cryptodev モジュールの自動ロードを追加するには、次の行を conf/local.conf に追加します。

```
KERNEL_MODULE_AUTOLOAD += "cryptodev"
```

それ以外の場合は、電源を入れ直すたびに modprobe cryptodev を使用してモジュールをロードする必要があります。

2.11.1.5.3 ビルドとデプロイ イメージのビルドの開始には次のコマンドを使います。そのあとで、もし SD カードを使っているならばカードに書き込みます。

```
bitbake core-image-base
```

```
cd tmp/deploy/images/imx8qxp0mek/
bzzcat core-image-base-imx8qxp0mek.sdcard.bz2 | sudo dd of=/dev/diskX bs=5m
```

注: 5m は macOS の場合に使用し、Linux の場合は 5M を使用します。diskX は SD カードの場所、つまり Mac であれば disc2、Linux であれば sdbX に置き換える必要があります。実行する前に、SD カードのディスク番号を確認してください。

後ほど wolfSSL/examples のビルドに使用するために、次のコマンドを使用してインストールディレクトリをエクスポートします。

```
export CRYPTODEV_DIR=`pwd`/tmp/sysroots-components/aarch64/cryptodev-
↪ linux/usr/include/
```

クロスコンパイルのためにツールチェーンをインストールするに、次の Yocto コマンドを使用します。

```
bitbake meta-toolchain
sudo ./tmp/deploy/sdk/<version>.sh
```

2.11.1.6 NXP HSM のビルド

2.11.1.6.1 zlib のビルド これを行うには複数の方法があり、その一つは Yocto ビルドに追加する方法です。bitbake を使用してビルドする方法は次のとおりです。

```
cd build-xwayland
bitbake zlib
```

このコマンドは実行結果を tmp/sysroots-components/aarch64/zlib/usr/ディレクトリに配置します。

後ほど wolfSSL/examples のビルドに使用するために、次のコマンドを使用してインストールディレクトリをエクスポートします。

```
export ZLIB_DIR=`pwd`/tmp/sysroots-components/aarch64/zlib/usr/
```

2.11.1.6.2 NXP HSM ライブラリのビルド NXP HSM ライブラリをダウンロードし、必要な zlib が見つかるように Makefile や環境変数を調整します。

```
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
```

さらに

```
make
make install
```

後ほど wolfSSL/examples のビルドに使用するために、次のコマンドを使用してインストールディレクトリをエクスポートします。

```
export HSM_DIR=`pwd`/export/usr/
```

make install を行うとデフォルトで結果を export サブディレクトリに配置します。

2.11.1.7 wolfSSL のビルド

2.11.1.7.1 Autoconf を使用したビルド 開発ツールを使用して Yocto イメージをセットアップすると、wolfSSL をシステム上に直接構築できます。より最小限のアプローチとして、クロスコンパイルを使用できます。

デバッグメッセージは --enable-debug で有効にできます。SECO 作業に固有の追加のデバッグメッセージは、マクロ DEBUG_SECO を定義し、/dev/crypto 呼び出しの DEBUG_DEVCRYPTO を定義することで有効にできます。どちらの追加のデバッグメッセージも printf を使用し、stdout パイプに出力します。

SECO で使用するための重要な有効化オプションがいくつかあります。--enable-caam=seco、--enable-devcrypto=seco、--with-seco=/hsm-lib/export などです。

HSM SECO のみを使用したビルド例 (追加アルゴリズムの devcrypto サポートなし)

```
source /opt/fsl-imx-wayland/5.4-zeus/environment-setup-aarch64-poky-linux
```

wolfSSL のビルドで依存しているコンポーネントをインストール

```
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
```

HSM SECO と追加の devcrypto サポートを使つてのビルド例。インクルードパスとして crypto/crypt-pdev.h を指定する必要があります。

```
./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
```

wolfCrypt_Init / wolfSSL_Init 関数呼び出しで早期にエラーを出すフェイルセーフがあります。一つのケースは、cryptodev モジュールがロードされていないか、目的の操作に利用可能なサポートがない場合です。初期化操作が失敗する可能性がある別のケースは、NXP HSM をセットアップできなかった場合です。

アプリケーションが初期化に失敗している場合、wolfSSL ビルドに --enable-debug を追加し、wolfSSL の初期化前に wolfSSL_Debugging_ON() 関数呼び出しを行うと、失敗の理由に関する有用なデバッグメッセージが表示されます。

デバッグオプションを有効にしてビルドする例

```
./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 user_settings.h を使用したビルド autotools なしでビルドするために有効にできるマクロは次のとおりです。

CAAM

- WOLFSSL_CAAM - CAAM サポートを有効にするメインマクロスイッチ。
- WOLF_CRYPT_CB - CAAM サポートは暗号コールバックを利用します。
- WOLFSSL_SECO_CAAM - CAAM での SECO HSM 使用を有効にします (AES-GCM が必要で、平文鍵を HSM にインポートするためのアルゴリズムとして使用されます)。
- WOLFSSL_HASH_KEEP - SHA256 などのアルゴリズムでハッシュする場合、メッセージを構築し、Final 呼び出し時にのみハッシュするために送信します。
- WOLFSSL_CAAM_ECC - CAAM ECC サポートを有効にします。
- WOLFSSL_CAAM_CMIC - CAAM CMAC サポートを有効にします。
- WOLFSSL_CAAM_CIPHER - CAAM AES サポートを有効にします。
- WOLFSSL_CAAM_HMAC - CAAM HMAC サポートを有効にします。
- WOLFSSL_CAAM_HASH - SHA256 などの CAAM ハッシングサポートを有効にします。
- WOLFSSL_CAAM_CURVE25519 - CAAM Curve25519 サポートを有効にします。

cryptodev-linux

- WOLFSSL_DEVCRYPTO - cryptodev-linux 使用を有効にするメインマクロスイッチ。
- WOLFSSL_DEVCRYPTO_HMAC - cryptodev-linux での HMAC サポートを有効にします。
- WOLFSSL_DEVCRYPTO_RSA - cryptodev-linux での RSA サポートを有効にします。
- WOLFSSL_DEVCRYPTO_CURVE25519 - cryptodev-linux での Curve25519 サポートを有効にします。
- WOLFSSL_DEVCRYPTO_ECDSA - cryptodev-linux での ECDSA サポートを有効にします。
- WOLFSSL_DEVCRYPTO_HASH_KEEP - cryptodev-linux でのハッシュ蓄積サポートを有効にします。

CAAM サポートのためにコンパイルする必要がある追加ファイル

- 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 サンプルプログラム

2.11.1.8.1 Testwolfcrypt の実行 wolfSSL にバンドルされている単体テストは、wolfcrypt/test/test.c にあります。デバイス上でテストを構築して実行する例は次のとおりです。これは WOLFSSL_CAAM_DEVID を使用するため、NXP HSM ライブラリではなく cryptodev モジュールを使用しているにご注意ください。

```
./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@imx8qpc0mek:~# /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@imx8qxp0mek:~#

```

その他のサンプルプログラムを、wolfSSL-examples リポジトリの caam/seco ディレクトリ配下に用意しています。

```

git clone https://github.com/wolfSSL/wolfSSL-examples
cd wolfSSL-examples/caam/seco
make

```

2.11.1.8.2 ソースコードをコンパイル [user_settings.h を使用] wolfSSL にリンクする単一のソースファイルを構築するには、次のコマンドを使用します。前のステップの環境変数がまだ設定されていると仮定します。

```
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 追加された API

- void wc_SECO_AesSetKeyID(Aes* aes, int keyId);
 - この関数は SECO キー ID を Aes 構造体にセットする際に使用します。
 - Aes 構造体が初期化された後で、暗号化/復号操作に使用される前に呼び出される必要があります。
- int wc_SECO_AesGetKeyID(Aes* aes);

- Aes 構造体にセットされている SECO キー ID のゲッター関数です。
- `void wc_SECO_CMACESetKeyID(Cmac* cmac, int keyId);`
 - `wc_SECO_AesSetKeyID` 関数と似ていますが、対象が `Cmac` 構造体になっています。
- `int wc_SECO_CMACEGetKeyID(Cmac* cmac);`
 - `Cmac` 構造体にセットされている SECO キー ID のゲッター関数です。
- `int wc_SECO_OpenHSM(word32 keyStoreId, word32 nonce, word16 maxUpdates, byte flag);`
 - この関数はキーストアが必要となるどの操作 (ECC や AES) の前に呼び出される必要があります。
 - 最初の引数はキーストア ID、nonce は特定の 32 ビットの連番で、キーストアを新規に作成する場合や、既存のキーストアをアンロックするのに使用されます。
 - maxUpdates はキーストアが更新できる最大回数を指定します。
 - flag はオプションフラグで、キーストアが生成されているのかを示すために使用されます。
 - HSM キーストアを生成するためには flag は `CAAM_KEYSTORE_CREATE` でなければなりません。
 - `CAAM_KEYSTORE_UPDATE` は既存のキーストアをオープンし更新します。
- `int wc_SECO_CloseHSM(void);`
 - この関数は、キーストアの使用が完了したとき、`wolfCrypt_Cleanup/wolfSSL_Cleanup` を呼び出す前に呼び出す必要があります。
 - 現在オープンしているキーストアを閉じます。
- `int wc_SECO_GenerateKey(int flags, int group, byte* out, int outSz, int keyType, int keyInfo, unsigned int* keyIdOut);`
 - この関数は、SECO で新しいキーを生成するために使用できます。
 - キー生成の場合、flags は `CAAM_GENERATE_KEY` である必要があります。
 - キーを更新する場合、flags は `CAAM_UPDATE_KEY` である必要があります。
 - keyIdOut 引数は、キーの作成時に設定される入出力引数であり、キーの更新時に入力として設定する必要があります。
 - キーを更新する場合は、一時的なタイプである必要があります、更新のために group を 0 に設定する必要があります。
 - 一時的なタイプは keyInfo 引数として設定されます。
 - keyInfo と keyType の可能なオプションは以下のとおりです。
 - * `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);`
 - キーストアから鍵を削除するのに使用されます。

2.11.1.9.2 CAAM をサポートするネイティブ wolfSSL API これは、このドキュメントで概説されている SECO ビルドで CAAM サポートを備えたネイティブ wolfSSL API のリストです。

AES 暗号化と復号化操作の生成には、次のプロセスで鍵を生成できます。

`wc_SECO_GenerateKey(CAAM_GENERATE_KEY, groupID, pubOut, 0, CAAM_KEYTPE_AES128, CAAM_KEY_PERSISTENT, &keyIdOut)` を使用します。

ここで `groupID` は指定されたグループ番号、`pubOut` は 32 バイトのバッファ、変数 `keyIdOut` は生成された新しい鍵 ID に設定されます。

この生成された新しい鍵 ID は、`wc_SECO_AesSetKeyID(Aes, keyIdOut)` を使用して `Aes` 構造体に設定できます。

鍵 ID が構造体に設定され、`Aes` 構造体が `WOLFSSL_SECO_DEVID` タイプとして初期化されると、すべての暗号化および復号化操作にその鍵 ID が使用されます。

AES (ECB/CBC)

AES ECB/CBC 鍵を生成する代わりに、`Aes` 構造体が `WOLFSSL_SECO_DEVID` で初期化されている場合は、プレーンテキストキーを渡して関数 `wc_AesSetKey` を呼び出すことができます。

API `wc_AesSetKey` は、一意の KEK を使用して鍵を暗号化し、SECO HSM にインポートしようとします。インポートが成功すると、値 0 が返され、鍵 ID が `Aes` 構造体に設定されます。

- CBC 暗号化は `wc_AesCbcEncrypt` で行われ、復号は `wc_AesCbcDecrypt` で行われます。
- ECB 暗号化は `wc_AesEcbEncrypt` で行われ、復号は `wc_AesEcbDecrypt` で行われます。

`Aes` 構造体の使用が完了したら、`wc_AesFree(Aes)` を使用して解放する必要があります。

AES-GCM

- GCM 暗号化は `wc_AesGcmEncrypt` で行われ、復号は `wc_AesGcmDecrypt` で行われます。

AES-GCM 暗号化関数は、`Aes` 構造体、出力バッファ、入力バッファ、入力バッファサイズ、ノンス、ノンスサイズ (12 バイト必要)、MAC またはタグとして知られる、タグサイズ (16 バイトが必要)、追加データ、追加データサイズ (4 バイト) を引数として取ります。

暗号化の場合、入力バッファは暗号化され、タグバッファは作成された MAC で埋められます。

AES-GCM 復号の場合、関数は `Aes` 構造体、プレーンテキスト出力バッファ、暗号テキスト入力バッファ、入力バッファサイズ、ノンス、ノンスサイズ (12 バイト)、暗号化呼び出しで以前に作成されたタグ、タグバッファサイズ、追加データ、追加のデータサイズを引数として取ります。

復号化時にタグバッファがチェックされ、メッセージの整合性が検証されます。

`Aes` 構造体の使用が完了したら、`wc_AesFree(Aes)` を使用して解放する必要があります。

AES-CCM

- CCM 暗号化は `wc_AesCcmEncrypt`、復号は `wc_AesCcmDecrypt` で行われます。

AES-CCM 暗号化関数は、`Aes` 構造体、出力バッファ、入力バッファ、入力バッファサイズ、ノンス、ノンスサイズ (12 バイトが必要)、MAC またはタグとして知られる、タグサイズ (16 バイト必要)、追加データ、追加データサイズ (0 バイト) を引数として取ります。追加のデータバッファは NULL にする必要があり、NXP HSM ライブラリではサイズ 0 が必要です。暗号化の場合、入力バッファは暗号化され、タグバッファは作成された MAC で埋められます。

AES-CCM 復号の場合、関数は `Aes` 構造体、プレーンテキスト出力バッファ、暗号テキスト入力バッファ、入力バッファサイズ、ノンス、ノンスサイズ (12 バイト)、暗号化呼び出しで以前に作成されたタグ、タグバッファサイズ、追加データ、追加のデータサイズを引数に取ります。暗号化関数と同様に、追加のデータバッファは NULL である必要があります。復号時にタグバッファがチェックされ、メッセージの整合性が検証されます。

`Aes` 構造体の使用が完了したら、`wc_AesFree(Aes)` を使用して解放する必要があります。

AES CMAC

AES CMAC 操作の場合、`wc_SECO_GenerateKey(CAAM_GENERATE_KEY, groupID, pubOut, 0, CAAM_KEYTYPE_AES128, CAAM_KEY_PERSISTENT, &keyIdOut);` を使用して AES 鍵を生成できます。ここで、`groupID` は指定されたグループ番号、`pubOut` は 32 バイトのバッファであり、変数 `keyIdOut` は生成された新しいキー ID に設定されます。生成されたこの新しいキー ID は、`wc_SECO_CMACESetKeyID(Cmac, keyIdOut)` を使用して Aes 構造体に設定できます。キー ID が構造内に設定され、Aes 構造体が `WOLFSSL_SECO_DEVID` タイプとして初期化されると、すべての暗号化および復号化操作にそのキー ID が使用されます。

SM ライブラリはシングルショットタイプであるため、`wc_CmacUpdate` を呼び出すたびに入力が内部バッファに保存されます。その後、`wc_CmacFinal` が呼び出されると、MAC を作成するためにバッファ全体がハードウェアに渡されます。

RSA

RSA 操作では、`cryptodev-linux` モジュールが使用されます。これには、`WOLFSSL_CAAM_DEVID` で初期化されたときのデフォルトである AES-ECB 暗号化されたブラックプライベートキーのサポートが含まれます。

`cryptodev-linux` モジュールで使用されるネイティブ wolfSSL API の例は次のとおりです。

```
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 署名および検証操作では、`cryptodev-linux` モジュールまたは NXP HSM ライブラリのいずれかを使用できます。共有シークレットを作成するための ECDH 操作は、`cryptodev-linux` モジュールを使用してのみ実行できます。

SECO で使用する場合 (NXP HSM ライブラリを使用)、`ecc_key` 構造体を初期化するときに `WOLFSSL_SECO_DEVID` の dev ID フラグを使用する必要があります。`cryptodev-linux` モジュールを使用するには、dev ID フラグ `WOLFSSL_CAAM_DEVID` を使用する必要があります。関数 `wc_ecc_init_ec(key, heap-hint (NULL 可), dev ID)` による初期化後、どちらのユースケースも、署名と検証を行うためのネイティブの wolfSSL 関数呼び出しの同じ関数に従います。

`ecc_key` 構造体の初期化後の関数呼び出しの例は次のようになります。

```
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);
```

また、`cryptodev-linux` モジュール (`WOLFSSL_CAAM_DEVID`) を使用すると、ECDH 関数を使用できます。

```
wc_ecc_shared_secret(keyA, keyB, sharedSecret, sharedSecretSz);
```

ハッシュ (Sha256, Sha384, HMAC)

SHA256 および SHA384 の操作では、NXP HSM ライブラリを使用します。HMAC 操作では、`cryptodev-linux` モジュールが使用されます。

デフォルトでは、SHA 操作は NXP HSM ライブラリを利用しようとしませんが、明示的に dev ID に設定すると、`WOLFSSL_SECO_DEVID` を使用できます。

```
wc_InitSha256_ex(sha256, heap-hint, WOLFSSL_SECO_DEVID);
wc_InitSha384_ex(sha384, heap-hint, WOLFSSL_SECO_DEVID);
```

NXP HSM ライブラリはハッシュのシングルショット操作をサポートしているため、“update” を呼び出すたびに “final” 関数が呼び出されるまでバッファが保存され、その後ハッシュダイジェストを作成するためにバッファ全体がハードウェアに渡されます。HMAC が `cryptodev-linux` を使用する場合、Hmac 構造体は dev ID `WOLFSSL_CAAM_DEVID` を使用して初期化する必要があります。

```
wc_HmacInit(hmac, heap-hint, WOLFSSL_CAAM_DEVID);
```

その後、通常のネイティブ wolfSSL API と同じように使用できます。

```
wc_HmacSetKey(hmac, hash-type, key, keySz);
wc_HmacUpdate(hmac, input, inputSz);
wc_HmacFinal(hmac, digestOut);
```

Curve25519

Curve25519 ポイント乗算は cryptodev-linux モジュールを使用して行われ、ハードウェアで使用するために dev ID WOLFSSL_CAAM_DEVID で初期化する必要があります。

API 呼び出しの例は次のようになります。

```
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

wolfSSL HASH-DRBG に乱数シードを与えるために使われる TRNG では、NXP HSM ライブラリを利用します。これは、wolfSSL が `--enable-caam=seco` でビルドされるときに、`wolfcrypt/src/random.c` ファイルにコンパイルされます。wolfSSL のすべての RNG 初期化では、シードに TRNG が使用されます。標準の RNG API 呼び出しは次のようになります。

```
wc_InitRng(rng);
wc_RNG_GenerateBlock(rng, output, outputSz);
wc_FreeRng(rng);
```

2.11.2 i.MX8 (QNX)

(ドキュメントは現在準備中です。ご入用でしたら、info@wolfssl.jp までお問い合わせください。)

2.11.3 i.MX6 (QNX)

(ドキュメントは現在準備中です。ご入用でしたら、info@wolfssl.jp までお問い合わせください。)

2.11.4 IMXRT1170 (FreeRTOS)

IMXRT1170 で使用する IDE セットアップの例は、IDE/MCUEXPRESSO/RT1170 ディレクトリにあります。

2.11.4.1 ビルド手順

1. MCUEXPRESSO を開き、ワークスペースを `wolfSSL/IDE/MCUEXPRESSO/RT1170` に設定します。
2. ファイル -> ファイル システムからプロジェクトを開く... -> ディレクトリ: 参照先を `wolfSSL/IDE/MCUEXPRESSO/RT1170` ディレクトリに設定し、「ディレクトリの選択」をクリックします。
3. `wolfSSL_cm7`、`wolfcrypt_test_cm7`、`CSR_example`、`PKCS7_example` を選択します。
4. プロジェクトを右クリック -> SDK 管理 -> SDK コンポーネントを更新し、「はい」をクリックします。
5. `FreeRTOSConfig.h` の `configTOTAL_HEAP_SIZE` のサイズを、CSR および PKCS7 の例では 60240、`wolfcrypt_test_cm7` では約 100000 に増やします。
6. (ノートボードファイルを再作成する必要があります。これは、同じ設定を持つ新しいプロジェクトを作成し、生成された `board/*` ファイルをコピーすることで実行できます)
7. プロジェクトをビルドします。

2.11.4.2 RT1170 CAAM ドライバーの拡張 ファイル RT1170/fs1_caam_h.patch および RT1170/fs1_caam_c.patch には、Blob の作成/オープン、および ECC ブラックキーの生成と使用のための既存の NXP CAAM ドライバーへの変更が含まれています。

パッチを適用するには、まず caam ドライバーを含むプロジェクトを作成します。これにより、ドライバーディレクトリにベースとなる fs1_caam.c と fs1_caam.h が生成されます (つまり、PKCS7_example_cm7/drivers/fls_caam.{c,h})。基本ファイルが生成されたら、ドライバーディレクトリに "cd" してパッチを適用します。

```
cd PKCS7_example_cm7/drivers/  
patch -p1 < ../../fs1_caam_c.patch  
patch -p1 < ../../fs1_caam_h.patch
```

fs1_caam.h のパッチには、ECC と Blob の拡張 (CAAM_ECC_EXPANSION および CAAM_BLOB_EXPANSION) の両方に対するマクロが定義されています。wolfSSL コードはこれらのマクロが定義されている (パッチが適用されている) ことを検出すると、拡張ドライバーの使用をコンパイルしようとします。

3 入門

3.1 概要

wolfSSL (旧称 CyaSSL) は、第 2 章に示したコンパイルオプションを使用した場合、yaSSL の約 10 倍小さく、OpenSSL の 20 倍小さくなります。ユーザーによるベンチマークとフィードバックからも、標準的な SSL 操作のほとんどにおいて、wolfSSL は OpenSSL と比較して劇的に優れたパフォーマンスを示していると報告されています。

ビルドプロセスの手順については、第 2 章をご覧ください。

3.2 テストスイート

テストスイートプログラムは、wolfSSL とその暗号化ライブラリである wolfCrypt が、システム上で正常に実行できるかをテストします。

サンプルプログラムとテストや実行する際は、wolfSSL のホームディレクトリからコマンドを投入する必要があります。これは `./certs` から証明書と鍵を見つけるためです。テストスイートを実行するには、次のコマンドを実行してください。

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

autoconf を使用して、次のように実行することもできます。

```
make test
```

*nix または Windows 環境においては、サンプルプログラムやテストスイートは現在のディレクトリがソースディレクトリかどうかを確認し、そうであれば、wolfSSL ホームディレクトリに変更しようとします。これはほとんどのセットアップケースで動作するはずです。もしうまく実行できない場合には、上記の最初の方法を使用してフルパスを指定してください。

正常に実行されると、次のような出力が表示されるはずです。ユニットテストと暗号スイートテストのための追加の出力も含んでいます。

```
-----
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!

これは、すべてが正しく構成およびビルドされていることを示しています。いずれかのテストに失敗した場合は、ビルドシステムが正しく設定されていることをご確認ください。エンディアンが間違っていたり、64ビットタイプが適切に設定されていなかったりする可能性があります。デフォルトの構成からオプションを変更している場合は、それらを削除して wolfSSL を再ビルドし、改めてテストを実行してください。

3.3 クライアントサンプルプログラム

examples/client にあるクライアントのサンプルプログラムを使用して、任意の SSL/TLS サーバーに対して wolfSSL をテストできます。利用可能なコマンドラインオプションのリストを表示するには、--help を付けてクライアントを実行します。

```
./examples/client/client --help
```

出力例

```

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)

```

example.com:443 に対してテストするには、次のようにします。ここでは、ビルドオプション--enable-opensslextra と--enable-supportedcurves を使用してコンパイルした wolfSSL を使用しています。

```
./examples/client/client -h example.com -p 443 -d -g
```

出力例

```

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

```

これにより、クライアントは example.com(-h) の HTTPS ポート 443(-p) に接続し、一般的な GET リクエスト (-g) を送信するように指示します。(-d) オプションにより、クライアントにサーバー証明書の検証をスキップするよう指示しています。このオプションを付加している理由については、以下の -188 エラーに関する説明をご覧ください。残りは読み取りバッファに収められたサーバーからの初期出力です。

コマンドライン引数が指定されていない場合、クライアントは localhost 上の wolfSSL デフォルトポート 11111 に接続を試みます。また、サーバーがクライアント認証を実行したい場合を想定しクライアント証明書も読み込みます。

クライアントは引数 -b < 整数 > を使用して接続のベンチマークを実行できます。このオプションを使用すると、クライアントは引数として渡された回数の接続を試行し、SSL_connect() の実行にかかった平均時間をミリ秒単位で表示します。

例えば、以下のように実行します。

```
/examples/client/client -b 100 -h example.com -p 443 -d
```

出力例

```
wolfSSL_connect avg took: 296.417 milliseconds
```

/wolfssl/test.h を編集することで、デフォルトのホストやポートを localhost・11111 から任意の値に変更できます。変数 wolfSSLIP / wolfSSLPort を編集してください。なお、これらの設定を変更した場合は、testsuite を含むすべてのサンプルプログラムを再ビルドする必要があります。そうしないと、テストプログラムは互いに接続できません。

デフォルトでは、クライアントサンプルプログラムは TLS 1.2 を使用して指定されたサーバーに接続しようとします。ユーザーは -v コマンドラインオプションを使用して、クライアントが使用する SSL/TLS バージョンを変更できます。このオプションでは、以下の値を指定できます。

- -v 0 - SSL 3.0 (デフォルトでは無効)
- -v 1 - TLS 1.0
- -v 2 - TLS 1.1
- -v 3 - TLS 1.2 (デフォルトで選択)
- -v 4 - TLS 1.3

クライアントサンプルプログラムを実行する際、次のように error -188 が出力される場合があります。

```
wolfSSL_connect error -188, ASN no signer error to confirm failure
wolfSSL error: wolfSSL_connect failed
```

これは通常、wolfSSL クライアントが接続先のサーバー証明書を検証できないことが原因です。デフォルトでは、wolfSSL クライアントは「yaSSL テスト CA 証明書」を信頼されたルート証明書として使用します。これはテスト用の CA 証明書であり、外部の CA によって署名されたサーバー証明書を検証することはできません。そのため、この問題を解決するには、ユーザーは -d オプションを使用してサーバーの検証をオフにするか、

```
./examples/client/client -h myhost.com -p 443 -d
```

-A コマンドラインオプションを使用して、正しい CA 証明書を wolfSSL クライアントに読み込ませる必要があります。

```
./examples/client/client -h myhost.com -p 443 -A serverCA.pem
```

3.4 サーバーサンプルプログラム

サーバーサンプルプログラムは、クライアント認証も実行可能なシンプルな SSL/TLS サーバーです。1 つのクライアント接続のみが受け入れられ、その後サーバーは終了します。

コマンドライン引数なしのクライアントサンプルプログラムは、サーバーサンプルプログラムに対して問題なく接続できます。しかし、クライアントサンプルプログラムにコマンドライン引数を指定すると、ク

クライアント証明書が読み込まれず、wolfSSL_connect() が失敗します。このとき、サーバーは error -245, peer didn't send cert を出力します。これを回避するためには、-d オプションを使用してクライアント証明書チェックを無効にしてください。

サンプルプログラムのクライアントと同様に、サーバーもいくつかのコマンドライン引数で使えます。

```
./examples/server/server --help
```

出力例

```
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 1111
-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 エコーサーバーのサンプルプログラム

エコーサーバーのサンプルプログラムは、無限ループで無制限のクライアント接続を待ち続けます。そしてクライアントが送信したものをすべてエコーバックします。

クライアント認証は実行されません。したがって、クライアントサンプルプログラムはすべてのモードでエコーサーバーに接続できます。

なお、以下に示す 4 つの特別なコマンドはエコーバックされず、エコーサーバーに異なる処理を実行するよう指示します。

1. quit - エコーサーバーが文字列「quit」を受信すると、シャットダウンします。
2. break - エコーサーバーが文字列「break」を受信すると、現在のセッションを停止しますが、リクエストの処理は継続します。これは DTLS テストに特に役立ちます。
3. printstats - エコーサーバーが文字列「printstats」を受信すると、セッションキャッシュの統計情報を出力します。
4. GET - エコーサーバーが文字列「GET」を受信すると、HTTP の GET として処理し、「greeting from wolfSSL」というメッセージを含む簡単なページを送り返します。これにより、Safari、IE、Firefox、GnuTLS といったさまざまな TLS/SSL クライアントのテストが可能になります。

NO_MAIN_DRIVER が定義されていない限り、エコーサーバーの出力は標準出力にエコーされます。シェルを通して、または 1 番目のコマンドライン引数を使用して出力をリダイレクトすることもできます。エコーサーバーからの出力を output.txt という名前のファイルに保存したい場合には、次のように実行します。

```
./examples/echoserver/echoserver output.txt
```

3.6 エコークライアントのサンプルプログラム

エコークライアントのサンプルプログラムは、インタラクティブモードまたはファイルによるバッチモードで実行できます。

インタラクティブモードで実行し、3 つの文字列「hello」、「wolfssl」、および「quit」を書き込むと、結果は以下のようになります。

```
./examples/echoclient/echoclient
hello
hello
wolfssl
wolfssl
quit
sending server shutdown command: quit!
```

ファイル入力を使用するには、コマンドライン上で最初の引数としてファイル名を指定します。ファイル input.txt の内容をエコーするには、以下のように実行します。

```
./examples/echoclient/echoclient input.txt
```

結果をファイルに書き出したい場合は、追加のコマンドライン引数として出力ファイル名を指定できます。ファイル input.txt の内容をエコーし、サーバーからの結果を output.txt に書き込むには、次のように実行します。

```
./examples/echoclient/echoclient input.txt output.txt
```

テストスイートプログラムではこの動作を行いますが、クライアントとサーバーが期待される結果を正しく送信/受信できていることを確認するため、入力・出力ファイルをハッシュ化しています。

3.7 ベンチマーク

組み込み向け SSL/TLS ライブラリ wolfSSL が、使用を検討しているハードウェアデバイスや特定の環境で、果たしてどれほどのパフォーマンスを発揮できるのかは非常に気になるものです。しかし、今日の組み込み / エンタープライズ / クラウド環境では、非常に多数のプラットフォーム・コンパイラが使用されています。これらの全てに対応できる、一般的なパフォーマンス指標の算出式を提供することは非常に困難です。

そこで、ユーザーが wolfSSL / wolfCrypt のパフォーマンスを容易に判断できるように、ベンチマークアプリケーションをバンドルしています。wolfSSL は、SSL/TLS の基盤であるすべての暗号操作に wolfCrypt 暗号ライブラリを使用します。したがって、ベンチマークアプリケーションは wolfCrypt のパフォーマンステストを実行します。

wolfcrypt/benchmark (./wolfcrypt/benchmark/benchmark) にあるベンチマークユーティリティを使用して、wolfCrypt の暗号処理のベンチマークを実行できます。

```
./wolfcrypt/benchmark/benchmark
```

以下のような出力がなされます。なお、ここでは、ECC、SHA-256、SHA-512、AES-GCM、AES-CCM、および Camellia を含むいくつかのオプションのアルゴリズム/暗号が有効になっています。

```
-----
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

```

これは、数学ライブラリを変更する前後で公開鍵演算の速度を比較するのに特に役立ちます。通常の数学ライブラリ (`./configure`)、`fastmath` ライブラリ (`./configure --enable-fastmath`)、`fasthugemath` ライブラリ (`./configure --enable-fasthugemath`) に対して、それぞれテストを実行できます。

詳細とベンチマーク結果については、wolfSSL ベンチマークページを参照してください。

<https://www.wolfssl.com/docs/benchmarks>

3.7.1 相対パフォーマンス

個々の暗号やアルゴリズムのパフォーマンスはホストプラットフォームによって異なりますが、wolfCrypt の暗号間の相対的なパフォーマンスを以下のグラフに示します。このテストは、2.2 GHz Intel Core i7 を搭載した Macbook Pro (OS X 10.6.8) で実施されました。

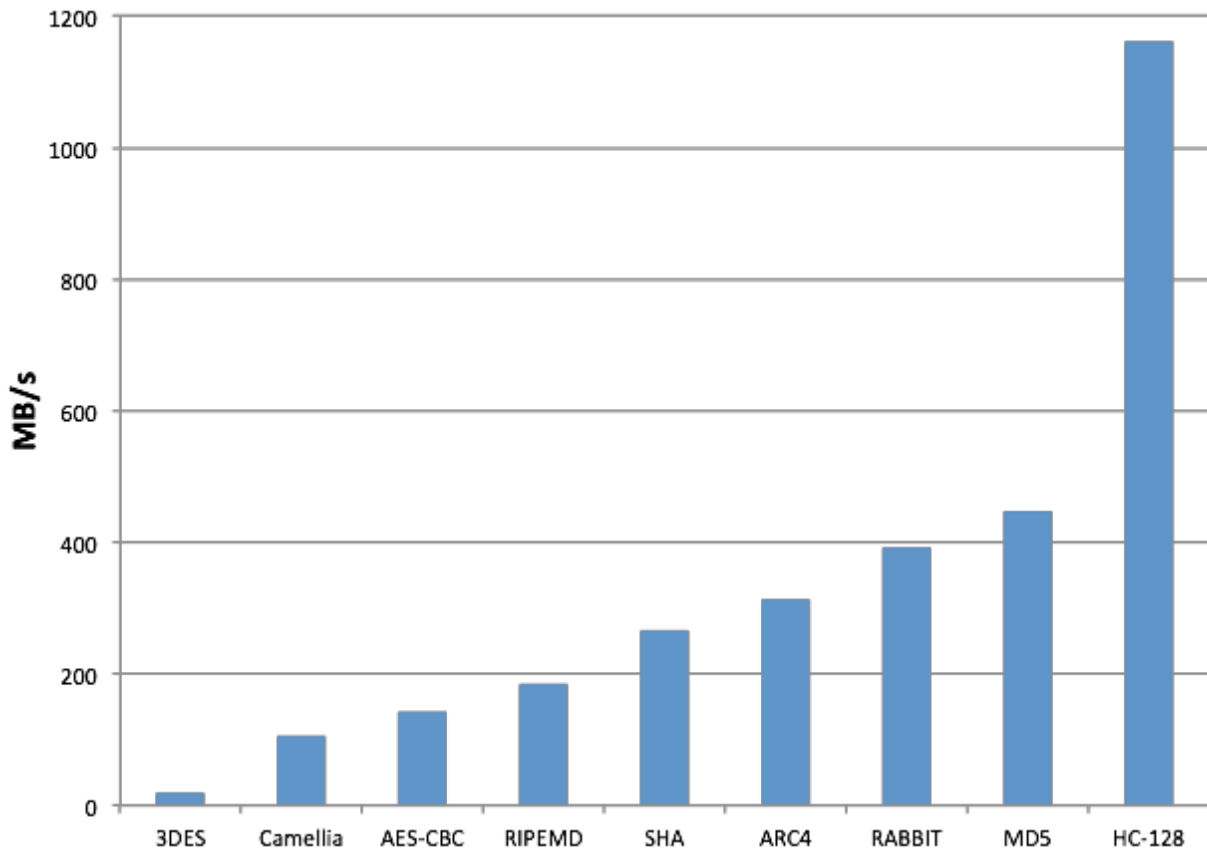


Figure 1: ベンチマーク

暗号のサブセットのみを使用したい場合は、SSL/TLS 接続を行う際に wolfSSL が使用する特定の暗号スイートや Cipher をカスタマイズできます。例えば 128 ビット AES を強制するには、`wolfSSL_CTX_new(SSL_CTX_new)` の呼び出しの後に、次の行を追加します。

```
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

3.7.2 ベンチマークに関する補足

1. プロセッサのネイティブレジスタサイズ (32 ビット対 64 ビット) は、1000 ビット以上の公開鍵操作を行う際、大きな違いをもたらす可能性があります。

2. **keygen** (`--enable-keygen`) により、ベンチマークユーティリティを実行する際、鍵生成速度もベンチマークすることができます。
3. **fastmath** (`--enable-fastmath`) は動的メモリ使用量を削減し、公開鍵操作を高速化します。32 ビットプラットフォームで **fastmath** を使用してビルドする際に問題がある場合は、PIC がレジスタを占有しないように共有ライブラリを無効にしてください。このとき、README の注意事項も併せてご参照ください。

```
./configure --enable-fastmath --disable-shared  
make clean  
make
```

注意： wolfSSL で構成オプションを切り替える際には、`make clean` を実行することを推奨しています。

4. デフォルトでは、**fastmath** は可能であればアセンブリ最適化を使用しようとします。アセンブリ最適化が機能しない場合でも、wolfSSL をビルドする際に `CFLAGS` に `TFM_NO_ASM` を追加することで、アセンブリ最適化なしで **fastmath** を使用できます。

```
./configure --enable-fastmath C_EXTRA_FLAGS="-DTFM_NO_ASM"
```

5. **fasthugemath** を使用すると、組み込みプラットフォームで実行していないユーザー向けに **fastmath** をさらに強化できます。

```
./configure --enable-fasthugemath
```

6. デフォルトの wolfSSL ビルドでは、メモリ使用量とパフォーマンスのバランスを取るよう努めています。どちらかをより重視する場合は、追加の wolfSSL 構成オプションについて第 2 章「ビルドオプション」節を参照してください。
7. **バルク転送：** wolfSSL はデフォルトで 128 バイトの I/O バッファを使用します。SSL トラフィックの約 80% がこのサイズに収まり、動的メモリの使用を制限するためです。バルク転送が必要な場合は、16K バッファ (SSL の最大サイズ) を使用するように構成できます。

3.7.3 組み込みシステムでのベンチマーク

組み込みシステム上で容易にベンチマークアプリケーションをビルドできるように、複数のビルドオプションがあります。

3.7.3.1 BENCH_EMBEDDED この定義を有効にすると、ベンチマークアプリケーションはメモリ使用量をメガバイトからキロバイトに切り替えます。具体的には、暗号処理と各アルゴリズムは 25kB でベンチマークを実行します。また、一部の組み込みプロセッサでは公開鍵操作に非常に長い時間を要するため、公開鍵アルゴリズムは 1 回のみベンチマークを実行します。メモリ使用量や繰り返し回数の値は、`benchmark.c` 内の `BENCH_EMBEDDED` 定義内にある変数 `numBlocks` と `times` を変更することで調整できます。

3.7.3.2 USE_CERT_BUFFERS_1024 この定義を有効にすると、ベンチマークアプリケーションはファイルシステムからテスト鍵と証明書を読み込む代わりに、`<wolfssl_root>/wolfssl/certs_test.h` にある 1024 ビットの鍵と証明書バッファを使用します。これは組み込みプラットフォームにファイルシステムがなく (`NO_FILESYSTEM` と併用)、2048 ビット公開鍵操作が現実的でないほど遅いプロセッサの場合に役立ちます。

3.7.3.3 USE_CERT_BUFFERS_2048 `USE_CERT_BUFFERS_1024` と似ていますが、1024 ビットの代わりに 2048 ビットの鍵と証明書バッファを使用します。プロセッサが 2048 ビット公開鍵操作を行うのに十分な速度を持っているものの、ファイルシステムがなくファイルから鍵と証明書を読み込むことが難しいケースに役立ちます。

3.8 クライアントアプリケーションで wolfSSL を使用する

この節では、wolfSSL ネイティブ API を使用して、クライアントアプリケーションに wolfSSL を追加するための基本的な手順について説明します。サーバーの説明については、本章の「サーバーアプリケーションで wolfSSL を使用する」節をご参照ください。より詳しい手順とサンプルコードは第 11 章 SSL チュートリアルに掲載しています。OpenSSL 互換性レイヤーの詳細については、第 13 章を参照してください。

1. wolfSSL ヘッダーをインクルードします。

```
#include <wolfssl/ssl.h>
```

2. wolfSSL と WOLFSSL_CTX を初期化します。最終的に作成する WOLFSSL オブジェクトの数に関係なく、1 つの WOLFSSL_CTX を使用します。基本的には、接続先のサーバーを検証するために CA 証明書を読み込む必要があります。基本的な初期化手順は次のようになります。

```
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. TCP 接続ごとに WOLFSSL オブジェクトを作成し、ファイルディスクリプタをセッションに関連付けます。

```
/*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. すべての read() (または recv()) の呼び出しを wolfSSL_read() に変更します。例えば以下の場合、

```
result = read(fd, buffer, bytes);
```

次のように変更します。

```
result = wolfSSL_read(ssl, buffer, bytes);
```

5. すべての write() (または send()) の呼び出しを wolfSSL_write() に変更します。例えば以下の場合、

```
result = write(fd, buffer, bytes);
```

次のように変更します。

```
result = wolfSSL_write(ssl, buffer, bytes);
```

6. wolfSSL_connect() を手動で呼び出すこともできますが、必ずしも必要ではありません。最初の wolfSSL_read() または wolfSSL_write() の呼び出し時に、必要に応じて wolfSSL_connect() を実行します。

- エラーチェックは以下のように実装します。それぞれの `wolfSSL_read()` と `wolfSSL_write()` 呼び出しは、成功時にはバイト数を返し、接続終了時には 0 を、エラー時には -1 を返します。これは `read()` と `write()` と同様です。エラーが発生した場合、次の 2 つの関数を使用して詳しい情報を入手できます。

```
char errorString[80];
int err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, errorString);
```

ノンブロッキングソケットを使用している場合は、`errno EAGAIN/EWOULDBLOCK` をテストするか、より正確には `wolfSSL_get_error()` が返す特定のエラーコードが `SSL_ERROR_WANT_READ` または `SSL_ERROR_WANT_WRITE` であるかをテストできます。

- クリーンアップは次のように実装します。各 WOLFSSL オブジェクトの使用が終了したら、以下を呼び出して解放してください。

```
wolfSSL_free(ssl);
```

SSL/TLS の使用が完全に終了したら、以下を呼び出して WOLFSSL_CTX オブジェクトを解放してください。

```
wolfSSL_CTX_free(ctx);
wolfSSL_Cleanup();
```

wolfSSL を使用したクライアントアプリケーションのサンプルプログラムについては、`<wolfssl_root>/examples/client` をご参照ください。

3.9 サーバーアプリケーションで wolfSSL を使用する

この節では、wolfSSL ネイティブ API を使用してサーバーアプリケーションに wolfSSL を追加するための基本的な手順について説明します。クライアントの説明については、本章の「クライアントアプリケーションで wolfSSL を使用する」節を参照してください。より詳しい手順とサンプルコードは第 11 章 **SSL チュートリアル** に掲載しています。

- 概ね上記のクライアントの手順と同様です。ただし、ステップ 2 のクライアントメソッド呼び出しをサーバーのものに変更します。例えば以下の行を、

```
wolfSSL_CTX_new(wolfTLSv1_client_method());
```

次のように変更します。

```
wolfSSL_CTX_new(wolfTLSv1_server_method());
```

SSLv3 や TLSv1 を使用するクライアントの接続も受け付ける場合には、以下のようにします。

```
wolfSSL_CTX_new(wolfSSLv23_server_method());
```

- 上記のステップ 5 の初期化に、サーバーの証明書と鍵ファイルを追加します。

```
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);
}
```

ファイルシステムが利用できない場合は、バッファから証明書と鍵を読み込むこともできます。詳しい情報は `wolfSSL_CTX_use_certificate_buffer()` と `wolfSSL_CTX_use_PrivateKey_buffer()` の API ドキュメントをご参照ください。

wolfSSL を使用したサーバーアプリケーションのサンプルプログラムは、`<wolfssl_root>/examples/server.c` ファイルを参照してください。

4 機能

wolfSSL は主要なインターフェースとして C 言語をサポートしていますが、[SWIG](#) インターフェースを通じて Java、PHP、Perl、Python などサポートしています。現在サポートされていない他のプログラミング言語で wolfSSL をホストすることに興味がある場合は、お問い合わせください。

本章では、ストリーム暗号、AES-NI、IPv6 サポート、SSL 検査（スニファー）サポートなど、wolfSSL の機能についてより詳しく説明します。

4.1 機能概要

wolfSSL の機能概要については、製品紹介ページをご覧ください。

<https://wolfssl.jp/products/wolfssl>

4.2 プロトコルサポート

wolfSSL は **SSL 3.0**、**TLS (1.0、1.1、1.2、1.3)**、および **DTLS (1.0 と 1.2)** をサポートしています。以下の関数のいずれかを使用することで、使用するプロトコルを簡単に選択できます。wolfSSL は SSL 2.0 をサポートしていません。OpenSSL 互換性レイヤーを使用する場合、以下の関数はわずかに異なります。OpenSSL 互換関数については、第 13 章 OpenSSL 互換性をご参照ください。

4.2.1 サーバー関数

- `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()` - SSLv3 から TLS 1.2 までの可能な限り新しいバージョンを使用

wolfSSL は `wolfSSLv23_server_method()` 関数による堅牢なサーバーダウングレードをサポートしています。詳しくは、本章の「堅牢なクライアントとサーバーのダウングレード」節をご参照ください。

4.2.2 クライアント関数

- `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()` - SSLv3 から TLS 1.2 までの可能な限り新しいバージョンを使用

wolfSSL は `wolfSSLv23_client_method()` 関数による堅牢なクライアントダウングレードをサポートしています。詳しくは、本章の「堅牢なクライアントとサーバーのダウングレード」節をご参照ください。

これらの関数の使用方法の詳細については、[第 3 章 入門](#)をご参照ください。SSL 3.0、TLS 1.0、1.1、1.2、および DTLS の比較については、付録 D SSL/TLS の概要をご参照ください。

4.2.3 堅牢なクライアントとサーバーのダウングレード

wolfSSL のクライアントとサーバーはどちらも堅牢なバージョンダウングレード機能を持っています。どちらかの側で特定のプロトコルバージョンメソッドが使用されている場合、そのバージョンによりネゴシエーションされるか、エラーが返されます。例えば、TLS 1.0 を使用するクライアントが SSL 3.0 のみのサーバーに接続しようとする、接続は失敗します。同様に、TLS 1.1 への接続も失敗します。

この問題を解決するには、クライアント側で `wolfSSLv23_client_method()` 関数を使用し、必要に応じてダウングレードすることでサーバーがサポートする最高のプロトコルバージョンをサポートします。この場合、クライアントは TLS 1.0~TLS 1.3（または SSL 3.0 を含むサブセットまたはスーパーセット、wolfSSL で設定されているプロトコルバージョンによる）を実行しているサーバーに接続できます。接続できないバージョンは、長年安全でないとされている SSL 2.0 と、デフォルトで無効化されている SSL 3.0 のみです。

同様に、サーバー側で `wolfSSLv23_server_method()` 関数を使用することで、TLS 1.0~TLS 1.2 のプロトコルバージョンをサポートするクライアントを処理できます。wolfSSL サーバーはセキュリティ上の問題から、SSLv2 からの接続を受け入れません。

4.2.4 IPv6 サポート

wolfSSL は IPv6 もサポートしています。

wolfSSL は IP ニュートラルとして設計しており、IPv4 と IPv6 の両方で動作します。ただしより広範なシステムに適用するために、現在のテストアプリケーションはデフォルトで IPv4 を使用します。テストアプリケーションで IPv6 を使用するには、wolfSSL のビルド時に `--enable-ipv6` オプションをお使いください。

IPv6 に関する一般的な情報はこちらで確認できます。

<https://en.wikipedia.org/wiki/IPv6>

4.2.5 DTLS

wolfSSL はクライアントとサーバーの両方で DTLS（「データグラム」TLS）をサポートしています。現在サポートしている最新バージョンは DTLS 1.3 です。

TLS プロトコルは**信頼性のある**媒体（TCP など）を介して安全なトランスポートチャネルを提供するように設計されました。UDP トランスポートを使用するアプリケーション層プロトコル（SIP やゲームプロトコルなど）が開発され始めるにつれて、遅延に敏感なアプリケーションの通信セキュリティを提供する必要性が生じました。この必要性が DTLS プロトコルの作成につながりました。

多くの人が TLS と DTLS の違いは TCP と UDP の違いと同じだと考えていますが、これは正しくありません。UDP はハンドシェイクがなく、切断がなく、何かが失われた場合に TCP と比較して遅延が少ないという利点があります。一方、DTLS は拡張された SSL ハンドシェイクと切断を持ち、ハンドシェイクに TCP のような動作を実装する必要があります。本質的に、DTLS は安全な接続と引き換えに UDP が提供する利点を逆転させます。

DTLS は wolfSSL をビルドする際に `--enable-dtls` ビルドオプションを使用することで有効にできます。

4.2.6 LwIP（軽量インターネットプロトコル）

wolfSSL は、軽量インターネットプロトコル実装をすぐ使えるようサポートしています。このプロトコルを使用するには、`WOLFSSL_LWIP` を定義するか、`settings.h` ファイルを編集して次の行のコメントアウトを解除してください。

```
/*#define WOLFSSL_LWIP*/
```

LwIP の焦点は、できるだけ小さな RAM 使用量で完全な TCP スタックを提供することです。これはまさに、wolfSSL の主なユースケースである組み込みシステムでの SSL/TLS ニーズとマッチしています。

4.2.7 TLS 拡張機能

wolfSSL がサポートする TLS 拡張機能のリストと、参照できる対応する RFC についての注記です。

RFC	拡張機能	wolfSSL タイプ
6066	サーバー名表示	TLSX_SERVER_NAME
6066	最大フラグメント長ネゴシエーション	TLSX_MAX_FRAGMENT_LENGTH
6066	短縮 HMAC	TLSX_TRUNCATED_HMAC
6066	ステータスリクエスト	TLSX_STATUS_REQUEST
7919	サポートされているグループ	TLSX_SUPPORTED_GROUPS
5246	署名アルゴリズム	TLSX_SIGNATURE_ALGORITHMS
7301	アプリケーション層プロトコルネゴシエーション	TLSX_APPLICATION_LAYER_PROTOCOL
6961	複数証明書ステータスリクエスト	TLSX_STATUS_REQUEST_V2
5077	セッションチケット	TLSX_SESSION_TICKET
5746	再ネゴシエーションの提示	TLSX_RENEGOTIATION_INFO
8446	鍵共有	TLSX_KEY_SHARE
8446	事前共有鍵	TLSX_PRE_SHARED_KEY
8446	PSK 鍵交換モード	TLSX_PSK_KEY_EXCHANGE_MODES
8446	早期データ	TLSX_EARLY_DATA
8446	クッキー	TLSX_COOKIE
8446	サポートされているバージョン	TLSX_SUPPORTED_VERSIONS
8446	ポストハンドシェイク認証	TLSX_POST_HANDSHAKE_AUTH

4.3 暗号サポート

4.3.1 暗号スイートの強度と適切な鍵サイズの選択

メソッド `wolfSSL_get_ciphers()` を呼び出すことで、現在有効になっている暗号スイートを確認できます。

暗号スイートはさまざまな強度で提供されています。それらはいくつかの異なるタイプのアルゴリズム（認証、暗号化、およびメッセージ認証コード（MAC））で構成されているため、各々の強度は選択された鍵サイズによって異なります。

暗号スイートの強度を評価する方法は多数あり、実際に使用される手法はプロジェクトや企業によって異なるようです。これには、共通鍵および公開鍵アルゴリズムの鍵サイズ、アルゴリズムの種類、性能、既知の弱点などが含まれる場合があります。

NIST（米国国立標準技術研究所）は、各鍵サイズに対して比較可能なアルゴリズムの強度を提供することにより、受け入れ可能な暗号スイートを選択するための推奨事項を提供しています。暗号アルゴリズムの強度は、アルゴリズムと使用される鍵サイズに依存します。NIST Special Publication [SP800-57](#)では、どのようにして2つのアルゴリズムが同等の強度であると見なすのか述べられています。

2つのアルゴリズムは、与えられた鍵サイズ（XとY）に対して、「アルゴリズムを破る」または鍵を決定するために必要な作業量が、与えられたリソースを使用して概ね同じである場合、同等の強度を持つと見なされます。特定の鍵サイズに対するアルゴリズムのセキュリティ強度は、伝統的に、ショートカット攻撃（つまり、最も効率的な攻撃はすべての可能な鍵を試すこと）がない「X」の鍵サイズを持つ対称アルゴリズムに対してすべての鍵を試すのにかかる作業量の観点から説明されています。

以下の2つの表は、[NIST SP800-57](#)の表2（56ページ）と表4（59ページ）の両方に基づいており、アルゴリズム間の比較可能なセキュリティ強度と強度測定（NISTが提案するアルゴリズムセキュリティの寿命をセキュリティビット数に基づいて）を示しています。

注意：次の表では、「L」は有限体暗号（FFC）の公開鍵のサイズ、「N」はFFCの秘密鍵のサイズ、「k」は整数因数分解暗号（IFC）の鍵サイズ、「f」は楕円曲線暗号（ECC）の鍵サイズを示しています。

セキュリティ ティビット 数	対称鍵アルゴ リズム	FFC 鍵サイズ (DSA, DH, 等)	IFC 鍵サイズ (RSA, 等)	ECC 鍵サイズ (ECDSA, 等)	説明
80	2TDEA, 等	L = 1024, N = 160	k = 1024	f = 160-223	セキ ュリ ティ は 2010 年ま で有効
128	AES-128, 等	L = 3072, N = 256	k = 3072	f = 256-383	セキ ュリ ティ は 2030 年ま で有効
192	AES-192, 等	L = 7680, N = 384	k = 7680	f = 384-511	長期 的に 有効
256	AES-256, 等	L = 15360, N = 512	k = 15360	f = 512+	予見 可能な 将来に わた って有効

この表をガイドとして使用すると、暗号スイートは対称暗号化アルゴリズムの強度に基づいて分類できます。これにより、セキュリティビット数に基づいて各暗号スイートを大まかに分類するための等級分類を考案できます（対称鍵サイズのみを考慮しています）。

- **低** - 128 ビット未満のセキュリティビット
- **中** - 128 ビットのセキュリティビット
- **高** - 128 ビットを超えるセキュリティビット

暗号スイートの強度は、対称暗号化アルゴリズムの強度以外にも、鍵交換および認証アルゴリズムの鍵サイズに大きく依存します。暗号スイートを構成する要素のうち、最も弱い要素の強度により評価されます。

上記の評価方法（および対称暗号化アルゴリズムの強度のみに基づく）に従うと、wolfSSL 2.0.0 は現在、以下にリストされているように、合計 0 の低強度暗号スイート、12 の中程度強度暗号スイート、および 8 の高強度暗号スイートをサポートしています。この強度分類は、他のアルゴリズムの選択された鍵サイズによって変わる可能性があります。ハッシュ関数のセキュリティ強度に関する参考として、[NIST SP800-57](#)の表 3 (56 ページ) をご参照ください。

媒体によっては、「**EXPORT**」暗号として示されていることがあります。これは 1992 年まで、米国では強力な暗号処理を含むソフトウェアを輸出することが違法とされていたことに由来しています。強力な暗号化は米国政府によって「弾薬」（核兵器、戦車、弾道ミサイルと同じカテゴリ）として分類されていました。この制限のため、輸出されるソフトウェアには「弱められた」暗号（主に小さい鍵サイズ）が含まれていました。現在ではこの制限は解除されており、EXPORT 暗号はもはや必要とされなくなりました。

4.3.2 サポートしている暗号スイート

以下の暗号スイートは wolfSSL でサポートしています。暗号スイートとは、SSL/TLS ハンドシェイク中にコネクションのセキュリティ設定をネゴシエーションするために使用される、認証、暗号化、メッセージ認証コード (MAC) アルゴリズムの組み合わせです。

各暗号スイートは、鍵交換アルゴリズム、バルク暗号化アルゴリズム、メッセージ認証コードアルゴリズム (MAC) を定義します。

鍵交換アルゴリズム (RSA、DSS、DH、EDH) は、ハンドシェイクプロセス中にクライアントとサーバーがどのように認証するかを決定します。

バルク暗号化アルゴリズム (DES、3DES、AES、ARC4) (ブロック暗号とストリーム暗号を含む) は、メッセージストリームを暗号化するために使用されます。

メッセージ認証コード (MAC) アルゴリズム (MD2、MD5、SHA-1、SHA-256、SHA-512、RIPEMD) は、メッセージダイジェストを作成するために使用されるハッシュ関数です。

以下の表は、<wolfssl_root>/wolfssl/internal.h (約 1097 行目以降に記載) にある暗号スイート (およびカテゴリ) に対応しています。以下のリストにない暗号スイートをお探しの場合は、お問い合わせください。

4.3.2.1 ECC 暗号スイート

- 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

4.3.2.2 静的 ECC 暗号スイート

- 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

4.3.2.3 Blake2b 暗号スイート

- TLS_RSA_WITH_AES_128_CBC_B2B256
- TLS_RSA_WITH_AES_256_CBC_B2B256

4.3.2.4 SHA-256 暗号スイート

- 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

4.3.2.5 SHA-384 暗号スイート

- TLS_DHE_PSK_WITH_AES_256_CBC_SHA384
- TLS_DHE_PSK_WITH_NULL_SHA384

4.3.2.6 AES-GCM 暗号スイート

- 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

4.3.2.7 ECC AES-GCM 暗号スイート

- 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

4.3.2.8 AES-CCM 暗号スイート

- 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

4.3.2.9 Camellia 暗号スイート

- 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

4.3.2.10 ChaCha 暗号スイート

- 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

4.3.2.11 再ネゴシエーション表示拡張特別スイート

- TLS_EMPTY_RENEGOTIATION_INFO_SCSV

4.3.3 AEAD スイート

wolfSSL は、AES-GCM、AES-CCM、CHACHA-POLY1305 などの AEAD スイートをサポートしています。これらの AEAD スイートと他のスイートの大きな違いは、追加のクリアテキストデータを使用して暗号化されたデータを認証することです。これは、データ改ざんを伴う中間者攻撃を軽減するのに役立ちます。AEAD スイートはブロック暗号（または最近ではストリーム暗号）アルゴリズムと、鍵付きハッシュアルゴリズムによって生成されるタグの組み合わせを使用します。これら 2 つのアルゴリズムの組み合わせは、wolfSSL の暗号化・復号プロセスによって処理されます。特定の AEAD スイートを使用するために必要なのは、サポートされているスイートで使用するアルゴリズムを有効にするだけです。

4.3.4 ブロック暗号とストリーム暗号

wolfSSL は **AES**、**DES**、**3DES**、**Camellia** ブロック暗号と、**RC4** および **CHACHA20** ストリーム暗号をサポートしています。AES、DES、3DES、RC4 はデフォルトで有効です。

Camellia と ChaCha20 は、wolfSSL のビルド時に（それぞれ `--enable-camellia` および `--disable-chacha` ビルドオプションで）有効にできます。

AES のデフォルトモードは CBC モードです。AES で GCM や CCM モードを有効にするには、ビルドオプション `--enable-aesgcm` または `--enable-aesccm` をお使いください。サンプルプログラムや第 10 章 wolfCrypt の使用法に詳しい情報を掲載しています。

SSL ではデフォルトのストリーム暗号として RC4 を使用していましたが、セキュリティ上の問題により廃止されました。最近、wolfSSL は ChaCha20 を追加しました。RC4 は ChaCha よりも約 11% 高い性能を示しますが、一般的に ChaCha よりも安全性が低いと考えられています。ChaCha はセキュリティ性能とのトレードオフとして、処理時間の短さを提供しています。

暗号性能の比較については、wolfSSL のベンチマークウェブページをご覧ください。

<https://www.wolfssl.com/docs/benchmarks>

4.3.4.1 違いは何でしょうか ブロック暗号は、暗号のブロックサイズのチャンク単位で暗号化する必要があります。例えば、AES のブロックサイズは 16 バイトです。したがって、2~3 バイトの小さなチャンクを暗号化する場合、データの 80% 以上が無駄なパディングとなります。これにより暗号化/復号処理速度は低下し、ネットワーク帯域幅を無駄に消費します。基本的に、ブロック暗号は大きなデータチャンク用に設計されており、パディングを必要とするブロックサイズを有し、固定の不変な変換を使用します。

ストリーム暗号は、大小さまざまなデータチャンクに適しています。ブロックサイズが不要なため、小さいデータサイズにも適しています。速度が重要な場合、ストリーム暗号が最適です。これは、一般的に XOR 処理された鍵ストリームを含むより単純な変換を使用するためです。メディアをストリーミングしたり、小さなサイズを含むさまざまなデータサイズを暗号化したり、高速な暗号が必要な場合は、ストリーム暗号が最適な選択です。

4.3.5 ハッシュ関数

wolfSSL はいくつかの異なるハッシュ関数をサポートしています。これには、**MD2**、**MD4**、**MD5**、**SHA-1**、**SHA-2** (SHA-224、SHA-256、SHA-384、SHA-512)、**SHA-3** (BLAKE2)、**RIPEMD-160** が含まれます。これらの関数の詳細な使用法は、第 10 章 wolfCrypt の使用法の「ハッシュ関数」節で確認できます。

4.3.6 公開鍵オプション

wolfSSL は **RSA**、**ECC**、**DSA/DSS**、**DH** の公開鍵オプションに加え、wolfSSL サーバー上での **EDH** (Ephemeral Diffie-Hellman) もサポートしています。これらの関数の詳細な使用法は、第 10 章 wolfCrypt の使用法の「公開鍵暗号」節で確認できます。

4.3.6.1 ML-KEM、ML-DSA ML-KEM (Module Lattice Key Encapsulation Mechanism) は、NIST が標準化した格子ベースの耐量子アルゴリズムです。Kyber をベースとして標準化されました。2 者間で鍵カプセル化メカニズムを使用して、安全でないチャネル上で共有鍵を確立します。これにより、従来型および量子コンピュータを用いた攻撃者の両方から保護できます。

ML-DSA (Module Lattice Digital Signature Algorithm) は、同じく NIST が標準化した格子ベースの耐量子デジタル署名方式です。Kyber をベースとして標準化されました。これにより、送信者はメッセージの送信元と完全性を証明する検証可能な署名を生成できます。

ML-KEM と ML-DSA はどちらも、暗号の復号に特化した量子コンピュータに耐性を持つように設計された公開鍵アルゴリズムです。これらは NIST の耐量子暗号標準 (FIPS 203 および FIPS 204) の一部です。量子コンピューティング時代に備えるために、ほとんどのケースでハイブリッド形式にてすぐに導入できます。

4.3.7 ECC サポート

wolfSSL は楕円曲線暗号(ECC)をサポートしています。これには、ECDH-ECDSA、ECDHE-ECDSA、ECDH-RSA、ECDHE-PSK、ECDHE-RSA などが含まれます。

wolfSSL の ECC 実装は、<wolfssl_root>/wolfssl/wolfcrypt/ecc.h ヘッダーファイルと <wolfssl_root>/wolfcrypt/src/ecc.c ソースファイルにあります。

サポートされている暗号スイートは上の表に示されています。ECC は非 x86_64 ビルドではデフォルトで無効になっていますが、HAVE_ECC 定義を使用するか、autoconf システムを使用して次のように wolfSSL をビルドすることで有効にできます。

```
./configure --enable-ecc
make
make check
```

make check を実行すると、wolfSSL がチェックする多数の暗号スイートを出力します。もし make check が暗号スイートのリストを生成しない場合は、./testsuite/testsuite.test を単独で実行してください。これらの暗号スイートは、それぞれ個別にテストできます。例えば、ECDH-ECDSA と AES256-SHA を試すには、wolfSSL サーバーを次のように起動します。

```
./examples/server/server -d -l ECDHE-ECDSA-AES256-SHA -c
↪ ./certs/server-ecc.pem -k ./certs/ecc-key.pem
```

(-d) はクライアント証明書チェックを無効にし、(-l) は暗号スイートリストを指定します。(-c) は使用する証明書、(-k) は対応する秘密鍵です。クライアントを接続するには次のようにします。

```
./examples/client/client -A ./certs/server-ecc.pem
```

ここで (-A) はサーバーを検証するための CA 証明書です。

4.3.8 PKCS サポート

PKCS (Public Key Cryptography Standards) は、RSA Security, Inc. によって作成され公開された標準規格のグループを指します。wolfSSL は **PKCS #1**、**PKCS #3**、**PKCS #5**、**PKCS #7**、**PKCS #8**、**PKCS #9**、**PKCS #10**、**PKCS #11**、**PKCS #12** をサポートしています。

さらに、wolfSSL は PKCS #1 の一部として標準化された RSA-Probabilistic Signature Scheme(PSS) もサポートしています。

4.3.8.1 PKCS #5、PBKDF1、PBKDF2、PKCS #12 PKCS #5 はパスワードベースの鍵導出方法です。具体的には、パスワード、ソルト、反復回数を組み合わせてパスワードベースの鍵を生成します。wolfSSL は鍵導出関数として、PBKDF1 と PBKDF2 の両方をサポートしています。鍵導出関数は、基本鍵と他のパラメータ（上記で説明したソルトや反復回数など）から派生鍵を生成します。PBKDF1 はハッシュ関数（MD5、SHA1 など）を適用して鍵を導出し、派生鍵の長さはハッシュ関数の出力長によって制限されます。PBKDF2 では、疑似乱数関数（HMAC-SHA-1 など）を適用して鍵を導出します。PBKDF2 の場合、派生鍵の長さに制限はありません。

wolfSSL は PBKDF1 と PBKDF2 に加えて、PKCS #12 からの PBKDF 関数もサポートしています。関数プロトタイプは次の通りです。

```
int PBKDF2(byte* output, const byte* passwd, int pLen,
           const byte* salt, int sLen, int iterations,
           int kLen, int hashType);

int PKCS12_PBKDF(byte* output, const byte* passwd, int pLen,
                 const byte* salt, int sLen, int iterations,
                 int kLen, int hashType, int purpose);
```

output には派生鍵が含まれ、passwd は長さ pLen のユーザーパスワードを保持します。salt は長さ sLen のソルト入力を保持します。iterations は実行する反復回数、kLen は希望する派生鍵の長さ、hashType は使用するハッシュ (MD5、SHA1、または SHA2) です。

./configure を使用して wolfSSL をビルドする場合、--enable-pwdbased オプションを使用することでこの機能を有効にできます。

完全な例は <wolfSSL Root>/wolfcrypt/test.c にあります。PKCS #5、PBKDF1、PBKDF2 に関する詳細情報は、以下の仕様書から入手できます。

PKCS#5、PBKDF1、PBKDF2 : [RFC 2898](#)

4.3.8.2 PKCS #8 PKCS #8 は、秘密鍵情報構文の標準として設計されました。公開鍵アルゴリズムの秘密鍵と属性セットを含む秘密鍵情報を保存するために使用されます。

PKCS #8 標準には 2 つのバージョンがあり、暗号化された秘密鍵と非暗号化鍵の両方を保存する構文について説明しています。wolfSSL は暗号化と非暗号化の両方の PKCS #8 をサポートしています。サポートしているフォーマットには、PKCS #5 バージョン 1 - バージョン 2、および PKCS#12 が含まれます。暗号化の種類として、DES、3DES、RC4、AES が利用可能です。

PKCS#8 : [RFC 5208](#)

4.3.8.3 PKCS #7 PKCS #7 は、暗号化された証明書や暗号化されていないが署名されたデータ文字列などのデータバンドルを転送するように設計されています。この機能は、ビルドオプション--enable-pkcs7 を使用するか、マクロ HAVE_PKCS7 を定義することで有効にできます。RFC に従って、デフォルトでは署名者の空セットを持つ縮退ケースが許可されています。縮退ケースの許可/不許可を切り替えるには、関数 wc_PKCS7-AllowDegenerate() を使用します。

サポートされている機能は以下の通りです。

- 縮退バンドル
- KARI、KEKRI、PWRI、ORI、KTRI バンドル
- 分離署名
- 圧縮およびファームウェアパッケージバンドル
- カスタムコールバックサポート
- 限定的なストリーミング機能

4.3.8.3.1 PKCS #7 コールバック PKCS7 バンドルが解析された後、ユーザーが鍵を選択できるようにするための追加のコールバックとサポート関数が追加されました。CEK をアンラップするには、wc_PKCS7_SetWrapCEKCb() を使用します。このコールバックは KARI および KEKRI バンドルの場合に呼び出されます。keyID または SKID は、KARI の場合、オリジネータ鍵と共に wolfSSL からユーザーに渡されます。ユーザーが KEK で CEK をアンラップした後、使用する復号された鍵を wolfSSL に戻す必要があります。この実装例は、[wolfssl-examples](#) リポジトリ内の signedData-EncryptionFirmwareCB.c で公開しています。

PKCS7 バンドルの復号のための追加のコールバックが追加されました。復号コールバック関数を設定するには、wc_PKCS7_SetDecodeEncryptedCb() を使用します。ユーザー定義のコンテキストを設定するには、wc_PKCS7_SetDecodeEncryptedCtx() を使用する必要があります。このコールバックは、wc_PKCS7_DecodeEncryptedData() の呼び出し時に実行されます。

4.3.8.3.2 PKCS #7 ストリーミング PKCS7 デコーディングのためのストリーム指向 API は、すべてを一度に渡すのではなく、より小さなチャンクで入力するオプションを提供します。デフォルトでは、PKCS7 のストリーミング機能はオンになっています。PKCS7 API のストリーミングサポートをオフにするには、マクロ NO_PKCS7_STREAM を定義してください。autoconf で行う場合は、./configure --enable-pkcs7 CFLAGS=-DNO_PKCS7_STREAM とします。

バンドルのデコード/検証のためのストリーミングでは、以下の関数がサポートされています。

1. `wc_PKCS7_DecodeEncryptedData()`
2. `wc_PKCS7_VerifySignedData()`
3. `wc_PKCS7_VerifySignedData_ex()`
4. `wc_PKCS7_DecodeEnvelopedData()`
5. `wc_PKCS7_DecodeAuthEnvelopedData()`

注意: `wc_PKCS7_VerifySignedData_ex` を呼び出す場合、引数 `pkiMsgFoot` は完全なバッファであることが想定されています。内部構造は 1 つのバッファのストリーミングのみをサポートしており、この場合は `pkiMsgHead` になります。

4.3.9 特定の暗号スイートの使用を強制する

デフォルトでは、wolfSSL は接続の両側がサポートしている最良の暗号スイートを選択します。128 ビット AES などの特定の暗号を強制するには、次のような内容を追加します。

```
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

`wolfSSL_CTX_new()` の呼び出しの後で、次のように実装してください。

```
ctx = wolfSSL_CTX_new(method);  
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

4.3.10 OpenQuantumSafe の liboqs 統合

付録 G ポスト量子暗号の実験をご覧ください。

4.4 ハードウェアによる支援機構の活用

wolfSSL は、さまざまなプロセッサやチップに搭載されたハードウェアアクセラレーテッド機能をいくつか活用できます。ここでは、wolfSSL がサポートしているテクノロジーについて説明します。

4.4.1 AES-NI

AES は世界中の政府によって使用される鍵暗号化標準です。Intel は、AES をより高速に処理する新しい命令セットをリリースしました。wolfSSL は、商用環境でこの命令セットを完全にサポートする最初の SSL ライブラリです。

基本的に、Intel と AMD は AES アルゴリズムの計算集約的な部分を実行するチップレベルの AES 命令を追加し、パフォーマンスを向上させています。AES-NI をサポートする Intel のチップのリストは、こちらで確認できます。

<https://ark.intel.com/search/advanced/?s=t&AESTech=true>

wolfSSL では、ソフトウェアでアルゴリズムを実行する代わりにチップから直接命令を呼び出す機能を追加しました。これにより、AES-NI をサポートするチップセットで wolfSSL を実行すると、AES 暗号を 5~10 倍高速に実行できるようになりました。

AES-NI をサポートするチップセットで実行している場合は、ビルドオプション `--enable-aesni` を使用します。AES-NI を使用して wolfSSL をビルドする場合、アセンブリコードを使用するために GCC 4.4.3 以降が必要です。wolfSSL は、同一のビルドオプションを使用して AMD プロセッサ上の ASM 命令もサポートしています。

AES-NI に関する文献を、いくつか以下に示します。AES-NI によるパフォーマンス向上に関する情報については、3 つ目の Intel Software Network ページをご参照ください。

- [AES \(Wikipedia\)](#)
- [AES-NI \(Wikipedia\)](#)
- [AES-NI \(Intel Software Network\)](#)

AES-NI は、AES 暗号モード AES-CBC、AES-GCM、AES-CCM-8、AES-CCM、AES-CTR に対応しています。AES-GCM は、GHASH 認証用に Intel チップに追加された 128 ビット乗算関数を使用することでさらに高速になります。

4.4.2 STM32F2

wolfSSL は、STM32F2 標準周辺機器ライブラリを通じて、STM32F2 ハードウェアベースの暗号処理および乱数生成器を使用できます。

使用するには、`settings.h` に `WOLFSSL_STM32F2` を定義してください。STM32F2 ハードウェア暗号と RNG サポートを有効にします。これらを個別に有効にするには、`STM32F2_CRYPT0`（ハードウェア暗号サポート用）と `STM32F2_RNG`（ハードウェア RNG サポート用）を使用します。

STM32F2 標準周辺機器ライブラリの詳細は、次のドキュメントで確認できます。

https://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/USER_MANUAL/DM00023896.pdf

4.4.3 Cavium NITROX

wolfSSL は Marvell（旧称：Cavium）NITROX (<https://www.marvell.com/products/security-solutions.html>) をサポートしています。Marvell NITROX サポートを有効にするには、wolfSSL をビルドする際に次のオプションを使用します。

```
./configure --with-cavium=/home/user/cavium/software
```

`--with-cavium=**` オプションは、ライセンスされた `cavium/software` ディレクトリを指しています。Cavium はライブラリをビルドしないため、wolfSSL は `cavium_common.o` ファイルを取り込み、これによってこの移植性に関する `libtool` 警告が表示されます。また、GitHub ソースツリーを使用している場合は、Cavium ヘッダーがこの警告に準拠していないため、生成された Makefile から `-wredundant-decls` 警告を削除する必要があります。

現在、wolfSSL は暗号レイヤーで直接 Cavium RNG、AES、3DES、RC4、HMAC、RSA をサポートしています。SSL レベルでのサポートは部分的であり、現在は AES、3DES、RC4 のみです。Cavium 呼び出しを非ブロッキングモードで利用できるようになるまで、RSA と HMAC は遅くなります。クライアントのサンプルプログラムでも、暗号テストとベンチマークと同様に Cavium サポートをオンにできます。HAVE_CAVIUM を定義してください。

4.4.4 ESP32-WROOM-32

wolfSSL は ESP32-WROOM-32 ハードウェアベースの暗号化を使用できます。

使用するには、`settings.h` に `WOLFSSL_ESP32WROOM32` を定義します。ESP32-WROOM-32 ハードウェア暗号と RNG サポートが有効になります。現在、wolfSSL は暗号レイヤーで RNG、AES、SHA、RSA プリミティブをサポートしています。TLS サーバー/クライアント、wolfCrypt テスト、ベンチマークを含むサンプルプロジェクトは、ファイルをデプロイした後の ESP-IDF の `/examples/protocols` ディレクトリにあります。

4.4.5 ESP8266

ESP32 とは異なり、ESP8266 ではハードウェアベースの暗号化は利用できません。ESP8266 用にコンパイルするには、`user_settings.h` に `WOLFSSL_ESP8266` を定義します。autoconf により `./configure CFLAGS="-DWOLFSSL_ESP8266"` とすることも可能です。

4.4.6 EFR32

wolfSSL は EFR32 ファミリーのハードウェアベースの暗号化を使用できます。

サポートを有効にするには、`user_settings.h` で `WOLFSSL_SILABS_SE_ACCEL` を定義します。wolfSSL は現在、EFR32 プラットフォーム上で RNG、AES-CBC、AES-GCM、AES-CCM、SHA-1、SHA-2、ECDHE、ECDSA のハードウェアアクセラレーションをサポートしています。

詳細とベンチマークは、wolfSSL リポジトリの `wolfcrypt/src/port/silabs/README.md` をご覧ください。

4.4.7 MAX32665/MAX32666

wolfSSL は、Analog Devices の [MAX32666/MAX32665](#) マイクロコントローラーのサポートモデルに搭載されている Trust Protection Unit (TPU)、Modular Arithmetic Accelerator (MAA)、TRNG の使用をサポートしています。

サポートを有効にするには、`WOLFSSL_MAX3266X` と `WOLFSSL_SP_MATH_ALL` を定義します。wolfSSL は現在、RNG、AES-CBC、AES-GCM、AES-ECB、SHA-1、SHA-2、RSA 2048、および ECDSA のハードウェアアクセラレーションをサポートしています。

このハードウェアは、ハードウェアとソフトウェア実装の両方を使用できるようにする wolfSSL の暗号コールバック機能の使用もサポートしています。

サポートの詳細は、wolfSSL リポジトリの `wolfcrypt/src/port/maxim/README.md` をご覧ください。

4.5 SSL 検査 (スニファアー)

wolfSSL 1.5.0 リリース以降、wolfSSL には SSL スニファアー (SSL 検査) 機能でビルドできるビルドオプションが含まれています。これは、SSL トラフィックパケットを収集し、正しい鍵ファイルを使用して復号できることを意味します。SSL トラフィックを「検査」する機能は、以下のような場面で役立ちます。

- ネットワーク問題の分析
- 内部および外部ユーザーによるネットワーク誤用の検出
- ネットワーク使用状況とデータコピーの監視
- クライアント/サーバー通信のデバッグ

スニファアーサポートを有効にするには、*nix では `--enable-sniffer` オプション、Windows では **vcproj** ファイルを使用して wolfSSL をビルドします。*nix では **pcap** を、Windows では **WinPcap** をインストールする必要があります。sniffer.h で見つけることができる主なスニファアー関数を、各機能の簡単な説明と共に以下に示します。

- `ssl_SetPrivateKey` - 特定のサーバーとポートの秘密鍵を設定します。
- `ssl_SetNamedPrivateKey` - 特定のサーバー、ポート、ドメイン名の秘密鍵を設定します。
- `ssl_DecodePacket` - 復号のために TCP/IP パケットを渡します。
- `ssl_Trace` - トレースファイルへのデバッグトレースを有効/無効にします。
- `ssl_InitSniffer` - 全体的なスニファアーを初期化します。
- `ssl_FreeSniffer` - 全体的なスニファアーを解放します。
- `ssl_EnableRecovery` - パケット損失の場合に SSL トラフィックの復号を再開しようとするオプションを有効にします。
- `ssl_GetSessionStats` - スニファアーセッションのメモリ使用量を取得します。

wolfSSL のスニファアーサポートに関する完全なサンプルとして、wolfSSL の `sslSniffer/sslSniffer-Test/sniffTest` アプリをご参照ください。

暗号化鍵は SSL ハンドシェイクで設定されるため、その後のアプリケーションデータを復号するためにはスニファアーによってハンドシェイクを復号する必要があることにご注意ください。例えば、wolfSSL のサンプルプログラム `echoserver` と `echoclient` で「sniffTest」を使用している場合、サーバーとクライアント間のハンドシェイクが始まる前に `sniffTest` アプリケーションを起動する必要があります。

スニファァは、AES-CBC、DES3-CBC、ARC4、Camellia-CBC で暗号化されたストリームのみを復号できません。ECDHE または DHE 鍵合意が使用されている場合、ストリームをスニッフすることはできません。RSA 及び ECDH 鍵交換のみがサポートされています。

wolfSSL スニファァでのウォッチコールバックは、WOLFSSL_SNIFFER_WATCH で有効にできます。スニファァウォッチ機能がコンパイルされている場合、関数 `ssl_SetWatchKeyCallback()` を使用してカスタムコールバックを設定できます。このコールバックは、証明書チェーン、エラー値、ピアから送信された証明書のダイジェストを検査するために使用されます。コールバックからゼロ以外の値が返された場合、ピアの証明書を処理する際にエラー状態が設定されます。ウォッチコールバックをサポートする追加関数は次のとおりです。

- `ssl_SetWatchKeyCtx`：ウォッチコールバックに渡されるカスタムユーザーコンテキストを設定します。
- `ssl_SetWatchKey_buffer`：新しい DER 形式の鍵をサーバーセッションにロードします。
- `ssl_SetWatchKey_file`：`ssl_SetWatchKey_buffer` のファイルバージョンです。

スニファァでの統計収集を有効にするには、マクロ `WOLFSSL_SNIFFER_STATS` を定義します。統計は `SSLStats` 構造体に保持され、`ssl_ReadStatistics` を呼び出すことでアプリケーションの `SSLStats` 構造体にコピーされます。スニファァ統計で使用する追加 API は `ssl_ResetStatistics`（統計収集をリセット）と `ssl_ReadResetStatistics`（現在の統計値を読み取り、内部状態をリセット）です。現在保持される統計値は、以下の通りです。

- `sslStandardConns`
- `sslClientAuthConns`
- `sslResumedConns`
- `sslEphemeralMisses`
- `sslResumeMisses`
- `sslCiphersUnsupported`
- `sslKeysUnmatched`
- `sslKeyFails`
- `sslDecodeFails`
- `sslAlerts`
- `sslDecryptedBytes`
- `sslEncryptedBytes`
- `sslEncryptedPackets`
- `sslDecryptedPackets`
- `sslKeyMatches`
- `sslEncryptedConns`

4.6 静的バッファ確保オプション

wolfSSL では、**動的**メモリ管理が提供されていることを前提に処理が記述されています。すなわち、`malloc/free` 関数でバッファを確保/解放できることを前提としています。しかし、wolfSSL が内部で使用している暗号化ライブラリ `wolfCrypt` では、動的メモリを使用しない設定も可能です。これは動的メモリ管理機能を提供していない環境や、あるいは安全面の制約で動的メモリ管理機能の使用が制限されている環境では好ましいと言えます。具体的には、組み込み機器で OS を使用しない（いわゆるベアメタル）環境で使用する場合、あるいはリアルタイム OS を使用している場合に動的メモリ管理機能が提供されていない場合があります。このような環境で wolfSSL を使用方法として**静的バッファ確保オプション**を提供しています。

4.6.1 静的バッファ確保の基本操作

先に説明した「動的メモリ管理」はバッファを「指定されたサイズ（可変長）」に動的に切り出して提供する管理方法です。バッファの使用効率は高いですが処理が比較的複雑です。一方、wolfSSL が提供する「静的バッファ確保機能」は、あらかじめ（静的に）用意した何種類かのバッファの中から、要求したサイズに近いものを検索して提供するメモリ管理機能のことを言います。バッファの要求元には要求サイズ以上の

大きさを持ったメモリブロックが割り当てられることがあります（そのため使用効率が低下します）が、バッファの確保と解放は `malloc/free` と同様に行え、割り当て処理は単純で済みます。まとめると、任意サイズのメモリブロックを動的に確保する動的メモリ確保を模擬した機能となっています。

静的バッファ確保機能の使用は、wolfSSL にとって動的メモリ機能と等価に使用されます。この機能は wolfSSL がメモリの確保/解放を `XMALLOC/XFREE` 関数呼び出しに抽象化してあることで実現できています。一旦、静的バッファ確保機能が設定されると、以降 wolfSSL は内部で使用するバッファや他の構造体の確保に静的バッファ確保機能を使用します。

静的バッファ割り当ては `WOLFSSL_CTX` で設定され、スレッドセーフです。同一の `WOLFSSL_CTX` を異なるスレッドで共有しながら使用している場合でも、バッファの確保/解放は wolfSSL 内部で排他制御しながら利用します。以下はコンテキスト構造を図示したもので、矢印はヒープへのポインタの受け渡しを示します。「...」はリストに挙げられたものだけでなく、すべての構造体を参照しています。

また、RTOS で使用されているメモリプールを使ったメモリ機能では、未使用のメモリブロックが見つからない場合にはそのスレッド（タスク）は空きブロックが発生するまでサスペンドされますが、wolfSSL の静的バッファ確保機能にはそのような待ち合わせ機能はありません。

4.6.2 静的バッファ使用の指定

静的バッファ確保機能では、2つの用途別にメモリを分けることが可能です。つまり、一般的な目的用と I/O 用に使用するバッファを分けて割り当て/解放を行うことが可能です。I/O に使用するバッファは、TLS での典型的必要サイズから最大 2^{16} バイトまでを扱えるよう、比較的大きく（17KB 程度）設定されており、それ以外の一般的な用途のバッファサイズと異なります。また I/O を行うスレッド（タスク）とそれ以外のスレッドでのメモリ獲得/解放に伴う排他制御を排除するためにも、用途別に分けることを推奨しています。さらにバッファを設定した際、同時に生成できる `WOLFSSL` オブジェクトの最大数を制限することができます。最大数を制限すると `wolfSSL_new()` 関数を実行する度に、生成できる `WOLFSSL` オブジェクトの数がチェックされ、上限を超えるとエラーとなります。

4.6.3 静的バッファ確保の有効化

wolfSSL をビルドする際に静的バッファ確保オプションを有効にします。autoconf を使用するシステムの場合、次のように `--enable-staticmemory` を指定します。

```
$ ./configure --enable-staticmemory
```

`user_settings.h` ヘッダーを使用している場合、次のマクロ定義を追加します。

```
user_settings.h
```

```
#define WOLFSSL_STATIC_MEMORY
```

静的バッファ確保機能は、与えられたバッファから確保するメモリブロックが枯渇した際、NULL を返さず標準関数の `malloc()` を呼び出す実装となっています。環境に動的メモリ管理機能が提供されていない場合には、ここでリンクエラーとなります。したがって、この機能を無効化する為に `WOLFSSL_NO_MALLOC` マクロも定義します。

```
user_settings.h
```

```
#define WOLFSSL_STATIC_MEMORY
#define WOLFSSL_NO_MALLOC
```

さらに、2つのビルド構成があります。`--enable-staticmemory=small` は、構造体のサイズが小さく、利用可能な API も限定される小さなバージョンです。もう1つは `--enable-staticmemory=debug` で、コールバック関数を設定する機能を有効にします。これは `printf()` が利用できない状況で、何が割り当てられどのバケットサイズが使用されているかを調査する際に有用です。以下に、サンプルコールバックを使用したクライアントの出力例を示します。

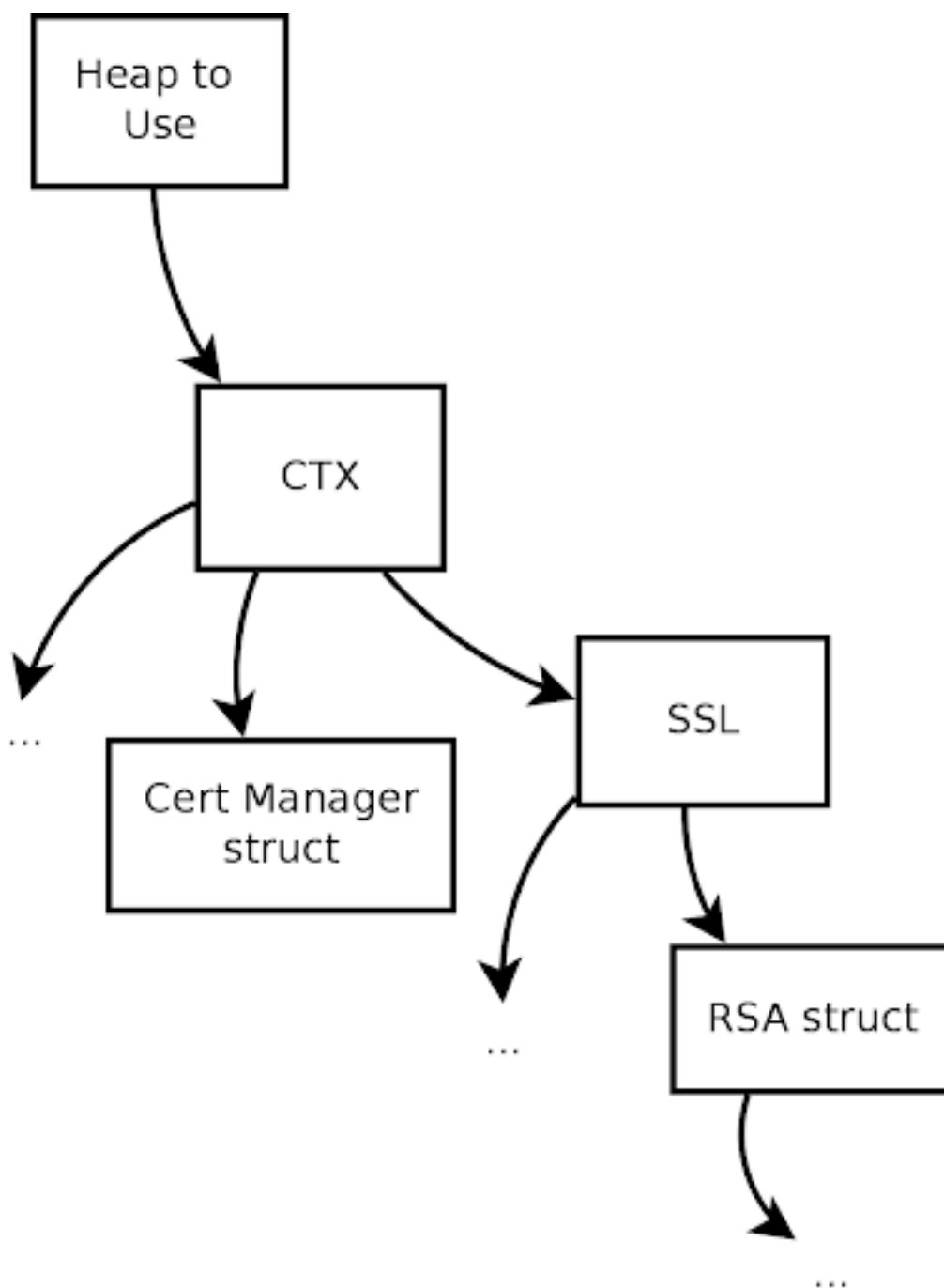


Figure 2: コンテキスト構造

```
./examples/client/client
...
...
...
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 静的バッファ確保の使用

上記設定で wolfSSL をビルドすることにより、静的バッファ確保機能が有効となります。

4.6.4.1 静的バッファ設定関数とその引数 この機能を利用する際、アプリケーションは次の関数を呼び出してヒープとして使用するバッファを指定します。

```
int wolfSSL_CTX_load_static_memory(
    WOLFSSL_CTX** ctx,          /* 生成した WOLFSSL_CTX を受け取る為の変数へのポイン
    ↪ タ */
    wolfSSL_Methos_func method, /* メソッド */
    unsigned char* buf,         /* ヒープとして使用するバッファへのポインタ */
    unsigned int sz,            /* ヒープとして使用するバッファのサイズ */
    int flag,                   /* ヒープの使用用途 */
    int max);                  /* 許可する最大並行動作数 */
```

- 引数 **ctx** には、生成された WOLFSSL_CTX 構造体へのポインタを受け取る変数のアドレスを指定します。
- 引数 **method** には wolfSSLv23_client_method_ex() などの ****_ex**** が付いた関数の戻り値を指定します。
 - 使用できる関数は、後の章に掲載しています。
- 引数 **buf, sz** には、それぞれヒープに使用するバッファのアドレスとそのサイズを指定します。
 - 設定するバッファサイズの決定に関する情報については、「**必要バッファサイズの取得**」をご参照ください。
- 引数 **flag** には、一般用途あるいは I/O 用を指定する用途と静的バッファの確保状況のトラッキングを行うかどうかのフラグを組み合わせて指定します。
 - 一般用途を指定する場合には、0 あるいは WOLFMEM_GENERAL を指定します。
 - I/O 用として指定する場合には、WOLFMEM_IO_POOL あるいは WOLFMEM_IO_POOL_FIXED を指定します。

- 静的バッファの確保状況のトラッキングを行う場合には、用途を指定する値に WOLFSSL_TRACK_STATS を OR で加えて指定します。
- 引数 **max** は、引数 flag で指定したバッファの用途に関係します。
 - バッファの用途が一般用の場合には、生成する WOLFSSL オブジェクトの最大同時生成数（同時に存在できるオブジェクト数）を設定することになります。
 - 制限を行う必要がなければ 0 を指定します。
 - 0 以外の制限値を指定した場合には、その後の wolfSSL_new() の呼び出しで、生成する WOLFSSL オブジェクトの同時オブジェクト数が設定値を超える際には生成が失敗することになります。

4.6.4.2 静的バッファ設定関数の呼び出し方法 静的バッファ確保オプションを使用する場合、関数 wolfSSL_CTX_load_static_memory() を **2 回**呼び出します。最初の呼び出しでは一般的な使用のためのバッファを設定し、そのバッファを使用して WOLFSSL_CTX 構造を割り当てます。2 回目の呼び出しで I/O バッファを設定します。

```
WOLFSSL_CTX* ctx = NULL; /* WOLFSSL_CTX を生成する場合には NULL を指定 */
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];
/* 最初の呼び出しで一般用静的バッファを設定してそこから WOLFSSL_CTX を生成する */
ret = wolfSSL_CTX_load_static_memory(
    &ctx, /* ctx 変数の内容には NULL をセットしておく */
    wolfSSLv23_client_method_ex(), /* "_ex" の付いた関数を使う */
    GEN_MEM, GEN_MEM_SIZE, /* 一般用途のバッファとそのサイズ */
    WOLFMEM_GENERAL, /* 一般用途で使用 */
    MAX_CONCURRENT_TLS); /* 最大許容同時接続数 */

/* 次は生成した WOLFSSL_CTX に I/O 用静的バッファをセットする */
ret = wolfSSL_CTX_load_static_memory(
    &ctx, /* ctx には生成済み構造体が格納されている */
    NULL, /* 今後は ctx の生成は行わないので NULL をセッ
    ト */
    IO_MEM, IO_MEM_SIZE, /* I/O 用途のバッファとそのサイズ */
    WOLFMEM_IO_FIXED,
    MAX_CONCURRENT_IO);
```

WOLFSSL_CTX 構造体の使用を終了した後は、通常の wolfSSL_CTX_free() を使って解放してください。

4.6.5 TLS なしでの静的メモリの使用（暗号のみ）

wolfSSL は、TLS 機能を必要とせずに、wolfCrypt 操作のための静的メモリ割り当ての使用もサポートしています。これは、TLS 接続のオーバーヘッドなしに暗号操作（ハッシュ化、暗号化、乱数生成など）のみを必要とするアプリケーションに役立ちます。

wc_LoadStaticMemory() 関数は、wolfCrypt で使用するための静的メモリを確保するために使用されます。メモリは、作成されたヒープヒントをそれをサポートする関数に渡すことで使用できます。この例として、wc_InitRng_ex() を呼び出す場合があります。

例:

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
int flag = WOLFMEM_GENERAL;

// 使用するメモリをロード
ret = wc_LoadStaticMemory(&hint, memory, memorySz, flag, 0);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}
```

```
ret = wc_InitRng_ex(&rng, hint, 0);
// ret 値を確認
```

グローバルヒープヒントを使用する例:

```
WOLFSSL_HEAP_HINT hint;
void* oldHint;
int ret;
unsigned char memory[MAX];

// 静的メモリをセットアップ
ret = wc_LoadStaticMemory(&hint, memory, MAX, WOLFMEM_GENERAL, 0);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}

// グローバルヒープヒントとして設定 (スレッドセーフではない)
oldHint = wolfSSL_SetGlobalHeapHint(hint);

// これ以降、すべての NULL ヒープヒント呼び出しはこのグローバルヒントを使用する
// ... wolfSSL 関数を使用 ...
```

```
// 完了時に以前のグローバルヒープヒントを復元
wolfSSL_SetGlobalHeapHint(oldHint);
wc_UnloadStaticMemory(hint);
```

カスタムバッファサイズを使用する例:

```
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];

// カスタム設定に必要なバッファサイズを計算
requiredSize = wolfSSL_StaticBufferSz_ex(5, customSizes, customDist,
                                         NULL, 0, WOLFMEM_GENERAL);
printf(" 必要なバッファサイズ: %d バイト\n", requiredSize);

// メモリバッファを割り当て
if (requiredSize <= MAX) {
    // カスタムバケット設定で静的メモリをロード
    ret = wc_LoadStaticMemory_ex(&hint, 5, customSizes, customDist,
                                memory, requiredSize, WOLFMEM_GENERAL, 0);
}
```

```

if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}

// カスタム設定された静的メモリを使用
// ... ヒントを使用して wolfSSL 関数を使用 ...

// 完了時にクリーンアップ
wc_UnloadStaticMemory(hint);
}

// パディングオーバーヘッドを計算
int paddingSize = wolfSSL_MemoryPaddingSz();
printf(" バケットごとのメモリパディング: %d バイト\n", paddingSize);

```

4.6.5.1 グローバルヒープヒントの設定 関数 `void* wolfSSL_SetGlobalHeapHint(void* heap)` を使用して、グローバルヒープヒントを設定できます。

グローバルヒープヒントが設定されると、NULL ポインタをヒープヒントとして使用する `XMALLOC` および `XFREE` へのすべての呼び出しが、設定されたグローバルヒープヒントを使用するようにリダイレクトされます。これは、使用するシステムに `malloc` がなく、NULL ヒープヒントポインタが使用されている場合に便利です。

関数 `wolfSSL_SetGlobalHeapHint()` は現在設定されているグローバルヒープヒントを返します。スレッドセーフとは見なされません。

ゲッター関数 `void* wolfSSL_GetGlobalHeapHint(void)` を使用して、現在設定されているグローバルヒープヒントを取得できます。

この機能は wolfSSL 5.7.0 で追加されました。

4.6.5.2 デバッグメモリコールバック `WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACK` ビルドオプションを使用して静的メモリ割り当てを使用する場合、`void wolfSSL_SetDebugMemoryCb(DebugMemoryCb cb)` を使用してデバッグコールバック関数を設定できます。このコールバックはメモリの割り当てと解放操作中に呼び出され、メモリ使用に関するデバッグ情報を提供します。

コールバック関数のシグネチャは次のとおりです。

```
typedef void (*DebugMemoryCb)(size_t sz, int bucketSz, byte st, int type);
```

各パラメータの詳細は次のとおりです。

- `sz`: 要求されたサイズ
- `bucketSz`: 使用されたバケットサイズ
- `st`: ステータス (0= 割り当て、1= 失敗、2= 解放、3= 初期化)
- `type`: メモリタイプ

4.6.5.3 静的メモリのアンロード 静的メモリ割り当ての使用が終了したら `void wc_UnloadStaticMemory(WOLFSSL heap)` を呼び出して、静的メモリヒープと関連するミューテックスを適切に解放する必要があります。これにより、リソースの適切なクリーンアップが保証されます。

4.6.5.4 高度な静的バッファサイズの計算 関数 `int wolfSSL_StaticBufferSz_ex(unsigned int listSz, const word32 *sizeList, const word32 *distList, byte* buffer, word32 sz, int flag)` は、提供されたバケットサイズと分布に基づいて、特定のバッファのうちどれだけが静的メモリ割り当てに効果的に使用できるかを計算する事前割り当て計画ツールです。使用できない余剰スペース

を除いた、完全なバケットに適合するバッファの使用可能な部分を返します。静的メモリプールのサイジングとチューニングを行うことで利用効率を最大化し、全体的なメモリ使用量を削減するのに役立ちます。

各パラメータの詳細は次のとおりです。

- listSz: リスト内のバケットサイズの数
- sizeList: バケットサイズの配列
- distList: 各バケットサイズの分布カウンタの配列
- buffer: テストするバッファ（計算のみの場合は NULL も可）
- sz: バッファのサイズ
- flag: メモリフラグ（WOLFMEM_GENERAL、WOLFMEM_IO_POOL など）

4.6.5.5 メモリパディングの計算 関数 `int wolfSSL_MemoryPaddingSz(void)` は、アライメントパディングを含む、各メモリバケットに必要な管理メモリのサイズを計算します。これは、静的メモリ割り当てのオーバーヘッドを理解し、正確なメモリ計画を立てるのに役立ちます。

4.6.6 静的バッファ確保の調整

wolfSSL で提供しているこの静的バッファ確保機能では、指定されたバッファを次の図のように複数の「バケット (bucket)」という領域に分けて管理します。バケット内には同一サイズの複数のメモリブロックがリンクされています。下図ではメモリブロックの管理のための構造は省略しています。実際にはそれらの管理領域も含めて必要なサイズを持つバッファが与えられなければなりません。

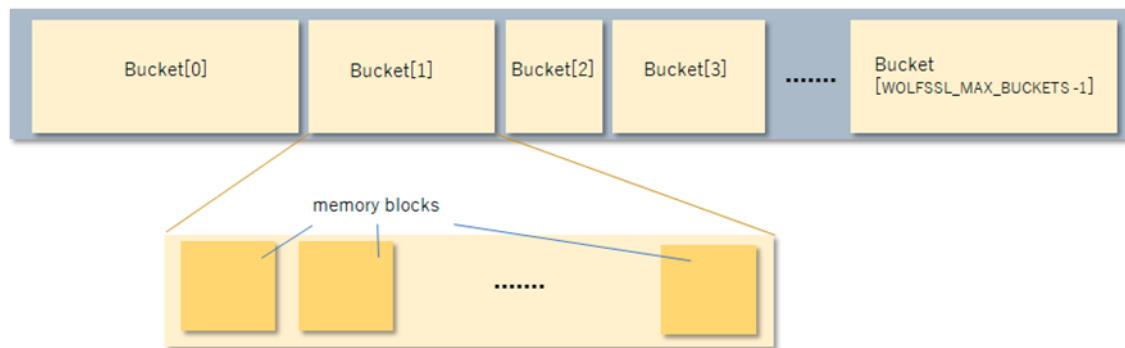


Figure 3: バケット

上記構造は一般用、I/O 用のいずれのバッファにも適用されますが、I/O 用バッファには Bucket は一種類しかありません。

4.6.6.1 一般用バッファを設定するためのマクロ 各バケットは、含まれるメモリブロックの数とそのサイズによってサイズが異なります。

各領域で使用するメモリブロックのサイズとブロック数は、`wolfssl/wolfcrypt/memory.h` の次のマクロで定義されています。

`/wolfssl/wolfcrypt/memory.h`

```
#define WOLFSSL_STATIC_ALIGN 16 /* 適用されるアライメント (デフォルト 16 バイト) */
#define WOLFMEM_MAX_BUCKETS 9 /* バケット数 */
#define WOLFMEM_IO_SZ 16992 /* I/O 用のバッファサイズ */
#define LARGEST_MEM_BUCKET 16128 /* 最大ブロックのサイズ */
#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** にて、バッファのアライメントサイズを指定します。
 - デフォルトで 16 バイトです。
 - ご使用の MCU でのアライメントサイズに合わせて変更する必要があります。
- **WOLFMEM_MAX_BUCKETS** は、バケットの数を示しています。
 - 9 種類のサイズのバケットを使用することを意味しています。
- **WOLFMEM_BUCKETS** では、各バケット内のブロックのバイト数を小さいものから順にコンマ区切りで指定しています。
 - この定義は一般用バッファに適用されます。
- **WOLFMEM_DIST** では、各バケットに含まれる同一サイズのブロックの数を **WOLFMEM_BUCKETS** の各ブロックに対応するように、それらの個数をコンマ区切りで指定しています。
 - この定義は一般用バッファに適用されます。

上記の例でいえば、ブロックサイズ 64 バイトのバケットが最小のサイズであり、そのバケットには 49 個のメモリブロックを用意することになります。次に大きいバケットはブロックサイズ 128 バイトで 10 個のメモリブロックを用意することを意味します。上記定義値はデフォルトの値として使用していただけますが、実際の環境での使用時には各バケットのサイズとそこに含まれるメモリブロック数は調整が必要かもしれません。

4.6.6.2 I/O 用バッファの設定用マクロ定義 I/O 用途のバッファは上記一般用途と管理方法は同じですが、バケット数は 1、バケット内のメモリブロックは 1 つだけです。またメモリブロックのサイズは、**WOLFMEM_IO_SZ** で定義された値となっています。この I/O バッファのサイズは、TLS ハンドシェイクで送受信される最大パケットサイズを考慮して設定されています。**wolfSSL_CTX_UseMaxFragment()** を使用して、より小さい値に設定することも可能です。この関数を使って最大パケットサイズを小さくした場合には、その値（この例では 660 バイト）を **WOLFMEM_IO_SZ** として設定してください。

```
./configure --enable-staticmemory C_EXTRA_FLAGS="-DWOLFMEM_IO_SZ=660"
```

4.6.6.3 必要バッファサイズの取得 静的バッファ確保機能に割り当てるバッファサイズ（すなわち関数 **wolfSSL_CTX_load_static_memory()** に渡すバッファサイズ）の決定に有用な関数を用意してあります。この章の冒頭で、バッファが内部でバケットと管理領域からなる構造によって構成、使用されることを説明しました。実際にバッファのサイズを決定する際には、メモリブロックのサイズと管理領域の占めるサイズ、そしてパディングによる余分に必要なサイズも含めて計算する必要があります。この計算を行い、必要なバッファサイズを返してくれる関数を用意してあります。

関数 **wolfSSL_StaticBufferSz()** は、直前のセクションで紹介したマクロ定義値を基に、必要なバッファサイズを計算して返します。この関数が 0 より大きい値を返すまで、第 2 引数に与えるサイズは 1000 などの適当な値からスタートし、戻り値が正となるまでサイズを増やして何度も呼び出します。第 3 引数の **flag** には、一般用バッファサイズを計算する場合には **WOLFMEM_GENERAL** を、I/O 用バッファサイズを計算する場合には **WOLFMEM_IO_FIXED** あるいは **WOLFMEM_IO_POOL** を指定してください。

```
int wolfSSL_StaticBufferSz(byte* buffer, /* バッファアドレス */
                           word32 sz,   /* バッファサイズ */
                           int flag);    /* バッファの用途 */
```

この関数を使って、一般用と I/O 用に必要なバッファサイズをそれぞれ取得し、前述の関数 **wolfSSL_CTX_load_static_memory()** に渡してください。一旦バッファサイズが決定したら、関数 **wolfSSL_StaticBufferSz()** は呼び出す必要はありません。製品コードから呼び出し部分をコメントアウトまたは削除していただけます。

4.6.7 静的バッファ利用状況のトラッキング

静的バッファ確保機能を利用する際に、バッファの確保、解放に関する使用状況を記録させることができます。この使用状況を記録する機能は、静的バッファ確保機能にデフォルトで含まれています。従って、ビルド時は有効化するための別段のマクロ設定は必要ありません。機能の有効化は実行時に行います。

4.6.7.1 トラッキングの有効化 トラッキング機能の有効化は、先に説明した関数 `wolfSSL_CTX_load_static_memory()` の第 5 引数に **WOLFMEM_TRACK_STATS** を OR で加えて指定します。`wolfSSL` 内部に構造体 `WOLFSSL_MEM_CONN_STATS` が確保され、そこにメモリブロックの使用状況が記録されていきます。

```
wolfssl/wolfcrypt/memory.h
struct WOLFSSL_MEM_CONN_STATS {
    word32 peakMem; /* メモリ使用量最大値 (バイト数) */
    word32 curMem; /* 現在のメモリ使用量 */
    word32 peakAlloc; /* メモリ確保量最大値 */
    word32 curAlloc; /* 現在のメモリ確保数 */
    word32 totalAlloc; /* 累計メモリ確保回数 */
    word32 totalFr; /* 累計メモリ解放回数 */
};
```

4.6.7.2 メモリ使用状況の取得 トラッキングが有効になった時点から、メモリブロックの使用状況記録が有効となります。プログラム実行の任意の時点で次の関数を呼び出して、引数で渡した `WOLFSSL_CTX` あるいは `WOLFSSL` オブジェクトに静的バッファ確保機能が使用されているか否かを戻り値で返します。使用されている場合には戻り値として 1 を返します。さらに、記録されているメモリの使用状況を取得できます。

```
int wolfSSL_CTX_is_static_memory(WOLFSSL_CTX* ctx,
                                WOLFSSL_MEM_STATS* mem_stats);
int wolfSSL_is_static_memory(WOLFSSL* ssl,
                              WOLFSSL_MEM_STATS* mem_stats);
```

上記関数の引数として渡した `WOLFSSL_MEM_STATS` 構造体は、次のように構成されています。

```
struct WOLFSSL_MEM_STATS {
    word32 curAlloc; /* 現在のメモリ確保数 */
    word32 totalAlloc; /* 累計メモリ確保回数 */
    word32 totalFr; /* 累計メモリ解放回数 */
    word32 totalUse; /* N/A */
    word32 avaIO; /* I/O 用ブロックの空きブロック数 */
    word32 maxHa; /* 一般用に設定した最大コネクション数 */
    word32 maxIO; /* I/O 用に設定した最大コネクション数 */
    word32 blockSz[WOLFMEM_MAX_BUCKETS]; /* ブロックサイズの配列 */
    word32 avaBlock[WOLFMEM_MAX_BUCKETS]; /* 空きブロック数の配列 */
    word32 usedBlock[WOLFMEM_MAX_BUCKETS];
    int flag; /* 静的メモリ管理機能に設定したフラグ (バッファ用途等) */
};
```

この構造体に返却された値が、呼び出した時点における静的バッファ管理状態を示します。この機能はメモリリークの検出や、無駄に多く割り当てたメモリブロック数の調査などに利用できます。

4.6.8 静的バッファ管理 API

ここで紹介する関数は関数 `wolfSSL_CTX_load_static_memory()` の第 2 引数に指定する `WOLFSSL_METHOD` 構造体へのポインタを取得する為に使用する関数です。これらの関数はアプリケーションを TLS あるいは DTLS プロトコルの、クライアントあるいはサーバーのいずれとして動作させるのかによって選択すべき関数が異なります。静的バッファ管理機能を使う際には、必ず関数名の最後に `_ex` が付加された以下の関数を指定してください。これらの関数は `_ex` がつかない関数と機能は同じで、内部で `WOLFSSL_METHOD` 構造体を、有効になった静的バッファ管理機能を使って確保する点だけが異なります。各関数の詳細は付録 A `wolfSSL` API リファレンスの「`wolfSSL` コンテキストの設定」節を参照してください。

4.6.8.1 WOLFSSL_METHOD 構造体を返す関数 TLS クライアント用

- wolfTLSv1_3_client_method_ex
- wolfTLSv1_2_client_method_ex
- wolfTLSv1_1_client_method_ex
- wolfSSLv23_client_method_ex

TLS サーバー用

- wolfTLSv1_3_server_method_ex
- wolfTLSv1_2_server_method_ex
- wolfTLSv1_1_server_method_ex
- wolfSSLv23_server_method_ex

DTLS クライアント用

- wolfDTLSv1_3_client_method_ex
- wolfDTLSv1_2_client_method_ex
- wolfDTLSv1_1_client_method_ex
- wolfSSLv23_client_method_ex

DTLS サーバー用

- wolfDTLSv1_3_server_method_ex
- wolfDTLSv1_2_server_method_ex
- wolfDTLSv1_1_server_method_ex
- wolfSSLv23_server_method_ex

4.6.8.2 静的バッファ確保用の API

API	説明
wolfSSL_CTX_load_static_memory	WOLFSSL_CTX に静的バッファ確保用のバッファを設定します。 静的バッファ確保が設定されているかと、設定されている場合にはその使用状況を取得します。 静的バッファ確保が設定されているかと、設定されている場合にはその使用状況を取得します。 wolfCrypt 用に静的メモリを確保するために使用されます。 カスタムバケットサイズと分布で wolfCrypt で使用する静的メモリを確保するために使用されます。
wolfSSL_CTX_is_static_memory	
wolfSSL_is_static_memory	
wc_LoadStaticMemory	
wc_LoadStaticMemory_ex	
wolfSSL_SetGlobalHeapHint	メモリ割り当て関数に NULL ヒープヒントが渡されたときに使用されるグローバルヒープヒントを設定します。以前のグローバルヒープヒントを返します。
wolfSSL_GetGlobalHeapHint	
wolfSSL_SetDebugMemoryCb	メモリ割り当て関数に NULL ヒープヒントが渡されたときに使用される現在のグローバルヒープヒントを取得します。 静的メモリ割り当て追跡用のデバッグコールバック関数を設定します。 WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACK ビルドオプションと共に使用されます。
wc_UnloadStaticMemory	
	静的メモリヒープと関連するミューテックスを解放します。静的メモリ割り当ての使用が終了したときに呼び出す必要があります。

API	説明
<code>wolfSSL_StaticBufferSz</code>	/wolfssl/wolfcrypt/memory.h で定義されたマクロに基づいて、静的バッファ確保に必要なバッファサイズを計算します。
<code>wolfSSL_StaticBufferSz_ex</code>	カスタムバケットサイズと分布を使用した静的メモリ割り当てに必要なバッファサイズを計算します。
<code>wolfSSL_MemoryPaddingSz</code>	アライメントパディングを含む、各メモリバケットに必要な管理メモリのサイズを計算します。

4.7 圧縮

wolfSSL は **zlib** ライブラリによるデータ圧縮をサポートしています。./configure ビルドシステムはこのライブラリの存在を検出しますが、他の方法でビルドしている場合は定数 `HAVE_LIBZ` を定義し、zlib.h へのパスをインクルードに含めてください。

圧縮は特定の暗号では標準でオフになっています。オンにするには、SSL 接続または受け入れの前に `wolfSSL_set_compression()` 関数をお使いください。圧縮が使用されるためには、クライアントとサーバーの両方で圧縮がオンになっている必要があります。

送信する前にデータを圧縮すると、送信されて受信されるメッセージの実際のサイズが減少します。しかし、圧縮によって保存されたデータは、著しく低速なネットワーク以外では、通常、生のデータを送信するよりも分析に時間がかかることに注意してください。

4.8 事前共有鍵

wolfSSL は、静的な事前共有鍵を持つ以下の暗号をサポートしています。

- 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

これらのスイートは `WOLFSSL_STATIC_PSK` がオンの状態で wolfSSL に組み込まれています。すべての PSK スイートはビルド時に定数 `NO_PSK` でオフにできます。実行時にこれらの暗号のみを使用するには、`wolfSSL_CTX_set_cipher_list()` 関数と目的の暗号スイートをお使いください。

wolfSSL は、エフェメラル鍵 PSK スイートもサポートしています。

- 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

クライアント側では、`wolfSSL_CTX_set_psk_client_callback()`関数を使用してコールバックを設定します。<wolfSSL_Home>/examples/client/client.c のクライアントサンプルプログラムでは、クライアント ID と鍵の設定の使用例を提供しています。ただし、実際のコールバックは `wolfssl/test.h` で実装されたものを使用しています。

サーバー側では、さらに 2 つの呼び出しが必要です。

- `wolfSSL_CTX_set_psk_server_callback()`
- `wolfSSL_CTX_use_psk_identity_hint()`

サーバーは 2 回目の呼び出しでクライアントを支援するために ID ヒントを保存します。サーバー例では「wolfssl server」です。サーバー PSK コールバックのサンプル実装も `wolfssl/test.h` の `my_psk_server_cb()` にあります。

wolfSSL は最大 128 オクテットの ID とヒント、および最大 64 オクテットの事前共有鍵をサポートしています。

4.9 クライアント認証

クライアント認証は、クライアントが接続する際にサーバーがクライアントへ証明書を送信するように要求し、クライアントを認証できるようにする機能です。クライアント認証には CA（または自己署名、あなたまたは CA 以外の誰かによって生成された場合）からの X.509 クライアント証明書が必要です。

デフォルトでは、wolfSSL は受け取るすべての証明書を検証します。これには、クライアントとサーバーの両方が含まれます。クライアント認証を設定するには、サーバーはクライアント証明書を検証するために使用される、信頼された CA 証明書のリストをロードする必要があります。

```
wolfSSL_CTX_load_verify_locations(ctx, caCert, 0);
```

クライアント検証をオンにしてその動作を制御するには、`wolfSSL_CTX_set_verify()`関数を使用します。次の例では、`SSL_VERIFY_PEER` はサーバーからクライアントへの証明書要求をオンにします。`SSL_VERIFY_FAIL_IF_NO_PEER_CERT` はクライアントがサーバー側で検証する証明書を提示しない場合に処理を失敗するようにサーバーに指示します。`wolfSSL_CTX_set_verify()`のその他のオプションには、`SSL_VERIFY_NONE` および `SSL_VERIFY_CLIENT_ONCE` があります。

```
wolfSSL_CTX_set_verify(ctx, SSL_VERIFY_PEER | (usePskPlus ?
                        SSL_VERIFY_FAIL_EXCEPT_PSK :
                        SSL_VERIFY_FAIL_IF_NO_PEER_CERT), 0);
```

クライアント認証の例は、wolfSSL リポジトリに含まれるサンプルサーバー（`server.c`）（`examples/server/server.c`）で提供しています。

4.10 Server Name Indication

SNI は、サーバーが単一の基盤となるネットワークアドレスで複数の「仮想」サーバーをホストする場合に役立ちます。クライアントが接続先のサーバーの名前を提供することが望ましい場合があります。wolfSSL で SNI を有効にするには、次のようにビルドします。

```
./configure --enable-sni
```

クライアント側で SNI を使用するには、追加の関数呼び出しが必要です。以下のいずれかの関数をお使いください。

- `wolfSSL_CTX_UseSNI()`
- `wolfSSL_UseSNI()`

`wolfSSL_CTX_UseSNI()` は、クライアントが同じサーバーに複数回接続する場合に適しています。コンテキストレベルで SNI 拡張を設定すると、呼び出しの瞬間から先に、同じコンテキストから作成されたすべての SSL オブジェクトで SNI の使用が有効になります。

`wolfSSL_UseSNI()` は、1 つの SSL オブジェクトに対してのみ SNI 使用を有効にします。セッション間でサーバー名が変わる場合には、この関数を使用することをお勧めします。

サーバー側では、同じ関数呼び出しのいずれかが必要です。wolfSSL サーバーは複数の仮想サーバーをホストしないため、SNI の使用は「SNI が一致しない場合に接続を終了させることが望ましい状況」で役立ちます。このシナリオでは、`wolfSSL_CTX_UseSNI()` が適しています。サーバーは同じコンテキストから SNI を持つすべての後続の SSL オブジェクトを作成して、コンテキストごとに一度だけ設定します。

4.11 ハンドシェイクの変更

4.11.1 ハンドシェイクメッセージのグループ化

wolfSSL は、ユーザーが望む場合にハンドシェイクメッセージをグループ化する機能を持っています。これはコンテキストレベルで `wolfSSL_CTX_set_group_messages(ctx);` で行うことができます。

4.12 短縮 HMAC

現在定義されている TLS 暗号スイートは、レコードレイヤー通信を認証するために HMAC を使用します。TLS では、ハッシュ関数の出力全体が MAC タグとして使用されます。ただし、リソースが制約された環境では、MAC タグを形成する際にハッシュ関数の出力を 80 ビットに切り詰めることで帯域幅を節約することが望ましい場合があります。wolfSSL で短縮 HMAC を有効にするには、次のようにします。

```
./configure --enable-truncatedhmac
```

クライアント側で短縮 HMAC を使用するには、追加の関数呼び出しが必要です。以下のいずれかの関数をお使いください。

- `wolfSSL_CTX_UseTruncatedHMAC()`
- `wolfSSL_UseTruncatedHMAC()`

`wolfSSL_CTX_UseTruncatedHMAC()` は、クライアントがすべてのセッションで短縮 HMAC を有効にしたい場合に適しています。コンテキストレベルで短縮 HMAC 拡張を設定すると、呼び出しの瞬間から先に、同じコンテキストから作成されたすべての SSL オブジェクトで有効になります。

`wolfSSL_UseTruncatedHMAC()` は 1 つの SSL オブジェクトに対してのみ有効にします。すべてのセッションで短縮 HMAC が必要ない場合に、この関数を使用することをお勧めします。

サーバー側での関数呼び出しは必要ありません。サーバーは短縮 HMAC のクライアントの要求に自動的に対応します。

すべての TLS 拡張機能も以下で有効にできます。

```
./configure --enable-tlsx
```

4.13 wolfSSL のタイミグ耐性

wolfSSL は、潜在的にリークタイミング情報を漏らす可能性のある比較操作を行うときに一定の時間を保証する関数 `ConstantCompare` を提供します。この API は、タイミグベースのサイドチャネル攻撃を阻止するために、wolfSSL の TLS レベルと暗号レベルの両方で使用されます。

wolfSSL の ECC 実装には、ECC アルゴリズムでタイミグ耐性を有効にするために、`ECC_TIMING_RESISTANT` を定義しています。同様に、fastmath ライブラリには RSA アルゴリズムのタイミグ耐性のために、`TFM_TIMING_RESISTANT` を定義しています。関数 `exptmod` はタイミグ耐性のある Montgomery ラダーを使用します。詳細は第 2 章 wolfSSL のビルドの `--disable-harden` をご覧ください。処理時間一定化とキャッシュ抵抗は、`--enable-harden` で有効になっています。

- 非推奨：WOLFSSL_SP_CACHE_RESISTANT：使用するアドレスをマスクするロジックを有効にします。デフォルトで常に有効です。デフォルトのキャッシュ耐性を無効にするには第 2 章 wolfSSL のビルドの `--disable-harden` をご覧ください。
- WC_RSA_BLINDING：タイミング攻撃を防ぐために、ブラインドモードを有効にします。
- ECC_TIMING_RESISTANT：ECC 固有の処理時間一定化。
- TFM_TIMING_RESISTANT：Fast Math 固有の処理時間一定化。

4.14 固定 ABI

wolfSSL は、アプリケーションプログラミングインターフェース (API) のサブセットに対して、固定アプリケーションバイナリインターフェース (ABI) を提供します。wolfSSL v4.3.0 以降、以下の関数は wolfSSL のすべての将来のリリースで互換性があります。

- `wolfSSL_Init()`
- `wolfTLSSv1_2_client_method()`
- `wolfTLSSv1_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()`

5 移植性

5.1 抽象化レイヤー

5.1.1 C 標準ライブラリ抽象化レイヤー

wolfSSL は C 標準ライブラリなしでビルドでき、開発者へより高いレベルの移植性と柔軟性を提供します。ユーザーは、C 標準の代わりに使用したい関数をマッピングする必要があります。

5.1.1.1 メモリ使用 ほとんどの C プログラムは動的メモリ割り当てに `malloc()` と `free()` を使用します。wolfSSL は、代わりに `XMALLOC()` と `XFREE()` を使用します。デフォルトでは、これらは C ランタイムバージョンを指します。`XMALLOC_USER` を定義することで、ユーザーは独自のフックを提供できます。各メモリ関数は標準的なものに加えて、ヒープヒントと割り当てタイプという 2 つの追加引数を取ります。ユーザーはこれらを無視するか、好きな方法で使用できます。wolfSSL のメモリ関数は `wolfssl/wolfcrypt/types.h` にあります。

wolfSSL はコンパイル時ではなく実行時に、メモリオーバーライド関数を登録する機能も提供します。`wolfssl/wolfcrypt/memory.h` はこの機能のヘッダーであり、ユーザーは以下の関数を呼び出してメモリ関数を設定できます。

```
int wolfSSL_SetAllocators(wolfSSL_Malloc_cb malloc_function,
                          wolfSSL_Free_cb free_function,
                          wolfSSL_Realloc_cb realloc_function);
```

詳しくは `wolfssl/wolfcrypt/memory.h` のコールバックプロトタイプと、`memory.c` の実装をご参照ください。

5.1.1.2 string.h wolfSSL は `string.h` の `memcpy()`、`memset()`、`memcmp()` などのように動作するいくつかの関数を使用します。これらはそれぞれ `XMEMCPY()`、`XMEMSET()`、`XMEMCMP()` として抽象化されています。デフォルトでは、それらは C 標準ライブラリのバージョンを指しています。`STRING_USER` を定義することで、ユーザーは `types.h` で独自のフックを提供できます。例えば、デフォルトでは `XMEMCPY()` は次の通りです。

```
#define XMEMCPY(d,s,l)    memcpy((d),(s),(l))
```

`STRING_USER` を定義した後、別途用意したバージョン `my_memcpy()` を指すように設定するには以下のようにします。

```
#define XMEMCPY(d,s,l)    my_memcpy((d),(s),(l))
```

マクロを避けたい場合、次のように示すこともできます。

```
external void* my_memcpy(void* d, const void* s, size_t n);
```

5.1.1.3 math.h wolfSSL は `math.h` の `pow()` と `log()` のように動作する 2 つの関数を使用します。これらは Diffie-Hellman でのみ必要とするため、ビルドから DH を除外すれば、独自のものを提供する必要はありません。これらは `XPOW()` と `XLOG()` として抽象化され、`wolfcrypt/src/dh.c` にあります。

5.1.1.4 ファイルシステムの使用 デフォルトでは、wolfSSL は鍵と証明書をロードするためにシステムのファイルシステムを使用します。これは `NO_FILESYSTEM` を定義することでオフにできます。代わりに、システムが提供するものとは別のファイルシステムを使用したい場合、`ssl.c` の `XFILE()` レイヤーを使用して、ファイルシステム呼び出しを使用したいものに向けることができます。詳細は MICRIUM の定義で提供される実装例をご参照ください。

5.1.2 カスタム入出力抽象化レイヤー

wolfSSL は、SSL 接続の I/O をより高度に制御したい、あるいは TCP/IP 以外の異なる転送媒体の上で SSL を実行したい場合のために、カスタム I/O 抽象化レイヤーを提供します。

ユーザーは 2 つの関数を定義する必要があります。

1. ネットワーク送信関数
2. ネットワーク受信関数

これら 2 つの関数は `ssl.h` の `CallbackIOSend` と `CallbackIORecv` でプロトタイプ化されています。

```
typedef int (*CallbackIORecv)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
typedef int (*CallbackIOSend)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
```

ユーザーは `WOLFSSL_CTX` ごとに、`wolfSSL_SetIOSend()` と `wolfSSL_SetIORecv()` を使ってこれらの関数を登録する必要があります。例えばデフォルトの場合、`CBIOSend()` と `CBIOSend()` は `io.c` の最後に登録されています。

```
void wolfSSL_SetIORecv(WOLFSSL_CTX *ctx, CallbackIORecv CBIOSend)
{
    ctx->CBIOSend = CBIOSend;
}
```

```
void wolfSSL_SetIOSend(WOLFSSL_CTX *ctx, CallbackIORecv CBIOSend)
{
    ctx->CBIOSend = CBIOSend;
}
```

ユーザーは `io.c` の最後で示されているように、`wolfSSL_SetIOWriteCtx()` と `wolfSSL_SetIOReadCtx()` を使用して、`WOLFSSL` オブジェクト (セッション) ごとにコンテキストを設定できます。例えば、ユーザーがメモリバッファを使用している場合、コンテキストはメモリバッファへのアクセス方法を説明する構造体へのポインタかもしれません。ユーザーによるオーバーライドがない場合、ソケットをコンテキストとして登録します。

`CBIOSend` と `CBIOSend` 関数ポインタは、カスタム I/O 関数を指すことができます。`io.c` にあるデフォルトの `Send()` および `Receive()` 関数である `EmbedSend()` と `EmbedReceive()` は、テンプレートやガイドとして使用できます。

`WOLFSSL_USER_IO` を定義すると、デフォルトの I/O 関数 `EmbedSend()` と `EmbedReceive()` の自動設定を削除できます。

5.1.3 オペレーティングシステム抽象化レイヤー

wolfSSL OS 抽象化レイヤーは、wolfSSL をユーザーのオペレーティングシステムに移植しやすくします。`wolfssl/wolfcrypt/settings.h` には、OS 抽象化レイヤーを起動する設定が含まれています。

OS 固有の定義は、`wolfCrypt` の場合 `wolfssl/wolfcrypt/types.h` に、`wolfSSL` の場合 `wolfssl/internal.h` にあります。

5.2 サポートしているオペレーティングシステム

wolfSSL を定義する 1 つの要因は、新しいプラットフォームに簡単に移植される能力です。そのため、wolfssl は、out-of-box のオペレーティングシステムの多くをサポートしています。現在サポートしているオペレーティングシステムは、次の通りです。

- 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/Gamecube (DevKitPro 経由)
- QNX
- MontaVista
- NonStop
- TRON/ITRON/ μ ITRON
- Micrium μ C/OS-III
- FreeRTOS
- SafeRTOS
- NXP/Freescale MQX
- Nucleus
- TinyOS
- HP/UX
- AIX
- ARC MQX
- TI-RTOS
- uTasker
- embOS
- INtime
- Mbed
- μ T-Kernel
- RIOT
- CMSIS-RTOS
- FROSTED
- Green Hills INTEGRITY
- Keil RTX
- TOPPERS
- PetaLinux
- Apache Mynewt

5.3 サポートしているチップメーカー

wolfSSL は ARM、Intel、Motorola、mbed、Freescale、Microchip (PIC32)、STMicro (STM32F2/F4)、NXP、Analog Devices、Texas Instruments、AMD などのチップセットをサポートしています。

5.4 C# ラッパー

wolfSSL は、制限付きですが C# での使用をサポートしています。ポートを含む Visual Studio プロジェクトは、`root_wolfSSL/wrapper/CSharp/`にあります。Visual Studio プロジェクトを開いた後、ビルド -> 構成マネージャーをクリックして「アクティブソリューション構成」と「アクティブソリューションプラ

ットフォーム」を設定します。サポートしているアクティブソリューション構成は DLL デバッグと DLL リリースです。サポートされているプラットフォームは Win32 および x64 です。

ソリューションとプラットフォームを設定したら、プリプロセッサフラグ HAVE_CSHARP を追加する必要があります。これにより、C# ラッパー及びサンプルプログラムで使用されるオプションがオンになります。

その後、ビルドするだけでソリューションをビルドを選択します。これにより wolfssl.dll、wolfSSL_CSharp.dll および例が作成されます。サンプルプログラムは、それらをエントリポイントとしてターゲットにして、Visual Studio でデバッグを実行することで実行できます。

作成された C# ラッパーを C# プロジェクトに追加するには、いくつかの方法があります。1 つは、作成された wolfssl.dll と wolfSSL_CSharp.dll をディレクトリ C:/Windows/System/にインストールすることです。

```
using wolfSSL.CSharp
```

```
public some_class {  
  
    public static main(){  
        wolfssl.Init()  
        ...  
    }  
    ...  
}
```

これにより、wolfSSL C# ラッパーへの呼び出しができるようになります。

もう 1 つの方法は、Visual Studio プロジェクトを作成し、wolfSSL にバンドルされている C# ラッパーソリューションを参照することです。

6 コールバック

6.1 ハンドシェイクコールバック

wolfSSL には、connect または accept のためのハンドシェイクコールバックを設定できる拡張機能があります。これは、別のデバッガーが利用できず、スニффイングが現実的でない場合に、組み込みシステムでのデバッグサポートに役立ちます。wolfSSL ハンドシェイクコールバックを使用するには、拡張関数 `wolfSSL_connect_ex()` と `wolfSSL_accept_ex()` を使用します。

```
int wolfSSL_connect_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                      Timeval)
int wolfSSL_accept_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                     Timeval)
```

HandShakeCallBack は次のように定義されています。

```
typedef int (*HandShakeCallBack)(HandShakeInfo*);
```

HandShakeInfo は `wolfssl/callbacks.h` (非標準ビルドに追加される必要があります) で定義されています。

```
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;
```

ハンドシェイク交換での最大 SSL パケット数は既知であるため、動的メモリは使用されません。パケット名は `packetNames[idx]` で `numberPackets` まで参照できます。コールバックは、ハンドシェイクエラーが発生したかどうかに関係なく呼び出されます。実装例はクライアントのサンプルプログラムに掲載しています。

6.2 タイムアウトコールバック

wolfSSL ハンドシェイクコールバックで使用する拡張機能は、wolfSSL タイムアウトコールバックにも使用できます。これらの拡張機能は、コールバック (ハンドシェイクやタイムアウト) のいずれか、両方、またはどちらも使用せずに使用できます。TimeoutCallback は、次のように定義されています。

```
typedef int (*TimeoutCallBack)(TimeoutInfo*);
```

TimeoutInfo は次のようになっています。

```
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;
```

ハンドシェイクに対する最大 SSL パケット数が分かっているため、ここでもこの構造体に動的メモリは使用されません。Timeval は struct timeval の typedef です。

PacketInfo は次のように定義されています。

```
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;

```

ここでは、動的メモリが使用される場合があります。SSL パケットが `value` に収まる場合は、そこに配置されます。`valueSz` は長さを保持し、`bufferValue` は 0 です。パケットが `value` にとって大きすぎる場合、パケットは `bufferValue` に配置されます。(Certificate パケットがこれを引き起こすことがあります。) `valueSz` はサイズを保持します。

Certificate パケットのためにメモリが割り当てられると、コールバックが戻った後に回収されます。タイムアウトはシグナル、特に SIGALRM を使用して実装され、スレッドセーフです。ITIMER_REAL タイプの以前のアラームが設定されている場合、その後、正しいハンドラとともにリセットされます。古いタイマーは、wolfSSL の処理にかかる時間に合わせて時間調整されます。既存のタイマーが渡されたタイマーよりも短い場合、既存のタイマー値が使用されます。それでも後でリセットされます。期限切れとなる既存のタイマーは、間隔が関連付けられている場合はリセットされます。コールバックはタイムアウトが発生した場合にのみ実行されます。

使用例については、第 3 章 入門の「クライアントサンプルプログラム」節をご参照ください。

6.3 ユーザーアトミックレコードレイヤー処理

wolfSSL は、SSL/TLS 接続中に MAC/暗号化/復号/検証機能をより詳細に制御したいユーザー向けに、アトミックレコード処理コールバックを提供します。

ユーザーは 2 つの関数を定義する必要があります。

1. MAC/暗号化コールバック関数
2. 復号/検証コールバック関数

これら 2 つの関数は `ssl.h` の `CallbackMacEncrypt` と `CallbackDecryptVerify` でプロトタイプ化されています。

```

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);

```

ユーザーはこれらの関数を作成し、`wolfSSL_CTX_SetMacEncryptCb()` と `wolfSSL_CTX_SetDecryptVerifyCb()` で wolfSSL コンテキスト (WOLFSSL_CTX) ごとに登録する必要があります。

ユーザーは `wolfSSL_SetMacEncryptCtx()` と `wolfSSL_SetDecryptVerifyCtx()` で WOLFSSL オブジェクト (セッション) ごとにコンテキストを設定できます。このコンテキストは、ユーザー指定のコンテキストへのポインタであり、`void* ctx` パラメータを通じて MAC/暗号化/復号/検証コールバックに返されます。

コールバックの実装例は、`wolfssl/test.h` の `myMacEncryptCb()` と `myDecryptVerifyCb()` に掲載しています。使用例は、wolfSSL のクライアントサンプルプログラム (`examples/client/client.c`) で -U コマンドラインオプションを使用することで見るができます。

アトミックレコードレイヤーコールバックを使用するには、wolfSSL を `--enable-atomicuser` 設定オプションを使用してコンパイルするか、`ATOMIC_USER` プリプロセッサフラグを定義する必要があります。

6.4 公開鍵コールバック

wolfSSL は、SSL/TLS 接続中に DH、ECC、Ed25519、X25519、Ed448、X448、および RSA 操作をより詳細に制御したいユーザー向けに公開鍵（PK）コールバックを提供します。

公開鍵コールバックを使用するには、wolfSSL を HAVE_PK_CALLBACKS を定義してコンパイルする必要があります。これは次の設定オプションを使用して行うことができます。

```
./configure --enable-pkcallbacks
```

さらに、各コールバックタイプには、特定の機能が有効になっている必要があります。

例：HAVE_DH、HAVE_ED25519

6.4.1 DH コールバック

wolfSSL は、SSL/TLS 接続中に DH 鍵生成と鍵合意操作をより詳細に制御したいユーザー向けに DH（Diffie-Hellman）コールバックを提供します。

ユーザーはオプションで 2 つの関数を定義できます。

1. DH 鍵生成コールバック
2. DH 鍵合意コールバック

これらの関数は ssl.h の CallbackDhGenerateKeyPair と CallbackDhAgree でプロトタイプ化されています。

```
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);
```

ユーザーはこれらの関数を作成し、wolfSSL コンテキスト（WOLFSSL_CTX）ごとに以下を使用して登録する必要があります。

- wolfSSL_CTX_SetDhGenerateKeyPair()
- wolfSSL_CTX_SetDhAgreeCb()

ユーザーは wolfSSL_SetDhAgreeCtx() を使用して WOLFSSL オブジェクト（セッション）ごとにコンテキストを設定できます。

コールバックの実装例は wolfssl/test.h の myDhCallback() で提供しています。使用例は wolfSSL のクライアントサンプルプログラム（examples/client/client.c）で見ることができます。

DH コールバックを使用するには、wolfSSL を HAVE_DH を定義してコンパイルする必要があります。

6.4.2 Ed25519 コールバック

wolfSSL は、SSL/TLS 接続中に Ed25519 署名/検証操作をより詳細に制御したいユーザー向けに Ed25519 コールバックを提供します。

ユーザーはオプションで 2 つの関数を定義できます。

1. Ed25519 署名コールバック
2. Ed25519 検証コールバック

これらの関数は `ssl.h` の `CallbackEd25519Sign` と `CallbackEd25519Verify` でプロトタイプ化されています。

```
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);
```

ユーザーはこれらの関数を作成し、wolfSSL コンテキスト (WOLFSSL_CTX) ごとに以下を使用して登録する必要があります。

- `wolfSSL_CTX_SetEd25519SignCb()`
- `wolfSSL_CTX_SetEd25519VerifyCb()`

ユーザーは以下を使用して WOLFSSL オブジェクト (セッション) ごとにコンテキストを設定できます。

- `wolfSSL_SetEd25519SignCtx()`
- `wolfSSL_SetEd25519VerifyCtx()`

コールバックの実装例は `wolfssl/test.h` の `myEd25519Sign()` と `myEd25519Verify()` で提供しています。使用例は wolfSSL のクライアントサンプルプログラム (`examples/client/client.c`) で見ることができます。

Ed25519 コールバックを使用するには、wolfSSL を `HAVE_PK_CALLBACKS` と `HAVE_ED25519` を定義してコンパイルする必要があります。

6.4.3 X25519 コールバック

wolfSSL は、SSL/TLS 接続中に X25519 鍵生成と共有秘密計算をより詳細に制御したいユーザー向けに X25519 コールバックを提供します。

ユーザーはオプションで 2 つの関数を定義できます。

1. X25519 鍵生成コールバック
2. X25519 共有秘密コールバック

これらの関数は `ssl.h` の `CallbackX25519KeyGen` と `CallbackX25519SharedSecret` でプロトタイプ化されています。

```
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);
```

ユーザーはこれらの関数を作成し、wolfSSL コンテキスト (WOLFSSL_CTX) ごとに以下を使用して登録する必要があります。

- `wolfSSL_CTX_SetX25519KeyGenCb()`
- `wolfSSL_CTX_SetX25519SharedSecretCb()`

ユーザーは以下を使用して WOLFSSL オブジェクト（セッション）ごとにコンテキストを設定できます。

- `wolfSSL_SetX25519KeyGenCtx()`
- `wolfSSL_SetX25519SharedSecretCtx()`

コールバックの実装例は `wolfssl/test.h` の `myX25519KeyGen()` と `myX25519SharedSecret()` で提供しています。使用例は wolfSSL のクライアントサンプルプログラム (`examples/client/client.c`) で見ることができます。

X25519 コールバックを使用するには、wolfSSL を `HAVE_PK_CALLBACKS` と `HAVE_CURVE25519` を定義してコンパイルする必要があります。

6.4.4 Ed448 コールバック

wolfSSL は、SSL/TLS 接続中に Ed448 署名/検証操作をより詳細に制御したいユーザー向けに Ed448 コールバックを提供します。

ユーザーはオプションで 2 つの関数を定義できます。

1. Ed448 署名コールバック
2. Ed448 検証コールバック

これらの関数は `ssl.h` の `CallbackEd448Sign` と `CallbackEd448Verify` でプロトタイプ化されています。

```
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);
```

ユーザーはこれらの関数を作成し、wolfSSL コンテキスト (`WOLFSSL_CTX`) ごとに以下を使用して登録する必要があります。

- `wolfSSL_CTX_SetEd448SignCb()`
- `wolfSSL_CTX_SetEd448VerifyCb()`

ユーザーは以下を使用して WOLFSSL オブジェクト（セッション）ごとにコンテキストを設定できます。

- `wolfSSL_SetEd448SignCtx()`
- `wolfSSL_SetEd448VerifyCtx()`

コールバックの実装例は `wolfssl/test.h` の `myEd448Sign()` と `myEd448Verify()` で提供しています。使用例は wolfSSL のクライアントサンプルプログラム (`examples/client/client.c`) で見ることができます。

Ed448 コールバックを使用するには、wolfSSL を `HAVE_PK_CALLBACKS` と `HAVE_ED448` を定義してコンパイルする必要があります。

6.4.5 X448 コールバック

wolfSSL は、SSL/TLS 接続中に X448 鍵生成と共有秘密操作をより詳細に制御したいユーザー向けに X448 コールバックを提供します。

ユーザーはオプションで 2 つの関数を定義できます。

1. X448 鍵生成コールバック
2. X448 共有秘密コールバック

これらの関数は `ssl.h` の `CallbackX448KeyGen` と `CallbackX448SharedSecret` でプロトタイプ化されています。

```
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);
```

ユーザーはこれらの関数を作成し、wolfSSL コンテキスト (WOLFSSL_CTX) ごとに以下を使用して登録する必要があります。

- `wolfSSL_CTX_SetX448KeyGenCb()`
- `wolfSSL_CTX_SetX448SharedSecretCb()`

ユーザーは以下を使用して WOLFSSL オブジェクト (セッション) ごとにコンテキストを設定できます。

- `wolfSSL_SetX448KeyGenCtx()`
- `wolfSSL_SetX448SharedSecretCtx()`

コールバックの実装例は `wolfssl/test.h` の `myX448KeyGen()` と `myX448SharedSecret()` で提供しています。使用例は wolfSSL のクライアントサンプルプログラム (`examples/client/client.c`) で見ることができます。

X448 コールバックを使用するには、wolfSSL を `HAVE_PK_CALLBACKS` と `HAVE_CURVE448` を定義してコンパイルする必要があります。

6.4.6 RSA PSS コールバック

wolfSSL は、SSL/TLS 接続中に RSA PSS 署名/検証操作をより詳細に制御したいユーザー向けに RSA PSS コールバックを提供します。

ユーザーはオプションで 4 つの関数を定義できます。

1. RSA PSS 署名コールバック
2. RSA PSS 検証コールバック
3. RSA PSS 署名チェックコールバック
4. RSA PSS 検証チェックコールバック

これらの関数は `ssl.h` の `CallbackRsaPssSign`、`CallbackRsaPssVerify`、および `CallbackRsaPssSignCheck` でプロトタイプ化されています。

```
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);
```

ユーザーはこれらの関数を作成し、wolfSSL コンテキスト (WOLFSSL_CTX) ごとに以下を使用して登録する必要があります。

- wolfSSL_CTX_SetRsaPssSignCb()
- wolfSSL_CTX_SetRsaPssVerifyCb()
- wolfSSL_CTX_SetRsaSignCheckCb()
- wolfSSL_CTX_SetRsaPssSignCheckCb()

ユーザーは以下を使用して WOLFSSL オブジェクト (セッション) ごとにコンテキストを設定できます。

- wolfSSL_SetRsaPssSignCtx()
- wolfSSL_SetRsaPssVerifyCtx()
- wolfSSL_SetRsaSignCheckCtx()
- wolfSSL_SetRsaPssSignCheckCtx()

コールバックの例は wolfssl/test.h の myRsaPssSign()、myRsaPssVerify()、および myRsaPssSignCheck() にあります。使用例は wolfSSL 例のクライアントで見ることができます。

コールバックの実装例は wolfssl/test.h の myRsaPssSign()、myRsaPssVerify()、myRsaPssSignCheck() で提供しています。使用例は wolfSSL のクライアントサンプルプログラム (examples/client/client.c) で見ることができます。

RSA PSS コールバックを使用するには、wolfSSL を HAVE_PK_CALLBACKS と WC_RSA_PSS を定義してコンパイルする必要があります。

6.4.7 ECC コールバック

ユーザーはオプションで 7 つの関数を定義できます。

1. ECC 署名コールバック
2. ECC 検証コールバック
3. ECC 共有秘密コールバック
4. RSA 署名コールバック
5. RSA 検証コールバック
6. RSA 暗号化コールバック
7. RSA 復号コールバック

これらの関数は ssl.h の CallbackEccSign、CallbackEccVerify、CallbackEccSharedSecret、CallbackRsaSign、CallbackRsaVerify、CallbackRsaEnc、および CallbackRsaDec でプロトタイプ化されています。

```
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);

```

ユーザーはこれらの関数を作成し、wolfSSL コンテキスト (WOLFSSL_CTX) ごとに以下を使用して登録する必要があります。

- wolfSSL_CTX_SetEccSignCb()
- wolfSSL_CTX_SetEccVerifyCb()
- wolfSSL_CTX_SetEccSharedSecretCb()
- wolfSSL_CTX_SetRsaSignCb()
- wolfSSL_CTX_SetRsaVerifyCb()
- wolfSSL_CTX_SetRsaEncCb()
- wolfSSL_CTX_SetRsaDecCb()

ユーザーは WOLFSSL オブジェクト (セッション) ごとに以下でコンテキストを設定できます。

- wolfSSL_SetEccSignCtx()
- wolfSSL_SetEccVerifyCtx()
- wolfSSL_SetEccSharedSecretCtx()
- wolfSSL_SetRsaSignCtx()
- wolfSSL_SetRsaVerifyCtx()
- wolfSSL_SetRsaEncCtx()
- wolfSSL_SetRsaDecCtx()

これらのコンテキストは、ユーザー指定のコンテキストへのポインタであり、それぞれの公開鍵コールバックに `void* ctx` パラメータを通じて返されます。

コールバックの実装例は `wolfssl/test.h` の `myEccSign()`、`myEccVerify()`、`myEccSharedSecret()`、`myRsaSign()`、`myRsaVerify()`、`myRsaEnc()`、および `myRsaDec()` で提供しています。使用例は、wolfSSL のクライアントサンプルプログラム (`examples/client/client.c`) で `-P` コマンドラインオプションを使用することで見るすることができます。

アトミックレコードレイヤーコールバックを使用するには、wolfSSL を **--enable-pkcallbacks** 設定オプションを使用してコンパイルするか、HAVE_PK_CALLBACKS プリプロセッサフラグを定義する必要があります。

6.5 暗号コールバック (cryptocb)

wolfSSL/wolfCrypt の暗号コールバックフレームワークは、ユーザーが特定の暗号アルゴリズムのデフォルト実装をオーバーライドし、実行時に独自のカスタム実装を提供できるようにします。暗号コールバックの最も一般的なユースケースは、アルゴリズムをカスタムハードウェアアクセラレーションにオフロードすることです。この他にも、追加のロギング/内部検査の追加、セキュアストレージからの鍵の取得、標準準拠の理由で別のライブラリから暗号を呼び出す、あるいはリモートプロシージャコールの実行にも使用できます。

6.5.1 暗号コールバックの使用方法

暗号コールバックを使用するには、次のようにします。

1. 暗号コールバックサポート付きで wolfSSL をコンパイルする
2. コールバックと一意のデバイス ID (devId) を wolfCrypt に登録する
3. devId を wolfCrypt API 関数の引数として渡す
4. (TLS のみ) devId を wolfSSL コンテキストに関連付ける

6.5.1.1 1. 暗号コールバックサポート付きで wolfSSL をコンパイルする 暗号コールバックのサポートは、**--enable-cryptocb** 設定オプション、または **#define WOLF_CRYPTO_CB** を使用して有効にできます。

6.5.1.2 2. コールバックと一意の devID を wolfCrypt に登録する 暗号コールバックを使用するには、ユーザーはトップレベルの暗号コールバック関数を wolfCrypt に登録し、このコールバック関数を wolfCrypt に一意に識別するデバイス ID (devId) を提供する必要があります。これにより、複数の暗号コールバックを一度に wolfCrypt に登録できます。適切なコールバック関数は、devId パラメータを取る全ての wolfCrypt 関数内部で、または WOLFSSL_CTX または WOLFSSL 構造体に関連付けられた場合に、wolfSSL によって呼び出されます。

暗号コールバック関数を wolfCrypt に登録するには、wc_CryptoCb_RegisterDevice API を使用します。これは (devId)、コールバック関数、および呼び出し時にコールバックに渡される任意のユーザーコンテキストを取ります。

```
typedef int (*CryptoDevCallbackFunc)(int devId, wc_CryptoInfo* info, void*
    ↪ ctx);
```

```
WOLFSSL_API int wc_CryptoCb_RegisterDevice(int devId,
                                           CryptoDevCallbackFunc cb,
                                           void* ctx);
```

注：登録された devId は、特別な INVALID_DEVICE 列挙値のために予約されている -2 以外の任意の値にできます。devId == INVALID_DEVICE を wolfCrypt API の引数として渡すと、コールバックが呼び出されず、代わりにデフォルトの内部実装を使用することを示します。

6.5.1.3 3. devId を wolfCrypt API 関数の引数として渡す wolfCrypt API の場合、devId を受け入れる初期化関数を使用します。

```
wc_InitRsaKey_ex
wc_ecc_init_ex
wc_AesInit
wc_InitSha256_ex
```

```

wc_InitSha_ex
wc_HmacInit
wc_InitCmac_ex

```

これにより、wolfCrypt はデフォルト実装の代わりに暗号コールバックを呼び出します。これは使用可能な API のすべてを示しているわけではありません。特定のアルゴリズムが暗号コールバックをサポートしているかどうかについては、wolfCrypt API ドキュメントをご参照ください。

6.5.1.4 4. (TLS のみ) devId を wolfSSL コンテキストに関連付ける TLS を使用する際に暗号コールバックの使用を有効にするには、WOLFSSL_CTX または WOLFSSL 構造体の初期化時に devId 引数を渡す必要があります。

```

wolfSSL_CTX_SetDevId(ctx, devId);
wolfSSL_SetDevId(ssl, devId);

```

これらの構造体に関連付けられた TLS 接続に関連する暗号処理は、暗号コールバックを呼び出します。

6.5.2 暗号コールバックの作成

暗号コールバックが呼び出されると、どの暗号操作が要求されているか、および入力データと出力データの場所と成功/失敗を wolfCrypt に伝える方法を知る必要があります。この情報は、wc_CryptoInfo* info 引数を通じてコールバックに中継されます。

高レベルでは、暗号コールバックは要求された操作をサポートしているかどうかを判断し、入力データに対して処理を実行し、関連する出力データで wc_CryptoInfo *info 引数を更新し、有効な wolfCrypt エラーコードを返す必要があります。

6.5.2.1 リクエストのタイプの判断 wc_CryptoInfo 構造体には、異なる構造体の共用体（一部は共用体自体）が含まれており、それぞれが特定のタイプの暗号操作に対応しています。実行される操作のタイプは info->algo_type によって決定され、これは wolfssl/wolfcrypt/types.h で定義されている enum wc_AlgoType のバリエーションである必要があります。この algo_type に基づいて、wc_CryptoInfo 内の適切な共用体要素にアクセスして、操作に固有のパラメータを取得または設定できます。このトップレベルの共用体を「アルゴタイプ共用体」と呼ぶことができます。

暗号リクエストのタイプを判断し、それに応じて処理するには、通常、wc_CryptoInfo 構造体の algo_type フィールドに対して switch-case 文を使用します。switch の各 case は、対称暗号、ハッシュ化、公開鍵操作などの異なる暗号操作に対応します。wc_AlgoType と対応するアルゴタイプ共用体の間には一対一のマッピングがあり、それらは wolfssl/wolfcrypt/cryptocb.h の wc_CryptoInfo 構造体定義にコメントされています。

暗号コールバックは成功時にゼロを返すか、失敗時に有効な wolfCrypt エラーコードを返す必要があります。サポートされていないアルゴリズムの場合、コールバックは CRYPTOCB_UNAVAILABLE を返す必要があります。これにより wolfCrypt は内部実装にフォールバックします。

これを説明するための、簡略化した実装例を以下に示します。

```

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
}

```

いくつかの単純なアルゴタイプ共用体（RNG やシード共用体など）はさらなる処理を必要とせず、すぐに実行できます。ただし、暗号や公開鍵タイプなどのより複雑な操作には、同じ方法で条件付きで解釈されなければならない複数の共用体バリエーションが自分自身に存在します。各共用体の変種と階層レベルは、アルゴリズムタイプの各カテゴリに固有であり、その完全な定義は `wc_CryptoInfo` 構造体の定義で見つけることができます。一般的なルールとして、共用体である各レベルには、共用体の残りの部分をどのように参照解除すべきかを知らせる何らかのタイプ指示子が含まれています。

以下は、複数のレベルの共用体デコーディングを持つ複雑なアルゴタイプ共用体の、簡略化した実装例です。コールバックには乱数生成のサポートと、ECC 鍵合意、署名、検証のサポートが含まれています。

```

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 リクエストの処理

各タイプのリクエストのデータ構造には、一般に入力データと出力データのポインタと関連するサイズが含まれます。リクエストによっては、暗号鍵、ノンス、またはさらなる設定を含む追加データが含まれる場合があります。暗号コールバックは入力データに対して操作し、関連する出力データをアルゴタイプ共用体の適切なバリエーションに書き戻す必要があります。問題のアルゴリズムタイプに対応しない共用体バリエーション（例えば、`info->algo_type == WC_ALGO_TYPE_RNG` の場合に `info->cipher` を使用する）にデータを書き込むことはメモリエラーであり、未定義の動作を引き起こす可能性があります。

6.5.4 トラブルシューティング

一部の古いコンパイラは、wolfCrypt がスペースを節約するために「匿名インライン集約体」を使用して入れ子になった `wcCryptoInfo` 匿名共用体を定義する際に使用する「匿名インライン集約体」を許可していません。匿名インライン集約体の使用を無効にするには、プリプロセッサマクロ `HAVE_ANONYMOUS_INLINE_AGGREGATES` を `0` として定義します。これにより暗号コールバックを使用できるようになりますが、`wcCryptoInfo` 構造体のサイズが大幅に増加します。

6.5.5 実装例

暗号コールバックの完全な実装例は以下のページで見つけることができます。

- [VaultIC Crypto Callbacks](#)
- [STSAFE-A100 ECC Crypto Callbacks](#)
- [TPM 2.0 wolfTPM Crypto Callbacks](#)
- [Generic wolfCrypt tests](#)

7 鍵と証明書

X.509 証明書の紹介、および SSL と TLS での使用方法については付録 A をご参照ください。

7.1 サポートしているフォーマットとサイズ

wolfSSL は証明書と鍵において **PEM** および **DER** フォーマットをサポートしています。また、PKCS#8 秘密鍵（PKCS#5 または PKCS#12 暗号化付き）もサポートしています。

PEM、つまり「Privacy Enhanced Mail」は、認証局によって証明書が発行される最も一般的なフォーマットです。PEM ファイルは Base64 エンコードされた ASCII ファイルで、複数のサーバー証明書、中間証明書、秘密鍵を含むことができ、通常は .pem、.crt、.cer、または .key のファイル拡張子を持ちます。PEM ファイル内の証明書は「-----BEGIN CERTIFICATE-----」と「-----END CERTIFICATE-----」のステートメントで囲まれています。

DER、つまり「Distinguished Encoding Rules」は、証明書のバイナリフォーマットです。DER ファイル拡張子には .der と .cer があり、テキストエディタでは表示できません。

X.509 証明書は ASN.1 フォーマットを使用してエンコードされます。DER フォーマットは ASN.1 エンコーディングです。PEM フォーマットは Base64 エンコードされ、人間が読めるヘッダーとフッターで囲まれています。TLS は証明書を DER フォーマットで送信します。

7.2 サポートしている証明書拡張

サポートしていない、または未知の拡張機能がクリティカルとしてマークされている場合、エラーメッセージを出力します。それ以外の場合では、これらの拡張機能は無視されます。証明書拡張機能の解析では、少なくとも `--enable-certtext`（マクロ `WOLFSSL_CERT_EXT`）が wolfSSL ライブラリのコンパイル時に使用されていることが期待されます。以下は、一部またはすべての拡張機能が使用でき、解析に対応している証明書拡張機能の一覧です。

RFC 5280からの拡張機能	サポート
Authority Key Identifier	はい
Subject Key Identifier	はい
Key Usage	はい
Certificate Policies	はい
Policy Mappings	いいえ
Subject Alternative Name	はい
Issuer Alternative Name	いいえ
Subject Directory Attributes	いいえ
Basic Constraints	はい
Name Constraints	はい
Policy Constraints	はい
Extended Key Usage	はい
CRL Distribution Points	はい
Inhibit anyPolicy	はい
Freshest CRL	いいえ

追加の拡張機能	サポート
Netscape	はい
Custom OID	はい

次のいくつかのセクションでは、個々の証明書拡張機能のサポート状況について詳しくご説明します。

7.2.0.1 Authority / Subject Key Identifier デフォルトでは、鍵 ID は鍵の SHA1 ハッシュです。鍵の SHA256 ハッシュもサポートしています。

7.2.0.2 Subject Alternative Names

Alternative Name Types	サポート
メール	はい
DNS 名	はい
IP アドレス	はい
URI	はい

7.2.0.3 Key Usage Key Usage は、証明書の解析完了後に取得・解析できます。

Key Usage	サポート
digitalSignature	はい
nonRepudiation	はい
keyEncipherment	はい
dataEncipherment	はい
keyAgreement	はい
keyCertSign	はい
cRLSign	はい
encipherOnly	はい
decipherOnly	はい

7.2.0.4 Extended Key Usage wolfSSL バージョン 3.15.5 以降では、不明/サポートしていない Extended Key Usage は無視されます。3.15.5 より前のバージョンでは、不明な Extended Key Usage OID は解析エラーを出力します。

Extended Key Usage	サポート
anyExtendedKeyUsage	はい
id-kp-serverAuth	はい
id-kp-clientAuth	はい
id-kp-codeSigning	はい
id-kp-emailProtection	はい
id-kp-timeStamping	はい
id-kp-OCSPSigning	はい

7.2.0.5 カスタム OID 拡張 カスタム OID X.509 拡張機能の挿入と解析は wolfSSL バージョン 5.3.0 で導入されました。有効化オプション `--enable-certgen` は、マクロ `WOLFSSL_ASN_TEMPLATE`、`WOLFSSL_CUSTOM_OID`、および `HAVE_OID_ENCODING` と一緒に使用する必要があります。`WOLFSSL_ASN_TEMPLATE` は `./configure` を使用する場合はデフォルトで定義されますが、`WOLFSSL_USER_SETTINGS` でビルドする場合は手動で定義する必要があります。

これらの設定で wolfSSL をビルドした後、`wc_SetCustomExtension()` 関数を使用して `Cert` 構造体にカスタム拡張機能を設定し、`wc_SetUnknownExtCallback()` を使用して `DecodedCert` 構造体内の未知の拡張機能 OID を処理するためのコールバックを登録できます。

7.3 証明書の読み込み

証明書は通常ファイルシステムを使用して読み込まれますが、メモリバッファからの読み込みもサポートしています。詳しくは、本章下部の「ファイルシステムなしでの証明書の使用」をご参照ください。

7.3.1 CA 証明書の読み込み

CA 証明書ファイルは`wolfSSL_CTX_load_verify_locations()`関数を使用して読み込むことができます。

```
int wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX *ctx,
                                     const char *CAfile,
                                     const char *CApath);
```

複数の CA 証明書情報を CAfile(PEM フォーマット)にまとめて記載しておくことで、それらを一括して解析することもできます。これは特に、クライアントが起動時に複数のルート CA を読み込む必要がある場合に便利です。このように処理することで、CA として単一のファイルを使用することを想定しているツールに移植しやすくしています。

注意：ルート証明書と中間証明書のチェーンを読み込む必要がある場合、信頼の順にそれらを読み込む必要があります。まずルート CA を読み込み、次に中間 1、次に中間 2 などを読み込みます。読み込む各証明書に対して`wolfSSL_CTX_load_verify_locations()`を呼び出すか、順序付けられた証明書（ファイルの先頭にルートがあり、信頼のチェーンによって順序付けられた証明書）を含むファイルで一度だけこの関数を実行します。

7.3.2 クライアントまたはサーバー証明書の読み込み

単一のクライアントまたはサーバー証明書の読み込みは`wolfSSL_CTX_use_certificate_file()`関数で行うことができます。この関数を証明書チェーンで使用する場合、実際の「最下位」の証明書のみが送信されます。

```
int wolfSSL_CTX_use_certificate_file(WOLFSSL_CTX *ctx,
                                     const char *CAfile,
                                     int type);
```

CAfile は CA 証明書ファイルであり、type は証明書のフォーマット（SSL_FILETYPE_PEM など）です。

サーバーとクライアントは`wolfSSL_CTX_use_certificate_chain_file()`関数を使用して証明書チェーンを送信できます。証明書チェーンファイルは PEM フォーマットである必要があり、サブジェクトの証明書（実際のクライアントまたはサーバー証明書）から始まり、中間証明書が続き、オプションで最上位のルート CA 証明書で終わるように並べる必要があります。サーバーのサンプルプログラム（/examples/server/server.c）では、この機能を使用しています。

注意：これは、検証のために証明書チェーンを読み込む場合とは正反対です。このシナリオでのファイル内容は、ファイルの先頭にエンティティ証明書があり、その次にチェーンの次の証明書が続き、ファイルの最後にルート CA がある形になります。

```
int wolfSSL_CTX_use_certificate_chain_file(WOLFSSL_CTX *ctx,
                                           const char *file);
```

7.3.3 秘密鍵の読み込み

サーバー秘密鍵は`wolfSSL_CTX_use_PrivateKey_file()`関数を使用して読み込むことができます。

```
int wolfSSL_CTX_use_PrivateKey_file(WOLFSSL_CTX *ctx,
                                     const char *keyFile, int type);
```

keyFile は秘密鍵ファイルであり、type は秘密鍵のフォーマット（例：SSL_FILETYPE_PEM）です。

7.3.4 信頼されるピア証明書の読み込み

信頼されるピア証明書の読み込みは`wolfSSL_CTX_trust_peer_cert()`で行うことができます。

```
int wolfSSL_CTX_trust_peer_cert(WOLFSSL_CTX *ctx,  
                                const char *trustCert, int type);
```

`trustCert` は読み込む証明書ファイルであり、`type` は秘密鍵のフォーマット(つまり `SSL_FILETYPE_PEM`)です。

7.4 証明書チェーンの検証

wolfSSL では、証明書チェーン全体を検証するために、チェーン内のトップまたは「ルート」証明書のみを信頼された証明書として読み込む必要があります。つまり、証明書チェーン (A -> B -> C) があり、C が B によって署名され、B が A によって署名されている場合、wolfSSL はチェーン全体 (A->B->C) を検証するために証明書 A のみを信頼された証明書として読み込む必要があります。

例えば、サーバー証明書チェーンが次のようになっているとき、

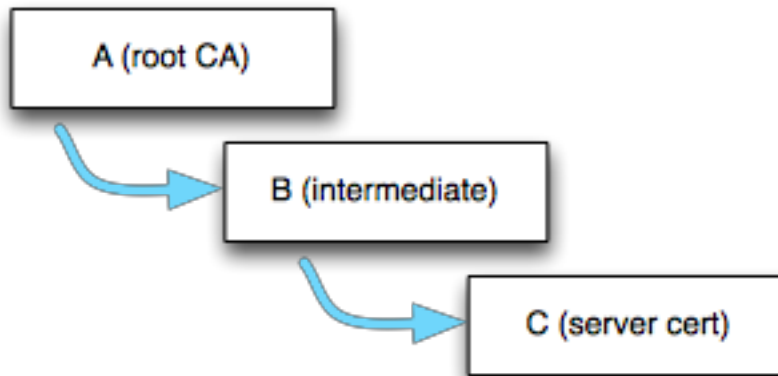


Figure 4: 証明書チェーン

wolfSSL クライアントは、少なくともルート証明書 (A) を信頼されたルートとして (`wolfSSL_CTX_load_verify_locations()`) あらかじめ読み込んでおく必要があります。クライアントがサーバー証明書チェーンを受信すると、A の署名を使用して B を検証し、B が以前に wolfSSL に信頼されたルートとして読み込まれていない場合、B は wolfSSL の内部証明書チェーンに保存されます (wolfSSL は証明書を検証するために必要なもの、つまり共通名ハッシュ、公開鍵、鍵タイプなどのみを保存します)。B が有効であれば、それを使用して C を検証します。

このモデルに従えば、ルート証明書「A」が wolfSSL サーバーに信頼されたルートとして読み込まれている限り、サーバーが (A->B->C) または (B->C) を送信した場合でも、サーバー証明書チェーンは検証できます。サーバーが中間証明書ではなく (C) のみを送信する場合、wolfSSL クライアントがすでに B を信頼されたルートとして読み込んでいない限り、チェーンは検証できません。

7.5 サーバー証明書のドメイン名チェック

wolfSSL には、サーバー証明書のドメインを自動的にチェックするクライアント拡張機能があります。OpenSSL モードでは、これを実行するために約十個の関数呼び出しが必要です。wolfSSL は証明書の日付が範囲内であることをチェックし、署名を検証し、さらに`wolfSSL_connect()`を呼び出した場合にドメインを検証します。wolfSSL はピアのサーバー証明書の X.509 発行者名を `dn` (期待されるドメイン名) と照合します。名前が一致すれば`wolfSSL_connect()`は `DOMAIN_NAME_MISMATCH` を返します。

証明書のドメイン名をチェックすることは、サーバーが実際に名乗っているものと一致していることを検証する重要なステップです。この拡張機能は、チェックを実行する負担を軽減するためのものです。

7.6 ファイルシステムなしでの証明書の使用

通常、秘密鍵、証明書、CA を読み込むためにファイルシステムを使用します。wolfSSL は完全なファイルシステムのない環境で使用されることがあるため、代わりにメモリバッファを使用するため拡張機能も提供しています。マクロ `NO_FILESYSTEM` を定義することで、以下の関数を使用できるようになります。

- `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);`

これらの関数は、*_file の代わりに *_buffer という名前の対応する関数と全く同じように使用します。そしてファイル名を提供する代わりにメモリバッファを提供します。使用法の詳細については API ドキュメントをご参照ください。

7.6.1 テスト証明書と鍵バッファ

wolfSSL は過去にテスト証明書と鍵をファイル形式でのみ同梱してきました。現在は、ファイルシステムが利用できない環境で使用するためのテスト証明書と鍵バッファも同梱しています。これらのバッファは `USE_CERT_BUFFERS_1024`、`USE_CERT_BUFFERS_2048`、または `USE_CERT_BUFFERS_256` のいずれかを定義すると `certs_test.h` で利用できます。

7.7 シリアル番号の取得

X.509 証明書のシリアル番号は `wolfSSL_X509_get_serial_number()` を使用して wolfSSL から抽出できます。シリアル番号は任意の長さになる可能性があります。

```
int wolfSSL_X509_get_serial_number(WOLFSSL_X509* x509,
    unsigned char* buffer, int* inOutSz)
```

buffer には入力時に最大 *inOutSz バイトが書き込まれます。成功した場合、0 が返され、呼び出し後に *inOutSz には buffer に実際に書き込まれたバイト数が保持されます。完全なサンプルプログラムは `wolfssl/test.h` で提供しています。

7.8 RSA 鍵生成

wolfSSL は、最大 4096 ビットのさまざまな長さの RSA 鍵生成をサポートしています。鍵生成はデフォルトではオフですが、`./configure` プロセスに `--enable-keygen` 引数を加えることで (Windows や非標準環境では、`WOLFSSL_KEY_GEN` を定義することで) 有効化できます。鍵の作成は簡単で、`rsa.h` から 1 つの関数のみを使用します。

```
int MakeRsaKey(RsaKey* key, int size, long e, RNG* rng);
```

ここで size はビット単位の長さ、e は公開指数で、通常 e には 65537 を使用するのが良いでしょう。wolfcrypt/test/test.c にて、1024 ビットの RSA 鍵を作成する実装例を提供しています。

```

RsaKey genKey;
RNG      rng;
int      ret;

InitRng(&rng);
InitRsaKey(&genKey, 0);

ret = MakeRsaKey(&genKey, 1024, 65537, &rng);
if (ret != 0)
    /* ret contains error */;

```

RsaKey genKey は、他の RsaKey と同じように使用できます。鍵をエクスポートする必要がある場合、wolfSSL は asn.h で DER と PEM の両方のフォーマットを提供しています。いずれの場合も、まずは鍵を DER フォーマットに変換し、PEM が必要であれば汎用の DerToPem() 関数を次のように使用します。

```

byte der[4096];
int  derSz = RsaKeyToDer(&genKey, der, sizeof(der));
if (derSz < 0)
    /* derSz contains error */;

```

バッファ der には現在、鍵の DER フォーマットが含まれています。DER バッファを PEM に変換するには、変換関数を使用します。

```

byte pem[4096];
int  pemSz = DerToPem(der, derSz, pem, sizeof(pem),
                      PRIVATEKEY_TYPE);
if (pemSz < 0)
    /* pemSz contains error */;

```

DerToPem() の最後の引数はタイプパラメータを取り、通常は PRIVATEKEY_TYPE または CERT_TYPE のいずれかです。現在バッファ pem には鍵の PEM フォーマットが含まれています。サポートしているタイプは、次の通りです。

- 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 鍵生成に関する注意

RSA 秘密鍵には公開鍵も含まれています。test.c で使用されているように、秘密鍵は wolfSSL によって秘密鍵と公開鍵の両方として使用できます。通常、SSL/TLS に必要なのは秘密鍵と（証明書の形式の）公開

鍵のみです。

必要に応じて、`RsaPublicKeyDecode()` 関数を使用して別の公開鍵を wolfSSL に手動で読み込むこともできます。そして、`wc_RsaKeyToPublicDer()` 関数を使用して RSA 公開鍵をエクスポートすることもできます。

7.9 証明書生成

wolfSSL は X.509 v3 証明書生成をサポートしています。証明書生成はデフォルトではオフですが、`./configure` プロセスに `--enable-certgen` 引数を加えることで (Windows や非標準環境では、`WOLFSSL_CERT_GEN` を定義することで) 有効化できます。

証明書を生成する前に、ユーザーは証明書のサブジェクト情報を入力する必要があります。この情報は `wolfssl/wolfcrypt/asn_public.h` にある `Cert` という構造体に含まれています。

```
/* 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;
```

`CertName` は次のようになります。

```
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;
```

サブジェクト情報を入力する前に、次のように初期化関数を呼び出す必要があります。

```
Cert myCert;
InitCert(&myCert);
```

`InitCert()` は、いくつかの変数にデフォルト値を設定します。バージョンを **3** (0x02)、シリアル番号を **0** (ランダム生成)、`sigType` を `CTC_SHA_RSA`、`daysValid` を **500**、`selfSigned` を **1** (TRUE) に設定します。

サポートしている署名タイプは次の通りです。

- `CTC_SHA_WDSA`

- CTC_MD2wRSA
- CTC_MD5wRSA
- CTC_SHAwRSA
- CTC_SHAwECDSA
- CTC_SHA256wRSA
- CTC_SHA256wECDSA
- CTC_SHA384wRSA
- CTC_SHA384wECDSA
- CTC_SHA512wRSA
- CTC_SHA512wECDSA

次に、以下のようにしてサブジェクト情報を初期化します。ただし実際には、ユーザの環境に応じた値を入力します。詳細は `wolfcrypt/test/test.c` にも掲載しています。

```
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);
```

そして、上記の鍵生成例からの変数 `genKey` と `rng` を使用して、自己署名証明書を生成できます。もちろん、有効なものであれば wolfSSL によって生成したものでなくとも構いません。

```
byte derCert[4096];
```

```
int certSz = MakeSelfCert(&myCert, derCert, sizeof(derCert), &key, &rng);
if (certSz < 0)
    /* certSz contains the error */;
```

バッファ `derCert` には DER フォーマットの証明書が含まれています。証明書の PEM フォーマットが必要な場合は、汎用の `DerToPem()` 関数を使用して、タイプを `CERT_TYPE` として指定します。

```
byte* pem;
```

```
int pemSz = DerToPem(derCert, certSz, pem, sizeof(pemCert), CERT_TYPE);
if (pemCertSz < 0)
    /* pemCertSz contains error */;
```

サポートしているタイプは以下の通りです。

- 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

これで、バッファ pemCert には証明書の PEM フォーマットが含まれています。

CA 署名付き証明書を作成したい場合は、いくつかの手順が必要です。前述のようにサブジェクト情報を入力した後、CA 証明書から発行者情報を設定する必要があります。これは SetIssuer() を使用して次のように実行できます。

```
ret = SetIssuer(&myCert, "ca-cert.pem");
if (ret < 0)
    /* ret contains error */;
```

その後、証明書を作成し、それに署名する 2 段階のプロセスを実行する必要があります。(MakeSelfCert() は一度にこれら両方を実行します) 発行者 (caKey) と対象 (key) の両方の秘密鍵が必要です。完全な使用方法については、test.c の実装例をご参照ください。

```
byte derCert[4096];

int certSz = MakeCert(&myCert, derCert, sizeof(derCert), &key, NULL, &rng);
if (certSz < 0);
    /*certSz contains the error*/;

certSz = SignCert(myCert.bodySz, myCert.sigType, derCert,
    sizeof(derCert), &caKey, NULL, &rng);
if (certSz < 0);
    /*certSz contains the error*/;
```

これで、バッファ derCert には CA 署名付き証明書の DER フォーマットが含まれています。証明書の PEM フォーマットが必要な場合は、上記の自己署名の実装例をご参照ください。MakeCert() と SignCert() は、RSA 鍵または ECC 鍵のいずれかを使用するための関数パラメータを提供していることに注意してください。上記の実装例では RSA 鍵を使用し、ECC 鍵パラメータには NULL を渡しています。

7.10 証明書署名要求 (CSR) の生成

wolfSSL は X.509 v3 証明書署名要求 (CSR) 生成をサポートしています。CSR 生成はデフォルトではオフですが、./configure プロセスに --enable-certreq --enable-certgen 引数を加えることで (Windows や非標準環境では、WOLFSSL_CERT_GEN および WOLFSSL_CERT_REQ を定義することで) 有効化できます。

CSR を生成する前に、ユーザーは証明書のサブジェクトに関する情報を提供する必要があります。この情報は wolfssl/wolfcrypt/asn_public.h にある Cert という構造体に含まれています。

Cert と CertName 構造体の詳細については本章の「証明書生成」節をご参照ください。

サブジェクトの情報を入力する前に、次のように初期化関数を呼び出す必要があります。

```
Cert request;
InitCert(&request);
```

InitCert() はいくつかの変数にデフォルト値を設定します。バージョンを 3 (0x02)、シリアル番号を 0 (ランダム生成)、sigType を CTC_SHAwRSA、daysValid を 500、selfSigned を 1 (TRUE) に設定します。

サポートしている署名タイプは次のとおりです。

- CTC_SHAwDSA
- CTC_MD2wRSA
- CTC_MD5wRSA
- CTC_SHAwRSA

- CTC_SHAwECDSA
- CTC_SHA256wRSA
- CTC_SHA256wECDSA
- CTC_SHA384wRSA
- CTC_SHA384wECDSA
- CTC_SHA512wRSA
- CTC_SHA512wECDSA

次に、以下のようにしてサブジェクト情報を初期化します。ただし実際には、ユーザの環境に応じた値を入力します。詳細はhttps://github.com/wolfSSL/wolfssl-examples/blob/master/certgen/csr_example.cにも掲載しています。

```
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);
```

そして、上記の鍵生成例からの変数 `key` と `rng` を使用して、署名付き CSR を生成できます。もちろん、有効なものであれば wolfSSL によって生成したものでもなくとも構いません。

```
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 */
```

次に、リクエストに署名して有効にします。上記の鍵生成例から `rng` 変数を使用します。こちらも同様に、wolfSSL によって生成したものである必要はありません。

```
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 */
```

最後に、CSR を PEM フォーマットに変換して、CA 機関に送信して証明書の発行に使用します。

```
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 制限事項

証明書には必須のフィールドがあり、CSR では除外されます。CSR には「オプション」とみなされる他のフィールドもあり、証明書では必須です。このため、すべての証明書フィールドを厳密にチェックし、すべてのフィールドを必須と見なす wolfSSL 証明書解析エンジンは、現時点では CSR の使用をサポートしていません。したがって、CSR 生成と最初からの証明書生成はサポートしていますが、wolfSSL は CSR からの証明書生成をサポートしていません。CSR を wolfSSL 解析エンジンに渡すと、現時点ではエラーが返されます。アップデートをお待ちいただくか、info@wolfssl.jpまでお問い合わせください。

7.11 生の ECC 鍵に変換

生の ECC 鍵インポートを用いて、ECC 鍵を PEM から DER に変換する機能を提供しています。これを実行するには、指定された引数で次の関数を実行します。

```
EccKeyToDer(ecc_key*, byte* output, word32 inLen);
```

7.11.1 実装例

```
#define FOURK_BUF 4096  
byte der[FOURK_BUF];  
ecc_key userB;  
  
EccKeyToDer(&userB, der, FOURK_BUF);
```

8 デバッグ

8.1 デバッグとロギング

wolfSSL は、デバッグが制限されている環境でのログメッセージによるデバッグをサポートしています。ロギングをオンにするには関数 `wolfSSL_Debugging_ON()` を使用し、オフにするには `wolfSSL_Debugging_OFF()` を使用します。通常のビルド（リリースモード）では、これらの関数は効果がありません。デバッグビルドでは、これらの関数が確実にオンになるように `DEBUG_WOLFSSL` を定義してください。

wolfSSL 2.0 以降では、ロギングの方法についてより柔軟性を提供するために、ランタイムでロギングコールバック関数を登録できます。ロギングコールバックは関数 `wolfSSL_SetLoggingCb()` で登録できます。

```
int wolfSSL_SetLoggingCb(wolfSSL_Logging_cb log_function);
```

```
typedef void (*wolfSSL_Logging_cb)(const int logLevel,
                                   const char *const logMessage);
```

ログレベルは `wolfssl/wolfcrypt/logging.h` にあり、実装は `logging.c` にあります。デフォルトでは、wolfSSL は `fprintf` を使用して `stderr` にログを出力します。

8.2 エラーコード

デバッグを支援するために、有益なエラーメッセージを提供するよう努めています。

`wolfSSL_read()` と `wolfSSL_write()` の呼び出しは、成功時にはバイト数、接続終了時には 0、エラー時には -1 を返します。これは `read()` や `write()` と同様です。エラーが発生した場合には、以下の 2 つの API を使用してより詳しい情報を入手できます。

関数 `wolfSSL_get_error()` は、現在のエラーコードを返します。現在の `WOLFSSL` オブジェクトと `wolfSSL_read()` または `wolfSSL_write()` の結果値を引数として取り、対応するエラーコードを返します。

```
int err = wolfSSL_get_error(ssl, result);
```

人間が読みやすいエラーコードの説明を得るには、`wolfSSL_ERR_error_string()` を使用します。この関数は `wolfSSL_get_error` の戻り値と出力保存用のバッファを引数として取り、対応するエラーの説明をそのバッファ（以下の例では `errorString`）に出力します。

```
char errorString[80];
wolfSSL_ERR_error_string(err, errorString);
```

ノンブロッキングソケットを使用している場合、`errno` `EAGAIN/EWOULDBLOCK` をテストするか、より正確には特定のエラーコード `SSL_ERROR_WANT_READ` または `SSL_ERROR_WANT_WRITE` をテストできます。

付録 F エラーコードに、wolfSSL と wolfCrypt のエラーコード一覧を掲載しています。

9 ライブラリデザイン

9.1 ライブラリヘッダー

wolfSSL 2.0.0 RC3 以降のリリースでは、ライブラリヘッダファイルを以下の場所に配置しています。

- wolfSSL: wolfssl/
- wolfCrypt: wolfssl/wolfcrypt/
- wolfSSL OpenSSL 互換レイヤー: wolfssl/openssl/

OpenSSL 互換レイヤーを使用する場合、/wolfssl/openssl/ssl.h ヘッダーをインクルードする必要があります。第 13 章 OpenSSL 互換性も併せてご参照ください。

```
#include <wolfssl/openssl/ssl.h>
```

wolfSSL ネイティブ API のみを使用する場合は、/wolfssl/ssl.h ヘッダーのみをインクルードする必要があります。

```
#include <wolfssl/ssl.h>
```

9.2 起動と終了

すべてのアプリケーションは、ライブラリを使用する前に wolfSSL_Init()、プログラム終了時に wolfSSL_Cleanup() を呼び出す必要があります。現在、これらの関数はマルチユーザーモードでセッションキャッシュ用の共有ミューテックスを初期化および解放するだけです。しかし、将来的にはより多くの処理を実行する可能性があるため、常にこれらを実行することをお勧めします。

9.3 構造体の使用

wolfSSL 2.0.0 RC3 リリースではヘッダーファイルの場所の変更に加え、wolfSSL のネイティブ API と OpenSSL 互換レイヤーをより明確に区別するようにしました。これに伴い、wolfSSL のネイティブ API で使用される SSL/TLS 構造体の名前を変更しています。新しい構造体は以下の通りです。なお OpenSSL 互換レイヤーを使用する場合は、引き続き旧称を使用します。詳細は第 13 章 OpenSSL 互換性をご参照ください。

- WOLFSSL (旧称: SSL)
- WOLFSSL_CTX (旧称: SSL_CTX)
- WOLFSSL_METHOD (旧称: SSL_METHOD)
- WOLFSSL_SESSION (旧称: SSL_SESSION)
- WOLFSSL_X509 (旧称: X509)
- WOLFSSL_X509_NAME (旧称: X509_NAME)
- WOLFSSL_X509_CHAIN (旧称: X509_CHAIN)

9.4 スレッドセーフ

wolfSSL はスレッドセーフとなるよう設計しています。グローバルデータ、静的データ、オブジェクトの共有を避けているため、複数のスレッドが競合せず同時にライブラリに入ることができます。ユーザーは、いくつかの領域で潜在的な問題を避けるよう注意する必要があります。

1. クライアントは複数のスレッド間で WOLFSSL オブジェクトを共有できますが、アクセスは同期する必要があります。つまり、同じ SSL ポインタを持つ 2 つの異なるスレッドから同時に Read/Write することはできません。

wolfSSL は共有できない関数に入ったときにより積極的（制約的）な姿勢をとり、他のユーザーをロックアウトする可能性があります。すべてのユーザー（シングルスレッドのユーザーでさえ）はロックの負荷を負担し、マルチスレッドユーザーはスレッド間でオブジェクトを共有していなくても、ラ

イブラリに再入力できません。このペナルティは高すぎるように思えますが、wolfSSL は共有オブジェクトを同期する責任をユーザーの手に委ねています。

2. WOLFSSL ポインタの共有に加えて、ユーザーは構造体を `wolfSSL_new()` に渡す前に `WOLFSSL_CTX` を完全に初期化する必要があります。同じ `WOLFSSL_CTX` で複数の `WOLFSSL` 構造体を作成できますが、`WOLFSSL_CTX` は `wolfSSL_new()` 作成時にのみ読み取られ、`WOLFSSL` オブジェクトが作成されると、`WOLFSSL_CTX` への将来の（または同時の）変更は反映されません。

繰り返しになりますが、複数のスレッドは `WOLFSSL_CTX` への書き込みアクセスを同期させるべきであり、上記の同期と更新の問題を避けるために、単一のスレッドが `WOLFSSL_CTX` を初期化することをお勧めします。

3. 一部の最適化では、スレッドごとにメモリが割り当てられます。固定小数点 ECC キャッシュが有効になっている場合 (FP_ECC)、スレッドはスレッド終了前に `wc_ecc_fp_free()` を使用してキャッシュされたバッファをリリースする必要があります。

9.5 入力と出力バッファ

wolfSSL は、入力と出力に動的バッファを使用しています。これらはデフォルトで 0 バイトであり、`wolfssl/internal.h` の `RECORD_SIZE` 定義によって制御されます。静的バッファよりも大きな入力レコードを受信した場合、一時的に動的バッファが使用され、その後解放されます。静的バッファサイズは `MAX_RECORD_SIZE` (2^{14} または 16,384) にて設定できます。

以前の wolfSSL の動作方法（動的メモリを必要としない 16Kb の静的バッファ）を希望する場合は、`LARGE_STATIC_BUFFERS` を定義することで、その動作を引き続き使用できます。

動的バッファが使用され、ユーザーがバッファサイズより大きい `wolfSSL_write()` をリクエストした場合、`MAX_RECORD_SIZE` までの動的ブロックがデータを送信するために使用されます。データを最大で `RECORD_SIZE` サイズのチャンクでのみ送信したいユーザーは、`STATIC_CHUNKS_ONLY` を定義してください。これにより、wolfSSL は `RECORD_SIZE`（デフォルトでは 128 バイト）まで拡大する I/O バッファを使用するようになります。

10 wolfCrypt の使用法

wolfCrypt は、主に wolfSSL が使用する暗号ライブラリです。速度、小さなフットプリント、移植性のために最適化されています。wolfSSL は、必要に応じて他の暗号ライブラリと交換します。

実装例で使用する型は次の通りです。

```
typedef unsigned char byte;
typedef unsigned int word32;
```

10.1 ハッシュ関数

10.1.1 MD4

注意：MD4 は古く、安全でないと考えられています。可能であれば、異なるハッシュ関数を使用してください。

MD4 を使用するには、MD4 ヘッダー wolfssl/wolfcrypt/md4.h をインクルードします。使用する構造体は Md4 で、これは typedef です。使用する前に、wc_InitMd4() を実行してハッシュの初期化を行う必要があります。ハッシュを更新するには wc_Md4Update() を使用します。

```
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 には、buffer のハッシュ化されたデータのダイジェストが含まれます。

10.1.2 MD5

注意：MD5 は古く、安全でないと考えられています。可能であれば、異なるハッシュ関数を使用してください。

MD5 を使用するには、MD5 ヘッダー wolfssl/wolfcrypt/md5.h をインクルードします。使用する構造体は Md5 で、これは typedef です。使用する前に、wc_InitMd5() を実行してハッシュの初期化を行う必要があります。ハッシュを更新するには wc_Md5Update() を使用します。

```
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 には、buffer のハッシュ化されたデータのダイジェストが含まれます。

10.1.3 SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512

SHA を使用するには、SHA ヘッダー `wolfssl/wolfcrypt/sha.h` をインクルードします。使用する構造体は `Sha` で、これは `typedef` です。使用する前に、`wc_InitSha()` を実行してハッシュの初期化を行う必要があります。ハッシュを更新するには `wc_ShaUpdate()` を使用します。

```
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` には、`buffer` のハッシュ化されたデータのダイジェストが含まれます。

SHA-224、SHA-256、SHA-384、SHA-512 のいずれかを使用するには、上記と同じ手順に従います。ただし、`wolfssl/wolfcrypt/sha256.h` または `wolfssl/wolfcrypt/sha512.h` (SHA-384 もこちら) のいずれかを使用します。SHA-256、SHA-384、SHA-512 関数は、SHA 関数と同様に名付けられています。

SHA-224 の場合、構造体 `Sha224` と共に `wc_InitSha224()` 関数を使用されます。

SHA-256 の場合、構造体 `Sha256` と共に `wc_InitSha256()` 関数を使用されます。

SHA-384 の場合、構造体 `Sha384` と共に `wc_InitSha384()` 関数を使用されます。

SHA-512 の場合、構造体 `Sha512` と共に `wc_InitSha512()` 関数を使用されます。

SHA インターリービングはデフォルトで有効になっています。(通常は、それをサポートするハードウェアアクセラレーションでのみ使用されます。) 無効にするには `NO_WOLFSSL_SHA256_INTERLEAVE` を定義します。ソフトウェア SHA は常にインターリービングをサポートしています。

10.1.4 BLAKE2b

BLAKE2b (SHA-3 ファイナリスト) を使用するには、BLAKE2b ヘッダー `wolfssl/wolfcrypt/blake2.h` をインクルードします。使用する構造体は `Blake2b` で、これは `typedef` です。使用する前に、`wc_InitBlake2b()` を実行してハッシュの初期化を行う必要があります。ハッシュを更新するには `wc_Blake2bUpdate()` を使用します。

```
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);
```

`wc_InitBlake2b()` の 2 番目のパラメータは、最終的なダイジェストサイズです。`digest` には、`buffer` のハッシュ化されたデータのダイジェストが含まれます。

使用例は `wolfCrypt` テストアプリケーション (`wolfcrypt/test/test.c`) の `blake2b_test()` 関数内にあります。

10.1.5 RIPEMD-160

RIPEMD-160 を使用するには、ヘッダー `wolfssl/wolfcrypt/ripemd.h` をインクルードします。使用する構造体は `RipeMd` で、これは `typedef` です。使用する前に、`wc_InitRipeMd()` を実行してハッシュの初期化を行う必要があります。ハッシュを更新するには `wc_RipeMdUpdate()` を使用します。

```
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` には、`buffer` のハッシュ化されたデータのダイジェストが含まれます。

10.2 鍵付きハッシュ関数

10.2.1 HMAC

`wolfCrypt` は現在、メッセージダイジェストのニーズに対して HMAC を提供しています。構造体 `Hmac` はヘッダー `wolfssl/wolfcrypt/hmac.h` にあります。HMAC の初期化は `wc_HmacSetKey()` で行われます。HMAC では、MD5、SHA、SHA-256、SHA-384、SHA-512 をサポートしています。

以下は、SHA-256 の例です。

```
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` には、`buffer` のハッシュ化されたデータのダイジェストが含まれます。

10.2.2 GMAC

`wolfCrypt` はメッセージダイジェストのニーズに対して GMAC も提供しています。構造体 `Gmac` はヘッダー `wolfssl/wolfcrypt/aes.h` にあります (AES-GCM のアプリケーションです)。GMAC の初期化は `wc_GmacSetKey()` で行われます。

```
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` には、`buffer` のハッシュ化されたデータのダイジェストが含まれます。

10.2.3 Poly1305

wolfCrypt はメッセージダイジェストのニーズに対して Poly1305 も提供しています。構造体 Poly1305 はヘッダー wolfssl/wolfcrypt/poly1305.h にあります。Poly1305 の初期化は `wc_Poly1305SetKey()` で行われます。`wc_Poly1305Final()` が呼び出された後、次に Poly1305 を使用する際には、Poly1305 の鍵の設定プロセスを新しい鍵で再度行う必要があります。

```
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 には、buffer のハッシュ化されたデータのダイジェストが含まれます。

10.3 ブロック暗号

10.3.1 AES

wolfCrypt は 16 バイト (128 ビット)、24 バイト (192 ビット)、32 バイト (256 ビット) の鍵サイズの AES をサポートしています。サポートされる AES モードには、CBC、CTR、GCM (GCM-ストリーミング)、OFB、CFB (1、8、128)、SIV、XTS (XTS-ストリーミング)、GMAC、CMAC、ECB、KW (KeyWrap)、CCM-8 が含まれます。

注意: `wc_AesSetKey()` を呼び出し、Aes 構造体を初期化する必要があります。

CBC モードは暗号化と復号の両方でサポートしており、`wc_AesSetKey()` 関数を通じて提供されます。AES を使用するには、ヘッダー wolfssl/wolfcrypt/aes.h をインクルードしてください。AES のブロックサイズは 16 バイトで、IV も 16 バイトである必要があります。関数の使用法は通常、次のとおりです。

```
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 には、平文から生成された暗号文が入ります。

```
/*decrypt*/
```

```
wc_AesSetKey(&dec, key, sizeof(key), iv, AES_DECRYPTION);
wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain には、暗号文を復号することで生成された平文が入ります。

wolfCrypt は AES の動作モードとして CTR (カウンター)、GCM (ガロア/カウンター)、CCM-8 (CBC-MAC 付きカウンター) もサポートしています。CBC と同様に、これらのモードを使用する場合は、ヘッダー

wolfssl/wolfcrypt/aes.h をインクルードしてください。

GCM モードは `wc_AesGcmSetKey()` 関数を通じて暗号化と復号の両方で利用できます。使用例については、`<wolfssl_root>/wolfcrypt/test/test.c` の `aesgcm_test()` 関数をご参照ください。

CCM-8 モードは `wc_AesCcmSetKey()` 関数を通じて暗号化と復号の両方でサポートしています。使用例については、`<wolfssl_root>/wolfcrypt/test/test.c` の `aesccm_test()` 関数をご参照ください。

CTR モードは `wc_AesCtrEncrypt()` 関数を通じて暗号化と復号の両方で利用できます。暗号化と復号のアクションは同一であるため、両方に同じ関数が使用されます。使用例については、ファイル `wolfcrypt/test/test.c` の `aes_test()` 関数をご参照ください。

10.3.1.1 DES および 3DES wolfCrypt は DES および 3DES をサポートしています。（ただし、C 言語では先頭に数字を使用できないため、Des3 と示しています。）これらを使用するには、ヘッダー `wolfssl/wolfcrypt/des.h` をインクルードします。使用できる構造体は Des および Des3 です。初期化は `wc_Des_SetKey()` を通じて行われます。CBC 暗号化/復号は `wc_Des_CbcEncrypt()` を通じて提供されます。Des の鍵サイズは 8 バイト（3DES では 24）で、ブロックサイズは 8 バイトなので、暗号化/復号関数には 8 バイトの倍数のみを渡してください。データがブロックサイズの倍数でない場合は、確実にそうなるようにパディングを追加する必要があります。各 `SetKey()` は、IV（鍵サイズと同じサイズの初期化ベクトル）も取ります。使用方法は通常、次のとおりです。

```
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 には、平文から生成された暗号文が入ります。

```
/*decrypt*/
```

```
wc_Des3_SetKey(&dec, key, iv, DES_DECRYPTION);
wc_Des3_CbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain には、暗号文を復号することで生成された平文が入ります。

10.3.1.2 Camellia wolfCrypt は Camellia ブロック暗号をサポートしています。Camellia を使用するには、ヘッダー `wolfssl/wolfcrypt/camellia.h` をインクルードします。使用できる構造体は Camellia と呼ばれます。初期化は `wc_CamelliaSetKey()` を通じて行われます。CBC 暗号化/復号は `wc_CamelliaCbcEncrypt()` を通じて提供されます。

使用例については、`<wolfssl_root>/wolfcrypt/test/test.c` の `camellia_test()` 関数をご参照ください。

10.4 ストリーム暗号

10.4.1 ARC4

注意：ARC4 は古く、安全でないと考えられています。他のストリーム暗号を使用してください。

wolfCrypt はヘッダー wolfssl/wolfcrypt/arc4.h を通じて ARC4 をサポートしています。ブロックサイズがなく、鍵の長さも任意の長さにできるため、使用方法はブロック暗号よりも簡単です。以下は ARC4 の典型的な使用方法です。

```
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 には、平文から生成された暗号文が入ります。

/*decrypt*/
wc_Arc4SetKey(&dec, key, sizeof(key));
wc_Arc4Process(&dec, plain, cipher, sizeof(cipher));
plain には、暗号文を復号することで生成された平文が入ります。
```

10.4.2 ChaCha

20 ラウンドの ChaCha は ARC4 よりもわずかに高速であり、高いレベルのセキュリティを維持しています。wolfCrypt で使用するには、ヘッダー wolfssl/wolfcrypt/chacha.h をインクルードしてください。ChaCha は通常 32 バイト (256 ビット) の鍵を使用しますが、16 バイト (128 ビット) の鍵も使用できます。

```
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 には、平文から生成された暗号文が入ります。

/*decrypt*/
wc_Chacha_SetKey(&enc, key, keySz);
wc_Chacha_SetIV(&enc, iv, counter);
wc_Chacha_Process(&enc, plain, cipher, sizeof(cipher));
plain には、暗号文を復号することで生成された平文が入ります。
```

`wc_Chacha_SetKey` を呼び出す必要があります。カウンターは、暗号化/復号プロセスを実行するときに異なるブロックから開始することで、部分的に情報を暗号化/復号できるように引数として設定されますが、ほとんどの場合は 0 に設定されます。

**** ChaCha は MAC アルゴリズム（例：Poly1305、HMAC）なしで使用しないでください。 ****

10.5 公開鍵暗号

10.5.1 RSA

wolfCrypt はヘッダー `wolfssl/wolfcrypt/rsa.h` を通じて RSA をサポートしています。RSA 鍵には、公開鍵と秘密鍵の 2 種類があります。公開鍵を使用すると、秘密鍵の保持者のみが復号できるように暗号化できます。また、秘密鍵の保持者が何かに署名することを可能にし、公開鍵を持つ誰もが、秘密鍵の保持者が署名したことを確認できます。使用 방법은 通常、次のとおりです。

```
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*/

RsaPublicKeyDecode(publicKeyBuffer, &idx, &rsaPublicKey,
    ↪ sizeof(publicKeyBuffer));

byte in[] = { /*plain text to encrypt*/ };
byte out[128];
RNG rng;

wc_InitRng(&rng);

word32 outLen = RsaPublicEncrypt(in, sizeof(in), out, sizeof(out),
    ↪ &rsaPublicKey, &rng);
```

これで `out` には、平文 `in` からの暗号文が含まれます。`wc_RsaPublicEncrypt()` は、`out` に書き込まれたバイト数を返すか、エラーの場合は負の数を返します。`wc_RsaPublicEncrypt()` には、暗号化に使用されるパディング用の RNG（乱数ジェネレータ）が必要であり、使用する前に初期化する必要があります。出力バッファが十分な大きさであることを確認するために、最初に `wc_RsaEncryptSize()` を使用します。これにより、`wc_RsaPublicEncrypt()` の成功した呼び出しが書き込むバイト数が返されます。

エラーが発生した場合、`wc_RsaPublicEncrypt()` から負の戻り値が返されます。その場合、`wc_ErrorString()` を呼び出して、発生したエラーを説明する文字列を取得できます。

```
void wc_ErrorString(int error, char* buffer);
```

バッファが少なくとも `MAX_ERROR_SZ` バイト（80）であることを確認してください。

続いて、`out` を復号します。

```
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_RsaPrivateKeyDecrypt(out, outLen, plain,
    sizeof(plain), &rsaPrivateKey);
```

これで、plain には plainSz バイトまたはエラーコードが含まれます。wolfCrypt での各タイプの完全な例については、ファイル wolfcrypt/test/test.c をご参照ください。wc_RsaPrivateKeyDecode関数は、生の DER 形式の鍵のみを受け入れます。

10.5.2 DH (Diffie-Hellman)

wolfCrypt はヘッダー wolfssl/wolfcrypt/dh.h を通じて Diffie-Hellman をサポートしています。Diffie-Hellman 鍵交換アルゴリズムにより、2 人の当事者間で共有秘密鍵を確立できます。使用 방법은 通常、次の例のようになります。ここで **sideA** と **sideB** は 2 つの当事者を示します。

以下の例では、dhPublicKey には認証局によって署名された（または自己署名された）Diffie-Hellman 公開パラメータが含まれています。privA は sideA に対して生成された秘密鍵、pubA は sideA に対して生成された公開鍵を保持します。agreeA は、両者が合意した相互鍵を保持します。

```
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(tmp, &idx, &dhPublicKey, publicKeyBuffer);
wc_InitRng(&rng); /*Initialize random number generator*/
```

wc_DhGenerateKeyPair() は、dhPublicKey の初期公開パラメータに基づいて公開鍵と秘密鍵の DH 鍵を生成します。

```
wc_DhGenerateKeyPair(&dhPublicKey, &rng, privA, &privASz,
                    pubA, &pubASz);
```

sideB が彼らの公開鍵 (pubB) を sideA に送信した後、sideA はwc_DhAgree()関数を使用して相互に合意された鍵 (agreeA) を生成できます。

```
wc_DhAgree(&dhPublicKey, agreeA, &agreeASz, privA, privASz,
          pubB, pubBSz);
```

これで、agreeA は sideA の相互に生成された鍵（サイズは agreeASz バイト）を保持します。同じプロセスが sideB でも行われています。

wolfCrypt での Diffie-Hellman の完全な実装例については、ファイル wolfcrypt/test/test.c をご参照ください。

10.5.3 EDH (Ephemeral Diffie-Hellman)

wolfSSL サーバーは Ephemeral Diffie-Hellman を実行できます。この機能を追加するためにビルドの変更は必要ありませんが、EDH 暗号スイートを有効にするためにはサーバー側でアプリケーションが一時的なグループパラメータを登録する必要があります。これには、次の API を使用します。


```
int wolfSSL_SetTmpDH(WOLFSSL* ssl, unsigned char* p,
                    int pSz, unsigned char* g, int gSz);
```

実装例として提供しているサーバーと echoserver は、SetDH() からこの関数を使用します。

10.5.4 DSA (デジタル署名アルゴリズム)

wolfCrypt はヘッダー wolfssl/wolfcrypt/dsa.h を通じて DSA と DSS をサポートしています。DSA は、与えられたデータハッシュに基づいてデジタル署名を作成することを可能にします。DSA は SHA ハッシュアルゴリズムを使用してデータブロックのハッシュを生成し、その後署名者の秘密鍵を使用してそのハッシュに署名します。標準的な使用法は次のようになります。

まず、DSA 鍵構造 (key) を宣言し、署名される初期メッセージ (message) を初期化し、DSA 鍵バッファ (dsaKeyBuffer) を初期化します。

```
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*/ }
```

次に、SHA 構造体 (sha)、乱数ジェネレータ (rng)、SHA ハッシュを格納する配列 (hash)、署名を格納する配列 (signature)、idx (dsaKeyBuffer の読み込み開始位置を示す)、検証後の戻り値を保持する int 型の変数 (answer) を宣言します。

```
Sha sha;
RNG rng;
byte hash[SHA_DIGEST_SIZE];
byte signature[40];
word32 idx = 0;
int answer;
```

SHA ハッシュを設定して作成します。wolfCrypt の SHA アルゴリズムの詳細については、本章の「SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512」節をご参照ください。

message の SHA ハッシュは、変数 hash に格納されます。

```
wc_InitSha(&sha);
wc_ShaUpdate(&sha, message, sizeof(message));
wc_ShaFinal(&sha, hash);
```

DSA 鍵構造体を初期化し、構造体の鍵値を設定し、乱数ジェネレータ (rng) を初期化します。

```
wc_InitDsaKey(&key);
wc_DsaPrivateKeyDecode(dsaKeyBuffer, &idx, &key,
                      sizeof(dsaKeyBuffer));
wc_InitRng(&rng);
```

wc_DsaSign() 関数は、DSA 秘密鍵、ハッシュ値、および乱数ジェネレータを使用して署名 (signature) を作成します。

```
wc_DsaSign(hash, signature, &key, &rng);
```

署名を検証するには、wc_DsaVerify() を使用します。検証が成功すると、answer は「1」となります。

完了したら、wc_FreeDsaKey() を使用して DSA 鍵構造体を解放します。

```
wc_DsaVerify(hash, signature, &key, &answer);
wc_FreeDsaKey(&key);
```

11 SSL/TLS チュートリアル

11.1 はじめに

wolfSSL 組み込み SSL/TLS ライブラリは、SSL と TLS を用いて通信セキュリティを強化するために、既存のアプリケーションやデバイスに簡単に統合できます。組み込みおよび RTOS 環境を主な対象としており、優れたパフォーマンスを維持しながら最小限のフットプリントで動作します。wolfSSL の最小ビルドサイズは、選択されたビルドオプションと使用されるプラットフォームに応じて 20~100kB の範囲です。

このチュートリアルの目的は、シンプルなアプリケーションへの SSL と TLS の統合を案内することです。このチュートリアルを通じて、SSL/TLS についての理解も深まることを期待しています。これから echoserver と echoclient という簡単な実装例を使用して、wolfSSL によりできるだけシンプルにアプリケーションに SSL/TLS サポートを追加する手順を示します。echoserver と echoclient の実装例は、Richard Stevens、Bill Fenner、Andrew Rudoff による書籍 [UNIX ネットワークプログラミング、第 1 版、第 3 版](#) を参考にしています。

このチュートリアルは、読者が GNU GCC コンパイラを使用した C コードの編集とコンパイルに慣れており、公開鍵暗号化の概念に精通していることを前提としています。書籍「UNIX ネットワークプログラミング」へのアクセスは必要ではありません。

11.1.1 このチュートリアルで使用される例

- echoclient - 図 5.4、124 ページ
- echoserver - 図 5.12、139 ページ

11.2 SSL/TLS の簡単な概要

TLS (Transport Layer Security) と **SSL** (Secure Sockets Layer) は、さまざまなトランスポートプロトコル間で安全な通信を可能にする暗号プロトコルです。主に使用されるトランスポートプロトコルは TCP/IP です。SSL/TLS の最新バージョンは TLS 1.3 です。wolfSSL は SSL 3.0、TLS 1.0、1.1、1.2、1.3 に加えて DTLS 1.0、1.2、1.3 をサポートしています。

SSL と TLS は OSI 参照モデルのトランスポート層とアプリケーション層の間に位置し、任意のプロトコル (TCP/IP、Bluetooth など) が下層のトランスポート媒体として機能します。HTTP や FTP、SMTP などのアプリケーションプロトコルは、SSL の上の層で構成されます。SSL/TLS が OSI モデルにどのように適合するかを図と、SSL/TLS ハンドシェイクプロセスの簡単な図は、付録 D SSL/TLS の概要にあります。

11.3 ソースコードの入手

このチュートリアルで使用されるすべてのソースコードは、wolfSSL のウェブサイトからダウンロードできます。この ZIP ファイルには、このチュートリアルで使用する echoserver と echoclient の両方のオリジナルコードと、完成されたソースコードが含まれています。

<https://www.wolfssl.com/documentation/ssl-tutorial-2.5.zip>

```
/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 基本例の修正

このチュートリアルとそれに付随するソースコードは、さまざまなプラットフォームに容易に移植できるように設計されています。そのため、またアプリケーションに SSL と TLS を追加する方法に焦点を当てるため、基本的な例はできるだけシンプルにしています。不必要な複雑さを取り除く、または動作させるプラットフォームの範囲を広げるために、Unix ネットワークプログラミングに掲載の実装例にいくつかの修正が加えられています。このチュートリアルの移植性を高めるために何か他にできることがあると思われる場合は、info@wolfssl.jpまでお知らせください。

以下は、上記の書籍に記載されているオリジナルの echoserver と echoclient の例に加えられた修正のリストです。

11.4.1 echoserver (tcpserv04.c) への修正

- `fork()` 関数の呼び出しを削除しました。これは `fork()` が Windows でサポートされていないためです。その結果、echoserver は同時に 1 つのクライアントのみを受け入れます。この削除に伴い、シグナル処理も削除しました。
- `str_echo()` 関数を `str_echo.c` ファイルから `tcpserv04.c` ファイルに移動しました
- クライアントのアドレスと接続ポートを表示するために `printf` 文を追加しました。
`printf("Connection from %s, port %d\n", inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)), ntohs(cliaddr.sin_port));`
- リスニングソケットの作成後に「Address already in use」バインドエラーを排除するために、`setsockopt()` への呼び出しを追加しました。
- 新しいコンパイラの警告をクリーンアップするための軽微な調整を行いました。

11.4.2 echoclient (tcpcli01.c) への修正

- `str_cli()` 関数を `str_cli.c` ファイルから `tcpcli01.c` ファイルに移動しました。
- 新しいコンパイラの警告をクリーンアップするための軽微な調整を行いました。

11.4.3 unp.h ヘッダーへの修正

- このヘッダーは、この例に必要なもののみを含むように簡略化しました。

これらのソースコード例では、特定の関数が大文字になっていることに注意してください。例えば、`Fputs()` や `Writen()` などです。Unix ネットワークプログラミングの著者たちは、エラーチェックを明確に処理するために、通常関数のためのカスタムラッパー関数を書いています。これについての詳細な説明は、Unix ネットワークプログラミング の **セクション 1.4** (11 ページ) をご参照ください。

11.5 wolfSSL のビルドとインストール

始める前に、上記の「ソースコードの入手」節を参照し、実装例のコード (echoserver と echoclient) をダウンロードしてください。このセクションでは、wolfSSL 組み込み SSL/TLS ライブラリをシステムにダウンロード、設定、インストールする方法について説明します。

wolfSSL の最新バージョンを wolfSSL の [ダウンロードページ](#) からダウンロードしてインストールする必要があります。

利用可能なビルドオプションの完全なリストについては、第 2 章 wolfSSL のビルドをご参照ください。wolfSSL は移植性を念頭に置いて書かれており、ほとんどのシステムで簡単にビルドできるはずです。wolfSSL のビルドに問題がある場合は、wolfSSL の [製品サポートフォーラム](#) でサポートを求めるか、info@wolfssl.jpまでお問い合わせください。

Linux、*BSD、OS X、Solaris、またはその他の *nix ライクなシステムで wolfSSL をビルドする場合、autoconf システムを使用できます。Windows での特定の指示については、wolfSSL マニュアルの第 2 章 wolfSSL のビルドセクションをご参照ください。wolfSSL を設定してビルドするには、ターミナルから次の 2 つのコマンドを実行します。任意のビルドオプションを ./configure に追加することもできます。

(例: ./configure --enable-opensslextra)

```
./configure  
make
```

wolfSSL をインストールするには、以下を実行します。

```
sudo make install
```

これにより wolfSSL ヘッダーが /usr/local/include/wolfssl に、wolfSSL ライブラリが /usr/local/lib にインストールされます。ビルドをテストするには、wolfSSL のルートディレクトリからテストスイートアプリケーションを実行します。

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

wolfCrypt と wolfSSL に対して一連のテストが実行され、正しくインストールされたかテストされます。テストスイートアプリケーションを正常に実行できると、以下のような出力が表示されるはずです。

```
-----  
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!

wolfSSL がインストールされたので、実装例のコードに SSL/TLS 機能を追加する修正を始めることができます。まず echoclient に SSL/TLS を追加し、その後 echoserver に進みます。

11.6 初回コンパイル

SSL チュートリアルの ZIP ファイルから実装例の echoclient と echoserver コードをコンパイルして実行するために、付属の Makefile を使用できます。echoclient または echoserver ディレクトリに移動 (cd) して、次のコマンドを実行します。

```
make
```

これにより実装例のコードがコンパイルされ、ビルドされているものに応じて echoserver または echoclient という名前の実行可能ファイルが生成されます。Makefile で使用される GCC コマンドは以下の通りです。提供された Makefile を使用せずに実装例の 1 つをビルドする場合は、実装例のディレクト

リに移動し、次のコマンドの `tcpcli01.c` (echoclient) または `tcpserv04.c` (echoserver) を正しいソースファイルに置き換えます。

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include
```

これにより、現在の実装例が実行可能ファイルにコンパイルされ、「echoserver」または「echoclient」アプリケーションが作成されます。コンパイル後に実装例の1つを実行するには、現在のディレクトリを目的の例のディレクトリに変更し、アプリケーションを起動します。例えば、echoserver を起動するには次を使用します。

```
./echoserver
```

ローカルホストで echoclient をテストするために、2つ目のターミナルウィンドウを開きます。アプリケーションを起動する際にはサーバーの IP アドレスを指定する必要があります。今回の場合では、127.0.0.1 を指定します。現在のディレクトリを「echoclient」ディレクトリに変更 (cd) し、次のコマンドを実行します。なお、echoserver はすでに実行されている必要があります。

```
./echoclient 127.0.0.1
```

echoserver と echoclient の両方が実行されている場合、echoserver は echoclient から受け取った入力をエコーバックするはずですが、echoserver または echoclient を終了するには、`_Ctrl + C_` を使用します。現在、これらの2つの実装例の間で行き来しているエコーデータは平文で送信されています。わずかなスキルさえ持っていれば、誰でも簡単にクライアントとサーバーの間に介入して通信を傍受できてしまいます。

11.7 ライブラリ

コンパイルされた wolfSSL ライブラリは `libwolfssl` と名付けられます。特に設定されていない限り、wolfSSL のビルドとインストールプロセスは、次のディレクトリの下に共有ライブラリのみを作成します。適切なビルドオプションを使用して、共有ライブラリと静的ライブラリの両方を有効または無効にすることができます。

```
/usr/local/lib
```

私たちが最初に行う必要があるのは、wolfSSL ライブラリを私たちの実装例のアプリケーションにリンクすることです。GCC コマンドを修正 (echoserver を例として使用) して、次の新しいコマンドを参照してください。wolfSSL はヘッダーファイルとライブラリを標準的な場所にインストールするため、コンパイラは明示的な指示 (`-l` または `-L` を使用) なしにそれらを見つけることができます。 `-lwolfssl` を使用することにより、コンパイラは自動的に正しいタイプのライブラリ (静的または共有) を選択します。

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include -lm -lwolfssl
```

11.8 ヘッダー

最初に行う必要があるのは、クライアントとサーバーの両方に wolfSSL ネイティブ API ヘッダーを含めることです。クライアントの `tcpcli01.c` ファイルとサーバーの `tcpserv04.c` ファイルの上部に次の行を追加します。

```
#include <wolfssl/ssl.h>
```

11.9 起動/シャットダウン

コード内で wolfSSL を使用する前に、ライブラリと `WOLFSSL_CTX` を初期化する必要があります。wolfSSL は `wolfSSL_Init()` を呼び出すことによって初期化されます。これはライブラリを使用して他のことを行う前に、必ず実行する必要があります。

11.10 WOLFSSL_CTX ファクトリ

WOLFSSL_CTX 構造体 (wolfSSL Context) には、証明書情報を含む各 SSL/TLS 接続のグローバルな値が含まれます。単一の WOLFSSL_CTX は、作成された任意の数の WOLFSSL オブジェクトで使用できます。これにより、信頼された CA 証明書のリストなどの特定の情報を、一度だけロードして使用できます。

新しい WOLFSSL_CTX を作成するには、`wolfSSL_CTX_new()` を使用します。この関数は、クライアントまたはサーバーが使用する SSL/TLS プロトコルを定義する引数が必要です。目的のプロトコルを選択するためのいくつかのオプションがあります。wolfSSL は現在、SSL 3.0、TLS 1.0、TLS 1.1、TLS 1.2、TLS 1.3、DTLS 1.0、DTLS 1.2、DTLS 1.3 をサポートしています。これらの各プロトコルには、`wolfSSL_CTX_new()` の引数として使用できる対応する関数があります。使用可能なクライアントとサーバーのプロトコルオプションを以下に示します。SSL 2.0 は数年間不安定であるため、wolfSSL ではサポートしていません。

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()`; - SSL 3.0 から TLS 1.3 までの最高バージョンを使用
- `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()`; - クライアントが TLS 1.0+ で接続するのを許可
- `wolfDTLSv1_server_method()`; - DTLS 1.0
- `wolfDTLSv1_2_server_method()`; - DTLS 1.2
- `wolfDTLSv1_3_server_method()`; - DTLS 1.3

echoclient が echoserver に接続するとき、サーバーの識別情報を確認できるように、CA（認証局）証明書を WOLFSSL_CTX にロードする必要があります。WOLFSSL_CTX に CA 証明書をロードするには、`wolfSSL_CTX_load_verify_locations()` を使用します。この関数は、WOLFSSL_CTX ポインタ、証明書ファイル、パス値の 3 つの引数が必要です。パス値は PEM 形式の CA 証明書を含むディレクトリを指します。証明書を検索するとき、wolfSSL はパスの場所を見る前に証明書ファイルの値を調べます。この場合、1 つの CA ファイルを指定するためパスを指定する必要はありません。したがって、パス引数には NULL を使用します。`wolfSSL_CTX_load_verify_locations` 関数は SSL_SUCCESS または SSL_FAILURE のいずれかを返します。

```
wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX* ctx, const char* file, const
↪ char* path)
```

これらのことをまとめると（ライブラリの初期化、プロトコルの選択、CA 証明書）、次のようになります。ここでは、TLS 1.2 を使用することになります。

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);
}

```

上記のコードを tcpcli01.c の main() 内の、「ユーザーが引数として IP アドレスを渡してクライアントを起動したかのチェック」の後に追加します。

EchoServer:

WOLFSSL_CTX に証明書をロードする場合、CA 証明書に加えてサーバー証明書と鍵ファイルをロードする必要があります。これにより、サーバーはクライアントに識別情報確認のための証明書を送信できるようになります。

```
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);
}

```

上記のコードは、tcpserv04.c の変数定義の後の main() の先頭に追加する必要があります。完成した

バージョンのコードは、SSL チュートリアルの ZIP ファイルに含まれています。必要に応じて参照してください。

wolfSSL と WOLFSSL_CTX が初期化されたら、アプリケーションが SSL/TLS の使用を完全に終了したときに WOLFSSL_CTX オブジェクトと wolfSSL ライブラリが解放されるようにしてください。クライアントとサーバーの両方で、以下の 2 行を main() 関数の最後（クライアントでは exit() の呼び出しの直前）に配置する必要があります。

```
wolfSSL_CTX_free(ctx);
wolfSSL_Cleanup();
```

11.11 WOLFSSL オブジェクト

11.11.1 EchoClient

TCP 接続の後に WOLFSSL オブジェクトを作成し、ソケットファイルディスクリプタをセッションに関連付ける必要があります。echoclient の実装例では、以下に示す connect() の呼び出しの後にこれを行います。

```
/* Connect to socket file descriptor */
connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
```

接続直後に、wolfSSL_new() 関数を使用して新しい WOLFSSL オブジェクトを作成します。この関数は成功した場合は WOLFSSL オブジェクトへのポインタを返し、失敗した場合は NULL を返します。その後、ソケットファイルディスクリプタ (sockfd) を新しい WOLFSSL オブジェクト (ssl) に関連付けることができます。

```
/* 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);
```

ここで注目すべき点は、wolfSSL_connect() 関数を呼び出していないことです。wolfSSL_connect() 中に呼び出されます。今回、明示的に wolfSSL_connect() にそれを行わせています。

11.11.2 EchoServer

main メソッドの for ループの最後で WOLFSSL オブジェクトを挿入し、ソケットファイルディスクリプタ (connfd) を WOLFSSL オブジェクト (ssl) に関連付けます。

```
/* 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);
```

繰り返しになりますが、TCP 接続の後に WOLFSSL オブジェクトを作成し、ソケットファイルディスクリプタをセッションに関連付ける必要があります。

11.12 データの送受信

11.12.1 EchoClient での送信

次のステップは、データを安全に送信し始めることです。echoclient の例では、main() 関数が送受信処理を str_cli() に渡していることに注意してください。str_cli() 関数は、私たちの関数で置き換えるところです。まず、str_cli() 関数で WOLFSSL オブジェクトにアクセスするため、引数を追加して ssl 変数を str_cli() に渡します。WOLFSSL オブジェクトが str_cli() 関数内で使用されるようになるため、sockfd パラメータを削除します。この修正を行った新しい str_cli() 関数のプロトタイプを以下に示します。

```
void str_cli(FILE *fp, WOLFSSL* ssl)
```

main() 関数では、新しい引数 (ssl) が str_cli() に渡されます。

```
str_cli(stdin, ssl);
```

str_cli() 関数内では、Writen() と Readline() を wolfSSL_write() 関数の呼び出しに置き換え、元のソケットファイルディスクリプタ (sockfd) の代わりに WOLFSSL オブジェクト (ssl) を使用します。

新しい str_cli() 関数を以下に示します。wolfSSL_write の呼び出しが成功したかどうかを確認する必要があることに注意してください。

Unix プログラミングの著者たちは、Writen() 関数にエラーチェックを組み込んでいたため、それが置き換えられた後にその分を補う必要があります。新しい int 変数 n を追加して wolfSSL_read の戻り値を監視し、バッファの内容 recvline を出力する前に、読み込んだデータの末尾を \0 でマークします。

```
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);
    }
}
```

最後に行うことは、完全に終了するときに WOLFSSL オブジェクトを解放することです。main() 関数内の WOLFSSL_CTX を解放する行の直前に、wolfSSL_free() を呼び出します。

```
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 EchoServer での受信

エコーサーバーは読み取りと書き込みを処理するために `str_echo()` を呼び出します。(クライアントでは `str_cli()` を呼び出しました。) クライアントと同様に、`sockfd` パラメータを `WOLFSSL` オブジェクト (`ssl`) パラメータに置き換えて `str_echo()` を修正します。

```
void str_echo(WOLFSSL* ssl)
```

`Read()` と `Writen()` の呼び出しを `wolfSSL_read()` 関数への呼び出しに置き換えます。戻り値のエラーチェックを含む、修正された `str_echo()` 関数を以下に示します。`read()` から `wolfSSL_read()` への変更に対応するため、変数 `n` のタイプが `ssize_t` から `int` に変更されていることに注意してください。

```
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");
}
```

`main()` で `for` ループ (すぐに `while` ループに変更される) の最後に `str_echo()` 関数を呼び出します。この関数の後、ループ内で `WOLFSSL` オブジェクトを解放し `connfd` ソケットを閉じる呼び出しを行います。

```
str_echo(ssl);                                /* process the request */

wolfSSL_free(ssl);                            /* Free WOLFSSL object */
Close(connfd);
```

`exit` の呼び出しの前に、必ず `ctx` を解放してクリーンアップします。

11.13 シグナル処理

11.13.1 Echoclient / Echoserver

`echoclient` と `echoserver` では、ユーザーが「Ctrl+C」を使用してアプリを閉じる場合のシグナルハンドラを追加する必要があります。エコーサーバーは継続的にループで実行されています。このため、ユーザーが「Ctrl+C」を押したときにそのループを中断する手段を実装する必要があります。これを行うにはまず、終了変数 (`cleanup`) が `true` に設定されたときに終了する `while` ループに変更する必要があります。

`tcpserv04.c` の `#include` 文の直後に、`cleanup` という新しい `static int` 変数を定義します。

```
static int cleanup; /* To handle shutdown */
```

`echoserver` ループを `for` ループから `while` ループに変更します。

```
while(cleanup != 1)
{
```

```
/* echo server code here */
}
```

echoserver ではハンドラが終了した後、シグナルが処理される前に実行していた呼び出しを OS が再開してしまわないようにする必要があります。これを無効にすることで、シグナルが処理された後、OS は `accept()` への呼び出しを再開しません。これを行わなければ、echoserver がリソースをクリーンアップして終了する前に、別のクライアントが接続してしまい、これを切断するのを待たなければならなくなってしまいます。シグナルハンドラを定義し `SA_RESTART` をオフにするには、まず echoserver の `main()` 関数に `act` と `oact` の構造体を定義します。

```
struct sigaction act, oact;
```

変数宣言の後、main 関数の `wolfSSL_Init()` の呼び出しの前に以下のコードを挿入します。

```
/* 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 */
```

echoserver の `sig_handler` 関数を以下に示します。

```
void sig_handler(const int sig)
{
    printf("\nSIGINT handled.\n");
    cleanup = 1;
    return;
}
```

以上で、echoclient と echoserver で TLSv1.2 が有効になりました。

私たちが行ったことは、以下のとおりです。

- wolfSSL ヘッダーをインクルード
- wolfSSL を初期化
- 使用したいプロトコルを選択した `WOLFSSL_CTX` 構造体を作成
- データの送受信に使用する `WOLFSSL` オブジェクトを作成
- `Writen()` と `Readline()` の呼び出しを `wolfSSL_write()` に置き換え
- `WOLFSSL`、`WOLFSSL_CTX` を解放
- シグナルハンドラでクライアントとサーバーのシャットダウンを確実に処理

SSL 接続の動作を設定および制御するための多くの側面とメソッドがあります。より詳細な情報については、wolfSSL ドキュメントをご参照ください。

繰り返しになりますが、完成したソースコードはこのセクションの先頭でダウンロードした ZIP ファイルにあります。

11.14 証明書

テスト目的では、wolfSSL が提供する証明書を使用できます。これらは wolfSSL のダウンロードに含まれており、特にこのチュートリアルでは、`finished_src` フォルダで見つけることができます。

本番アプリケーションでは、信頼できる認証局から正確で正当な証明書を取得する必要があります。

11.15 まとめ

このチュートリアルでは、wolfSSL 組み込み SSL/TLS ライブラリを、簡単なクライアントとサーバーアプリケーションに統合するプロセスを説明しました。この例はシンプルですが、ご自身のアプリケーションに SSL/TLS を追加するために同じ手順を適用できます。wolfSSL 組み込み SSL/TLS ライブラリは、サイズと速度の両方に最適化されたコンパクトで効率的なパッケージで、必要とするすべての機能を提供します。

GPLv2 と標準的な商用ライセンスのデュアルライセンスで提供しているため、ウェブサイトから直接 wolfSSL のソースコードをダウンロードできます。質問やコメントがあれば、サポートフォーラム (<https://www.wolfssl.com/forums>) に投稿してください。製品についての詳細情報が必要な場合は、info@wolfssl.jpまでお問い合わせください。

この SSL/TLS チュートリアルに関するフィードバックを歓迎します。より有用にしたり、理解しやすくしたり、移植性を高めるために改善や強化できると思われる場合は、support@wolfssl.comまでお知らせください。

12 組み込みデバイスのベストプラクティス

12.1 秘密鍵の作成

秘密鍵をファームウェアに埋め込むと誰でも鍵を抽出でき、安全性が保たれなくなってしまう。

SSL/TLS を使用するデバイスの秘密鍵作成に関して、いくつかのアイデアがあります。

1. サーバーとして動作する各デバイスは、非組み込み環境と同様に、固有の秘密鍵を持つ必要があります。
2. デバイスの配布前に鍵を配置できない場合は、セットアップ中に生成させます。
3. デバイスがセットアップ中に独自の鍵を生成できない場合、デバイスを設定するクライアントで鍵を生成し、デバイスに送信させます。
4. クライアントが秘密鍵を生成できない場合、クライアントにデバイスの既知のウェブサイトから SSL/TLS 接続を通じて固有の秘密鍵を取得させます。

wolfSSL はこれらのすべてのステップで使用でき、すべての組み込みデバイス内に秘密鍵を安全に配置するのに役立ちます。これらの手順を踏むことで、SSL/TLS 接続自体の安全性を大幅に高めることができます。

12.2 wolfSSL によるデジタル署名と認証

wolfSSL は、アプリケーション、ライブラリ、ファイルを組み込みデバイスにロードする前にデジタル署名するためにも使用されます。なお、デスクトップマシンやサーバーを対象としたほとんどの OS では、システムライブラリを通じてこのような処理を実行できます。しかし、組み込み向けの RTOS では省略されていることも少なくありません。歴史的にほとんどの組み込みアプリケーションにおいて、実行する必要がなかったためです。今日ではデバイスをネットワークに接続することは一般的となり、これに伴ってセキュリティリスクは日々増大しています。したがって、ターゲットデバイスが組み込みまたはモバイルデバイスであっても、配布するものにあらかじめデジタル署名を施し、検証を行うことが非常に重要となっています。

この流れは、セットトップボックス、DVR、POS システム、VoIP や携帯電話、スマートホームデバイス、さらには自動車向けのコンピューティングシステムで特に顕著です。wolfSSL は主要な組み込みおよびリアルタイムオペレーティングシステム、暗号標準、認証機能をサポートしています。組み込みシステム開発者がデジタル署名機能を追加する際、wolfSSL は非常に優れた選択肢となります。

一般的に、組み込みデバイスでコードやファイルに署名を行うには、次のステップを実行します。

1. 組み込みシステム開発者が、RSA 鍵ペアを生成します。
2. サーバー側で、スクリプトベースのツールを実行します。
 1. デバイ스에 로드される 코드의 해시 (예: SHA-256) 을 생성합니다.
 2. 해시에 디지털 서명を行います.
 3. 코드와 디지털 서명을 포함하는 패키지를 생성합니다.
3. 패키지를 RSA 公開鍵とともにデバイスにロードします。デバイス上では再度ハッシュを作成し、デジタル署名を用いて検証を行います。

デバイスでデジタル署名を有効にする利点は、次の通りです。

1. サードパーティがファイルをデバイスにロードできる、安全な方法を簡単に構築できます。
2. 悪意のあるファイルがデバイスに侵入するのを防ぐことができます。
3. ファームウェアアップデートを電子的に保護できます。
4. 権限のない第三者によるファームウェアアップデートを防ぐことができます。

コード署名に関する一般的な情報は、以下のページからご確認いただけます。

https://en.wikipedia.org/wiki/Code_signing

13 OpenSSL 互換性

13.1 OpenSSL との互換性

wolfSSL は wolfSSL ネイティブ API に加えて、OpenSSL 互換性ヘッダー `wolfssl/openssl/ssl.h` を提供しています。これを用いることで、既存の OpenSSL アプリケーションを簡単に wolfSSL に移植したり、移行したりすることができます。OpenSSL 互換性レイヤーの概要については、以下をご覧ください。wolfSSL がサポートする OpenSSL 関数の一覧は、`wolfssl/openssl/ssl.h` ファイルをご確認ください。

OpenSSL 互換性レイヤーは、よく使用される OpenSSL コマンドのサブセットを wolfSSL のネイティブ API 関数にマッピングします。これにより、アプリケーションコードにわずかな変更を加えるだけで、OpenSSL を wolfSSL へ簡単に置き換えることができます。

私たちは、OpenSSL 互換性のテストベッドとして stunnel と Lighttpd を使用しています。これらを弊社でビルドして、OpenSSL 互換 API が正しく機能するか検証しています。

互換性レイヤーを有効にした wolfSSL のビルド手順。

1. `--enable-opensslextra` を有効化するか、マクロ `OPENSSL_EXTRA` を定義します。
`./configure --enable-opensslextra`
2. 最初の wolfSSL ヘッダーとして `<wolfssl/options.h>` をインクルードします。
3. 移行用のヘッダーファイルは、以下の場所にあります。
 - `./wolfssl/openssl/*.h`
 - 例。 `<wolfssl/openssl/ssl.h>`

13.2 wolfSSL と OpenSSL の違い

wolfSSL と OpenSSL にはどのような違いがあるのか、組み込み向けに最適化された TLS ライブラリを使用することで、どのような利点をもたらされるのか、よくご質問をいただきます。OpenSSL は確かに無料で使い始めることができますが、wolfSSL はより柔軟性が高く、既存のプラットフォームに容易に TLS などの機能を統合できます。このほか多くの最新アルゴリズムをサポートしており、以下に示すようなたくさんのメリットを、利用しやすいライセンスモデルでご提供しています。

ここでは、wolfSSL と OpenSSL の主な違いをいくつかご説明します。

1. wolfSSL のビルドサイズは概ね 20~100kB で、条件が揃えば OpenSSL の 20 分の 1 ほどの大きさになります。wolfSSL はリソースに制約の厳しい環境に適した選択肢です。
2. wolfSSL は DTLS を含む TLS 1.3 の最新標準に対応しています。wolfSSL チームは、wolfSSL を常に最新の標準に更新し続けることに専念しています。
3. wolfSSL は、ストリーミングメディアサポート用の暗号を含め、今日利用可能な最高の暗号と標準を提供しています。さらに、liboqs の統合や各種の独自実装により、耐量子暗号アルゴリズムにも対応しています。
4. wolfSSL は、GPLv2 のオープンソースライセンスと商用ライセンスのデュアルライセンスで提供していますが、OpenSSL は複数のソースに基づく独自のライセンスでのみ利用可能です。
5. wolfSSL は、ユーザーとその安全性を気にかけ、常に支援を惜しまない優れた企業によるサポートを提供しています。wolfSSL チームは主にモンタナ州ボーズマン、オレゴン州ポートランド、ワシントン州シアトルを拠点とし、その他世界中で多数のメンバーが wolfSSL の向上と拡張に積極的に取り組んでいます。
6. wolfSSL は、多様なプラットフォームサポートと組み込み環境での実装実績を有する、RTOS、モバイル、組み込みシステム向けの優れた SSL/TLS ライブラリです。これらの領域ではほとんどのデバイスに対する移植が完了しています。もし実行できない環境がございましたら、気兼ねなくお知らせください。速やかに対応いたします。

7. wolfSSL は、プラットフォームへできるだけ簡単に SSL/TLS を統合するために、いくつかの抽象化レイヤーを提供しています。OS レイヤー、カスタム I/O レイヤー、C スタンダードライブラリ抽象化レイヤーにより、統合がこれまでになく簡単になりました。
8. wolfSSL は、複数のサポートパッケージを提供し、ユーザーに最適なものをお選びいただけるようにしています。電話、メール、サポートフォーラムを通じてお問い合わせいただくことが可能です。ご質問には迅速かつ正確に回答し、プロジェクトをできるだけ早く進められるようお手伝いいたします。

13.3 サポートしている OpenSSL 構造体

- SSL_METHOD は SSL バージョン情報を保持し、クライアントメソッドかサーバーメソッドのいずれかです。(ネイティブ wolfSSL API での WOLFSSL_METHOD と同等)
- SSL_CTX は証明書を含むコンテキスト情報を保持します。(ネイティブ wolfSSL API での WOLFSSL_CTX と同等)
- SSL は安全な接続のためのセッション情報を保持します。(ネイティブ wolfSSL API での WOLFSSL と同等)

13.4 サポートしている OpenSSL 関数

上記の 3 つの構造体は通常、次のように初期化されます。

```
SSL_METHOD* method = SSLv3_client_method();  
SSL_CTX* ctx = SSL_CTX_new(method);  
SSL* ssl = SSL_new(ctx);
```

このコードは、クライアント側 SSL バージョン 3 メソッドを確立し、そのメソッドに基づいてコンテキストを作成し、コンテキストで SSL セッションを初期化します。

サーバー側のプログラムも同様ですが、SSL_METHOD は SSLv3_server_method()、または利用可能なその他の関数を使用して作成します。サポートしている関数についての詳細は、第 4 章 機能をご参照ください。OpenSSL 互換レイヤーを使用する場合、このセクションの関数は「wolf」プレフィックスを削除する必要があります。

例えば、次に示すネイティブ wolfSSL API 関数であれば、

```
wolfTLsv1_client_method()
```

以下のように変更します。

```
TLsv1_client_method()
```

SSL/TLS 接続が不要となった場合、初期化中に作成された構造体を解放するために以下の関数を実行します。

```
SSL_CTX_free(ctx);  
SSL_free(ssl);
```

SSL_CTX_free() には、関連する SSL_METHOD を解放する追加の責任があります。XXX_free() 関数を使用しないと、リソースリークが発生します。SSL/TLS のものではなくシステムの free() を使用すると、未定義の動作が発生します。必ず上記のようにご使用ください。

アプリケーションが SSL_new() から有効な SSL ポインタを取得すると、SSL/TLS ハンドシェイクプロセスを開始できます。クライアントからは、SSL_connect() が安全な接続の確立を試みます。

```
SSL_set_fd(ssl, sockfd);  
SSL_connect(ssl);
```

SSL_connect() を実行する前に、ユーザーは wolfSSL に有効なソケットファイルディスクリプタ（上の例では sockfd）を提供する必要があります。sockfd は通常、TCP の socket() 関数の返り値として取得

され、その後 TCP の `connect()` を使用して確立されます。以下は、ポート 11111 のローカル wolfSSL サーバーで使用するための、有効なクライアント側ソケット記述子を作成するサンプルコードです。簡略化のために、エラー処理は省略しています。

```
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));
```

接続が確立されると、クライアントはサーバーに対して読み書きができるようになります。TCP の `send()` と `receive()` 関数の代わりに、wolfSSL は SSL 関数 `SSL_write()` と `SSL_read()` を使用します。以下はクライアントデモから抜粋した簡単なサンプルコードです。

```
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);
```

サーバーも同様に接続しますが、TCP API に類似して、`SSL_connect()` の代わりに `SSL_accept()` を使用します。

完全なサーバー/クライアントのサンプルプログラムについては、第 3 章 入門をご参照ください。

13.5 x509 証明書

wolfSSL ではサーバーとクライアントの両方で、**PEM** または **DER** 形式の証明書を使用できます。

一般的な使用方法是次の通りです。

```
SSL_CTX_use_certificate_file(ctx, "certs/cert.pem",
SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "certs/key.der",
SSL_FILETYPE_ASN1);
```

鍵ファイルもどちらの形式でも使用できます。SSL_FILETYPE_PEM はファイルが PEM 形式であることを示し、SSL_FILETYPE_ASN1 はファイルが DER 形式であることを宣言します。鍵ファイルが証明書での使用に適切であることを確認するために、次の関数を使用できます。

```
SSL_CTX_check_private_key(ctx);
```


14 ライセンス

14.1 オープンソース

wolfSSL、wolfCrypt、wolfMQTT、wolfTPM、wolfBoot、wolfSentry は、無料でダウンロードできるソフトウェアです。ユーザーが GPLv2 ライセンスに準拠する限り、ユーザーのニーズに合わせて変更を加えることができます。GPLv2 ライセンスは、gnu.org の Web サイト (<https://www.gnu.org/licenses/old-licenses/gpl-2.0.html>) で確認できます。

wolfSSH は無料でダウンロードできるソフトウェアです。ユーザーが GPLv3 ライセンスに準拠する限り、ユーザーのニーズに合わせて変更を加えることができます。GPLv3 ライセンスは、gnu.org の Web サイト (<https://www.gnu.org/licenses/gpl.html>) で確認できます。

14.2 商用ライセンス

再配布のために wolfSSL 製品を独自のアプライアンスやその他の商用ソフトウェア製品に組み込みたい企業は、商用バージョンのライセンスを取得する必要があります。wolfSSL と wolfCrypt の商用ライセンスは、最終製品または SKU ごとに利用できます。通常、ライセンスは 1 つの製品に対して発行され、無制限のロイヤリティフリー配布が含まれます。カスタムライセンス条項も利用できます。

wolfMQTT、wolfSSH、wolfTPM、wolfBoot、wolfSentry の商用ライセンスもご用意しています。info@wolfssl.jpまでお問い合わせください。

14.3 FIPS 140-2/3 検証

wolfSSL は現在、組み込み FIPS 認証のリーダーです。現在アクティブな FIPS 認証と検証オプションの詳細については、[FIPS 140 の FAQ](#) をご覧いただくか、info@wolfssl.jpまでお問い合わせください。

14.4 サポートパッケージ

wolfSSL 製品のサポートパッケージは、wolfSSL から直接年間ベースで利用可能です。複数のパッケージがあり、お客様のニーズに最適なパッケージを選ぶことができます。詳細については、[サポート・メンテナンスパッケージ](#)をご覧ください。

15 サポートとコンサルティング

15.1 サポートの受け方

一般的な製品サポートのために、wolfSSL 製品ファミリー向けのオンラインフォーラムをご用意しています。ご質問がある場合はフォーラムに投稿するか、wolfSSL に直接お問い合わせください。

- wolfSSL フォーラム：<https://www.wolfssl.com/forums>
- E メールサポート：support@wolfssl.com

wolfSSL 製品に関する情報、ライセンスに関するご質問、または一般的なコメントについては、info@wolfssl.jpへお問い合わせください。サポートパッケージについては、第 14 章 ライセンスをご覧ください。

15.1.1 バグの報告とサポート

バグのご報告や、発生した問題についてご質問のある際は、以下の情報をお知らせください。

1. wolfSSL のバージョン
2. オペレーティングシステムのバージョン
3. コンパイラのバージョン
4. 表示されている正確なエラー
5. この問題を再現するための手順

これらの情報がなければ、問題の原因を特定することは非常に困難です。wolfSSL はお客様のフィードバックを大切にし、できるだけ早くご連絡を差し上げることを最優先事項としています。

15.2 コンサルティング

機能の追加、移植、デザインコンサルティングを提供します。

15.2.1 機能追加と移植

現在当社の製品で提供していない機能を、契約または共同開発により追加することができます。詳細は info@wolfssl.jp へお問い合わせください。

15.2.2 デザインコンサルティング

アプリケーションやフレームワークを SSL/TLS で保護するにあたって、どのように設計すべきかお困りでしたらお手伝いいたします。

当社のコンサルタントは、以下のサービスを提供できます。

1. アセスメント：現在の SSL/TLS 実装を評価し、アドバイスします。
2. 設計：お客様のシステム要件とパラメータを確認し、最適なセキュリティを提供するために wolfSSL をどのようにアプリケーションに実装するか、お客様と緊密に連携して推奨事項を作成します。

アプリケーションやデバイスへの SSL/TLS 環境構築のためのデザインコンサルティングについて詳細を知りたい場合は、info@wolfssl.jpまでお問い合わせください。

16 wolfSSL のアップデート

16.1 製品のリリース情報

最新の情報は、公式 X([@wolfSSL_Japan](https://x.com/wolfSSL_Japan), [@wolfSSL](https://x.com/wolfSSL))にてご確認ください。そのほか、GitHub リポジトリやブログポストもぜひご覧ください。

ニュースレターも発行しています。ご希望の方はinfo@wolfssl.jpまでお問い合わせください。

- GitHub リポジトリ - <https://www.github.com/wolfssl/wolfssl>
- X
 - 日本 https://x.com/wolfSSL_Japan
 - グローバル <https://x.com/wolfSSL>
- Facebook - <https://www.facebook.com/wolfSSL>
- Reddit - <https://www.reddit.com/r/wolfssl/>
- ブログ
 - 日本 <https://wolfssl.jp/blog/>
 - グローバル <https://www.wolfssl.com/blog>

A wolfSSL API リファレンス

A.1 証明書マネージャー API

A.1.1 Functions

	Name
WOLFSSL_CERT_MANAGER *	wolfSSL_CertManagerNew_ex (void * heap) 新しい証明書マネージャーコンテキストを割り当て、初期化します。このコンテキストは SSL の必要性とは独立して使用できます。証明書のロード、証明書の検証、失効ステータスのチェックに使用できます。
WOLFSSL_CERT_MANAGER *	wolfSSL_CertManagerNew (void) 新しい証明書マネージャーコンテキストを割り当て、初期化します。このコンテキストは SSL の必要性とは独立して使用できます。証明書のロード、証明書の検証、失効ステータスのチェックに使用できます。
void	wolfSSL_CertManagerFree (WOLFSSL_CERT_MANAGER * cm) 証明書マネージャーコンテキストに関連するすべてのリソースを解放します。証明書マネージャーを使用する必要がなくなったときにこれを呼び出します。
int	wolfSSL_CertManagerLoadCA (WOLFSSL_CERT_MANAGER * cm, const char * f, const char * d) マネージャーコンテキストへの CA 証明書ロードの場所を指定します。PEM 証明書 CAfile には複数の信頼された CA 証明書を含めることができます。CApath が NULL でない場合、PEM 形式の CA 証明書を含むディレクトリを指定します。
int	wolfSSL_CertManagerLoadCABuffer (WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format)wolfSSL_CTX_load_verify_buffer を呼び出し、関数に渡された cm の情報を失わないように一時的な cm を使用してその結果を返すことで、CA バッファをロードします。
int	wolfSSL_CertManagerUnloadCAs (WOLFSSL_CERT_MANAGER * cm) この関数は CA 署名者リストをアンロードします。
int	wolfSSL_CertManagerUnloadIntermediateCerts (WOLFSSL_CERT_MANAGER * cm) この関数は CA 署名者リストに追加された中間証明書をアンロードします。
int	wolfSSL_CertManagerUnload_trust_peers (WOLFSSL_CERT_MANAGER * cm) この関数は、Trusted Peer リンクリストを解放し、トラステッドピアリストのロックを解除します。
int	wolfSSL_CertManagerVerify (WOLFSSL_CERT_MANAGER * cm, const char * f, int format)Certificate Manager コンテキストで検証する証明書を指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 を指定できます。

	Name
int	wolfSSL_CertManagerVerifyBuffer (WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format) Certificate Manager コンテキストで検証する証明書バッファを指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 を指定できます。
void	wolfSSL_CertManagerSetVerify (WOLFSSL_CERT_MANAGER * cm, VerifyCallback verify_callback) この関数は、Certificate Manager 内に verifyCallback 関数を設定します。存在する場合、ロードされた各証明書に対して呼び出されます。検証エラーがある場合、verify callback を使用してエラーをオーバーライドできます。
int	wolfSSL_CertManagerEnableCRL (WOLFSSL_CERT_MANAGER * cm, int options) Certificate Manager で証明書を検証する際に、Certificate Revocation List チェックをオンにします。デフォルトでは、CRL チェックはオフです。options には、リーフ証明書のみ (デフォルト) ではなくチェーン内の各証明書に対して CRL チェックを実行する WOLFSSL_CRL_CHECKALL が含まれます。
int	wolfSSL_CertManagerDisableCRL (WOLFSSL_CERT_MANAGER * cm) Certificate Manager で証明書を検証する際に、Certificate Revocation List チェックをオフにします。デフォルトでは、CRL チェックはオフです。この関数を使用して、以前に CRL チェックを有効にしていたこの Certificate Manager コンテキストに対して、CRL チェックを一時的または永続的に無効にできます。
int	wolfSSL_CertManagerLoadCRL (WOLFSSL_CERT_MANAGER * cm, const char * path, int type, int monitor) エラーチェックを行い、LoadCRL() に渡して失効チェックのために CRL に証明書をロードします。更新された CRL をロードするには、まず wolfSSL_CertManagerFreeCRL を呼び出してから、新しい CRL をロードします。
int	wolfSSL_CertManagerLoadCRLBuffer (WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int type) この関数は、BufferLoadCRL を呼び出して CRL ファイルをロードします。
int	wolfSSL_CertManagerSetCRL_Cb (WOLFSSL_CERT_MANAGER * cm, CbMissingCRL cb) この関数は、CRL Certificate Manager コールバックを設定します。HAVE_CRL が定義されており、一致する CRL レコードが見つからない場合、cbMissingCRL が呼び出されます (wolfSSL_CertManagerSetCRL_Cb で設定)。これにより、外部から CRL を取得してロードできます。

	Name
int	wolfSSL_CertManagerSetCRLUpdate_Cb (WOLFSSL_CERT_MANAGER * cm, CbUpdateCRL cb) この関数は、CRL 更新コールバックを設定します。HAVE_CRL と HAVE_CRL_UPDATE_CB が定義されており、CRL が追加されるときに同じ発行者でより低い CRL 番号を持つエントリが存在する場合、既存のエントリと、それを置き換える新しいエントリの詳細と共に CbUpdateCRL が呼び出されます。
int	wolfSSL_CertManagerGetCRLInfo (WOLFSSL_CERT_MANAGER * cm, CrlInfo * info, const byte * buff, long sz, int type) この関数は、エンコードされた CRL バッファから解析された CRL 情報を含む構造体を生成します。
int	wolfSSL_CertManagerCheckOCSP (WOLFSSL_CERT_MANAGER * cm, const unsigned char * der, int sz) この関数は、WOLFSSL_CERT_MANAGER のメンバである ocsEnabled を有効にして、OCSP チェックオプションが有効になっていることを示します。
int	wolfSSL_CertManagerEnableOCSP (WOLFSSL_CERT_MANAGER * cm, int options) OCSP がオフになっている場合にオンにし、設定オプションでコンパイルされている場合に有効にします。
int	wolfSSL_CertManagerDisableOCSP (WOLFSSL_CERT_MANAGER * cm) OCSP 証明書失効を無効にします。
int	wolfSSL_CertManagerSetOCSPOverrideURL (WOLFSSL_CERT_MANAGER * cm, const char * url) この関数は、URL を WOLFSSL_CERT_MANAGER 構造体の ocsOverrideURL メンバにコピーします。
int	wolfSSL_CertManagerSetOCSP_Cb (WOLFSSL_CERT_MANAGER * cm, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx) この関数は、WOLFSSL_CERT_MANAGER 内の OCSP コールバックを設定します。
int	wolfSSL_CertManagerEnableOCSPStapling (WOLFSSL_CERT_MANAGER * cm) この関数は、OCSP ステープリングがオンになっていない場合にオンにし、オプションを設定します。

A.1.2 Functions Documentation

A.1.2.1 function wolfSSL_CertManagerNew_ex

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew_ex(
    void * heap
)
```

新しい証明書マネージャーコンテキストを割り当て、初期化します。このコンテキストは SSL の必要性和は独立して使用できます。証明書のロード、証明書の検証、失効ステータスのチェックに使用できます。

Parameters:

- **none** パラメータなし。

See: [wolfSSL_CertManagerFree](#)

Return:

- WOLFSSL_CERT_MANAGER 成功した場合、呼び出しは有効な WOLFSSL_CERT_MANAGER ポインタを返します。
- NULL エラー状態の場合に返されます。

A.1.2.2 function wolfSSL_CertManagerNew

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew(
    void
)
```

新しい証明書マネージャーコンテキストを割り当て、初期化します。このコンテキストは SSL の必要性とは独立して使用できます。証明書のロード、証明書の検証、失効ステータスのチェックに使用できます。

Parameters:

- **none** パラメータなし。

See: [wolfSSL_CertManagerFree](#)

Return:

- WOLFSSL_CERT_MANAGER 成功した場合、呼び出しは有効な WOLFSSL_CERT_MANAGER ポインタを返します。
- NULL エラー状態の場合に返されます。

Example

```
#import <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
cm = wolfSSL_CertManagerNew();
if (cm == NULL) {
    // 新しい証明書マネージャーの作成エラー
}
```

A.1.2.3 function wolfSSL_CertManagerFree

```
void wolfSSL_CertManagerFree(
    WOLFSSL_CERT_MANAGER * cm
)
```

証明書マネージャーコンテキストに関連するすべてのリソースを解放します。証明書マネージャーを使用する必要がなくなったときにこれを呼び出します。

Parameters:

- **cm** [wolfSSL_CertManagerNew\(\)](#)を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: [wolfSSL_CertManagerNew](#)

Return: none 戻り値なし。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
...
wolfSSL_CertManagerFree(cm);
```

A.1.2.4 function wolfSSL_CertManagerLoadCA

```
int wolfSSL_CertManagerLoadCA(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    const char * d
)
```

マネージャーコンテキストへの CA 証明書ロードの場所を指定します。PEM 証明書 CAfile には複数の信頼された CA 証明書を含めることができます。CApath が NULL でない場合、PEM 形式の CA 証明書を含むディレクトリを指定します。

Parameters:

- **cm** [wolfSSL_CertManagerNew\(\)](#)を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **file** ロードする CA 証明書を含むファイル名へのポインタ。
- **path** ロードする CA 証明書を含むディレクトリパスの名前へのポインタ。証明書ディレクトリが不要な場合は NULL ポインタを使用できます。

See: [wolfSSL_CertManagerVerify](#)

Return:

- SSL_SUCCESS 成功した場合に返されます。
- SSL_BAD_FILETYPE ファイルの形式が間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラーです。
- SSL_FATAL_ERROR 失敗時に返されます。

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) {
    // 証明書マネージャーへの CA 証明書のロードエラー
}
```

A.1.2.5 function wolfSSL_CertManagerLoadCABuffer

```
int wolfSSL_CertManagerLoadCABuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int format
)
```

wolfSSL_CTX_load_verify_buffer を呼び出し、関数に渡された cm の情報を失わないように一時的な cm を使用してその結果を返すことで、CA バッファをロードします。

Parameters:

- **cm** [wolfSSL_CertManagerNew\(\)](#)を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

- **in** 証明書情報のバッファ。
- **sz** バッファの長さ。
- **format** 証明書の形式、PEM または DER。

See:

- `wolfSSL_CTX_load_verify_buffer`
- `ProcessChainBuffer`
- `ProcessBuffer`
- `cm_pick_method`

Return:

- `SSL_FATAL_ERROR` WOLFSSL_CERT_MANAGER 構造体が NULL、または `wolfSSL_CTX_new()` が NULL を返した場合に返されます。
- `SSL_SUCCESS` 正常に実行された場合に返されます。

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){
    // エラーが返されました。失敗ケースのコードブロック。
}
```

A.1.2.6 function `wolfSSL_CertManagerUnloadCAs`

```
int wolfSSL_CertManagerUnloadCAs(
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は CA 署名者リストをアンロードします。

Parameters:

- **cm** `wolfSSL_CertManagerNew()` を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: `UnlockMutex`

Return:

- `SSL_SUCCESS` 関数の正常な実行時に返されます。
- `BAD_FUNC_ARG` WOLFSSL_CERT_MANAGER が NULL の場合に返されます。
- `BAD_MUTEX_E` mutex エラーが発生した場合に返されます。

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){
    // 失敗ケース。
}
```


A.1.2.7 function wolfSSL_CertManagerUnloadIntermediateCerts

```
int wolfSSL_CertManagerUnloadIntermediateCerts(  
    WOLFSSL_CERT_MANAGER * cm  
)
```

この関数は CA 署名者リストに追加された中間証明書をアンロードします。

Parameters:

- **cm** [wolfSSL_CertManagerNew\(\)](#)を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: UnlockMutex

Return:

- SSL_SUCCESS 関数の正常な実行時に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER が NULL の場合に返されます。
- BAD_MUTEX_E mutex エラーが発生した場合に返されます。

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){  
    // 失敗ケース。  
}
```

A.1.2.8 function wolfSSL_CertManagerUnload_trust_peers

```
int wolfSSL_CertManagerUnload_trust_peers(  
    WOLFSSL_CERT_MANAGER * cm  
)
```

この関数は、Trusted Peer リンクリストを解放し、トラステッドピアリストのロックを解除します。

Parameters:

- **cm** [wolfSSL_CertManagerNew\(\)](#)を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: UnLockMutex

Return:

- SSL_SUCCESS 関数が正常に完了した場合。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER が NULL の場合。
- BAD_MUTEX_E WOLFSSL_CERT_MANAGER 構造体のメンバである tpLock が 0(null) の場合、mutex エラー。

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){
```

```
// 関数は正常に実行されませんでした。
}
```

A.1.2.9 function wolfSSL_CertManagerVerify

```
int wolfSSL_CertManagerVerify(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    int format
)
```

Certificate Manager コンテキストで検証する証明書を指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 を指定できます。

Parameters:

- **cm** `wolfSSL_CertManagerNew()` を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **fname** 検証する証明書を含むファイルの名前へのポインタ。
- **format** 検証する証明書のフォーマット - SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。

See:

- `wolfSSL_CertManagerLoadCA`
- `wolfSSL_CertManagerVerifyBuffer`

Return:

- SSL_SUCCESS 成功時。
- ASN_SIG_CONFIRM_E 署名を検証できなかった場合に返されます。
- ASN_SIG_OID_E 署名タイプがサポートされていない場合に返されます。
- CRL_CERT_REVOKED この証明書が失効している場合に返されるエラー。
- CRL_MISSING 現在の発行者 CRL が利用できない場合に返されるエラー。
- ASN_BEFORE_DATE_E 現在の日付が before date より前の場合に返されます。
- ASN_AFTER_DATE_E 現在の日付が after date より後の場合に返されます。
- SSL_BAD_FILETYPE ファイルのフォーマットが間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードに失敗した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラー。

Example

```
int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerVerify(cm, "path/to/cert-file.pem",
    SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // 証明書の検証エラー
}
```

A.1.2.10 function wolfSSL_CertManagerVerifyBuffer

```
int wolfSSL_CertManagerVerifyBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
```

```
    int format
)
```

Certificate Manager コンテキストで検証する証明書バッファを指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 を指定できます。

Parameters:

- **cm** wolfSSL_CertManagerNew() を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **buff** 検証する証明書を含むバッファ。
- **sz** バッファ buf のサイズ。
- **format** buf 内にある検証する証明書のフォーマット - SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS 成功時。
- ASN_SIG_CONFIRM_E 署名を検証できなかった場合に返されます。
- ASN_SIG_OID_E 署名タイプがサポートされていない場合に返されます。
- CRL_CERT_REVOKED この証明書が失効している場合に返されるエラー。
- CRL_MISSING 現在の発行者 CRL が利用できない場合に返されるエラー。
- ASN_BEFORE_DATE_E 現在の日付が before date より前の場合に返されます。
- ASN_AFTER_DATE_E 現在の日付が after date より後の場合に返されます。
- SSL_BAD_FILETYPE ファイルのフォーマットが間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードに失敗した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラー。

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) {
    // 証明書の検証エラー
}
```

A.1.2.11 function wolfSSL_CertManagerSetVerify

```
void wolfSSL_CertManagerSetVerify(
    WOLFSSL_CERT_MANAGER * cm,
    VerifyCallback verify_callback
)
```

この関数は、Certificate Manager 内に verifyCallback 関数を設定します。存在する場合、ロードされた各証明書に対して呼び出されます。検証エラーがある場合、verify callback を使用してエラーをオーバーライドできます。

Parameters:

- **cm** `wolfSSL_CertManagerNew()`を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **verify_callback** コールバックルーチンへの VerifyCallback 関数ポインタ。

See: `wolfSSL_CertManagerVerify`

Return: none 戻り値はありません。

Example

```
#include <wolfssl/ssl.h>

int myVerify(int preverify, WOLFSSL_X509_STORE_CTX* store)
{ // 証明書のカスタム検証を実行 }

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(Protocol define);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
wolfSSL_CertManagerSetVerify(cm, myVerify);
```

A.1.2.12 function wolfSSL_CertManagerEnableCRL

```
int wolfSSL_CertManagerEnableCRL(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

Certificate Manager で証明書を検証する際に、Certificate Revocation List チェックをオンにします。デフォルトでは、CRL チェックはオフです。options には、リーフ証明書のみ (デフォルト) ではなくチェーン内の各証明書に対して CRL チェックを実行する WOLFSSL_CRL_CHECKALL が含まれます。

Parameters:

- **cm** `wolfSSL_CertManagerNew()`を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **options** Certification Manager(cm) を有効にする際に使用するオプション。

See: `wolfSSL_CertManagerDisableCRL`

Return:

- SSL_SUCCESS 成功した場合、呼び出しは戻ります。
- NOT_COMPILED_IN wolfSSL が CRL を有効にしてビルドされていない場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラー。
- SSL_FAILURE CRL コンテキストを適切に初期化できない場合に返されます。

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerEnableCRL(cm, 0);
if (ret != SSL_SUCCESS) {
    // cert manager の有効化エラー
}
```

...

A.1.2.13 function wolfSSL_CertManagerDisableCRL

```
int wolfSSL_CertManagerDisableCRL(
    WOLFSSL_CERT_MANAGER * cm
)
```

Certificate Manager で証明書を検証する際に、Certificate Revocation List チェックをオフにします。デフォルトでは、CRL チェックはオフです。この関数を使用して、以前に CRL チェックを有効にしていたこの Certificate Manager コンテキストに対して、CRL チェックを一時的または永続的に無効にできます。

Parameters:

- **cm** [wolfSSL_CertManagerNew\(\)](#)を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: [wolfSSL_CertManagerEnableCRL](#)

Return:

- SSL_SUCCESS 成功した場合、呼び出しは戻ります。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラー。

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...
ret = wolfSSL_CertManagerDisableCRL(cm);
if (ret != SSL_SUCCESS) {
    // cert manager の無効化エラー
}
...
```

A.1.2.14 function wolfSSL_CertManagerLoadCRL

```
int wolfSSL_CertManagerLoadCRL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * path,
    int type,
    int monitor
)
```

エラーチェックを行い、LoadCRL() に渡して失効チェックのために CRL に証明書をロードします。更新された CRL をロードするには、まず wolfSSL_CertManagerFreeCRL を呼び出してから、新しい CRL をロードします。

Parameters:

- **cm** [wolfSSL_CertManagerNew\(\)](#)を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **path** CRL パスを保持する定数 char 型ポインタ。
- **type** ロードする証明書のタイプ。
- **monitor** LoadCRL() での監視を要求します。

See:

- [wolfSSL_CertManagerEnableCRL](#)
- [wolfSSL_LoadCRL](#)
- [wolfSSL_CertManagerFreeCRL](#)

Return:

- SSL_SUCCESS wolfSSL_CertManagerLoadCRL にエラーがなく、LoadCRL が正常に戻った場合。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合。
- SSL_FATAL_ERROR wolfSSL_CertManagerEnableCRL が SSL_SUCCESS 以外を返した場合。
- BAD_PATH_ERROR path が NULL の場合。
- MEMORY_E LoadCRL がヒープメモリの割り当てに失敗した場合。

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);
```

A.1.2.15 function wolfSSL_CertManagerLoadCRLBuffer

```
int wolfSSL_CertManagerLoadCRLBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int type
)
```

この関数は、BufferLoadCRL を呼び出して CRL ファイルをロードします。

Parameters:

- **cm** WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **buff** 定数 byte 型でバッファです。
- **sz** バッファのサイズを表す long int 型。
- **type** 証明書タイプを保持する long integer 型。

See:

- BufferLoadCRL
- [wolfSSL_CertManagerEnableCRL](#)

Return:

- SSL_SUCCESS 関数がエラーなく完了した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER が NULL の場合に返されます。
- SSL_FATAL_ERROR WOLFSSL_CERT_MANAGER に関連するエラーがある場合に返されます。

Example

```
#include <wolfssl/ssl.h>
```

```
WOLFSSL_CERT_MANAGER* cm;
const unsigned char* buff;
long sz; バッファのサイズ
int type; 証明書タイプ
...
int ret = wolfSSL_CertManagerLoadCRLBuffer(cm, buff, sz, type);
if(ret == SSL_SUCCESS){
```

```

    return ret;
} else {
    // 失敗ケース。
}

```

A.1.2.16 function wolfSSL_CertManagerSetCRL_Cb

```

int wolfSSL_CertManagerSetCRL_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbMissingCRL cb
)

```

この関数は、CRL Certificate Manager コールバックを設定します。HAVE_CRL が定義されており、一致する CRL レコードが見つからない場合、cbMissingCRL が呼び出されます (wolfSSL_CertManagerSetCRL_Cb で設定)。これにより、外部から CRL を取得してロードできます。

Parameters:

- **cm** 証明書の情報を保持する WOLFSSL_CERT_MANAGER 構造体。
- **cb** WOLFSSL_CERT_MANAGER の cbMissingCRL メンバに設定される (*CbMissingCRL) への関数ポインタ。

See:

- CbMissingCRL
- [wolfSSL_SetCRL_Cb](#)

Return:

- SSL_SUCCESS 関数とサブルーチンの実行が成功した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。

Example

```

#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
void cb(const char* url){
    // 関数本体。
}
...
CbMissingCRL cb = CbMissingCRL;
...
if(ctx){
    return wolfSSL_CertManagerSetCRL_Cb(SSL_CM(ssl), cb);
}

```

A.1.2.17 function wolfSSL_CertManagerSetCRLUpdate_Cb

```

int wolfSSL_CertManagerSetCRLUpdate_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbUpdateCRL cb
)

```

この関数は、CRL 更新コールバックを設定します。HAVE_CRL と HAVE_CRL_UPDATE_CB が定義されており、CRL が追加されるときに同じ発行者でより低い CRL 番号を持つエントリが存在する場合、既存のエントリと、それを置き換える新しいエントリの詳細と共に CbUpdateCRL が呼び出されます。

Parameters:

- **cm** 証明書の情報を保持する WOLFSSL_CERT_MANAGER 構造体。
- **cb** WOLFSSL_CERT_MANAGER の cbUpdateCRL メンバに設定される (CbUpdateCRL) への関数ポインタ。シグネチャ要件: `void (CbUpdateCRL)(CrlInfo old, CrlInfo new);`

See: CbUpdateCRL

Return:

- SSL_SUCCESS 関数とサブルーチンが正常に実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。

Example

```
#include <wolfssl/ssl.h>
```

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
void cb(CrlInfo *old, CrlInfo *new){
    // 関数本体。
}
...
CbUpdateCRL cb = CbUpdateCRL;
...
if(ctx){
    return wolfSSL_CertManagerSetCRLUpdate_Cb(SSL_CM(ssl), cb);
}
```

A.1.2.18 function wolfSSL_CertManagerGetCRLInfo

```
int wolfSSL_CertManagerGetCRLInfo(
    WOLFSSL_CERT_MANAGER * cm,
    CrlInfo * info,
    const byte * buff,
    long sz,
    int type
)
```

この関数は、エンコードされた CRL バッファから解析された CRL 情報を含む構造体を生成します。

Parameters:

- **cm** WOLFSSL_CERT_MANAGER 構造体。
- **info** CRL 情報を受け取る呼び出し元管理の CrlInfo 構造体へのポインタ。
- **buff** エンコードされた CRL を含む入力バッファ。
- **sz** buff 内の入力 CRL データの長さ (バイト単位)。
- **type** WOLFSSL_FILETYPE_PEM または WOLFSSL_FILETYPE_DER
- **cm** `wolfSSL_CertManagerNew()` を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See:

- CbUpdateCRL
- `wolfSSL_SetCRL_Cb`
- `wolfSSL_CertManagerLoadCRL`

Return:

- SSL_SUCCESS 関数とサブルーチンが正常に実行された場合に返されます。

- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。
- SSL_SUCCESS 関数とサブルーチンが正常に実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。

Example

```
#include <wolfssl/ssl.h>
```

```
CrlInfo info;
WOLFSSL_CERT_MANAGER* cm = NULL;
```

```
cm = wolfSSL_CertManagerNew();
```

```
// ファイルからバッファへ crl データを読み込む
```

```
wolfSSL_CertManagerGetCRLInfo(cm, &info, crlData, crlDataLen,
                               WOLFSSL_FILETYPE_PEM);
```

この関数は、証明書マネージャに保存されている CRL を解放します。アプリケーションは、wolfSSL_CertManagerFreeCRL を呼び出してから新しい CRL を読み込むことで、CRL を更新できます。

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);
```

A.1.2.19 function wolfSSL_CertManagerCheckOCSP

```
int wolfSSL_CertManagerCheckOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * der,
    int sz
)
```

この関数は、WOLFSSL_CERT_MANAGER のメンバである ocsEnabled を有効にして、OCSP チェックオプションが有効になっていることを示します。

Parameters:

- **cm** wolfSSL_CertManagerNew() を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **der** 証明書への byte ポインタ。
- **sz** DER 証明書のサイズを表す int 型。

See:

- ParseCertRelative
- CheckCertOCSP

Return:

- SSL_SUCCESS 関数の実行が成功した場合に返されます。WOLFSSL_CERT_MANAGER の ocsEnabled メンバが有効になります。

- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合、またはサブルーチンに許可されていない引数値が渡された場合に返されます。
- MEMORY_E この関数またはサブルーチン内でメモリの割り当てにエラーがある場合に返されます。

Example

```
#import <wolfssl/ssl.h>

WOLFSSL* ssl = wolfSSL_new(ctx);
byte* der;
int sz; // der のサイズ
...
if(wolfSSL_CertManagerCheckOCSP(cm, der, sz) != SSL_SUCCESS){
    // 失敗ケース。
}
```

A.1.2.20 function wolfSSL_CertManagerEnableOCSP

```
int wolfSSL_CertManagerEnableOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

OCSP がオフになっている場合にオンにし、設定オプションでコンパイルされている場合に有効にします。

Parameters:

- **cm** [wolfSSL_CertManagerNew\(\)](#)を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **options** WOLFSSL_CERT_MANAGER 構造体の値を設定するために使用されます。

See: [wolfSSL_CertManagerNew](#)

Return:

- SSL_SUCCESS 関数呼び出しが成功した場合に返されます。
- BAD_FUNC_ARG cm 構造体が NULL の場合。
- MEMORY_E WOLFSSL_OCSP 構造体の値が NULL の場合。
- SSL_FAILURE WOLFSSL_OCSP 構造体の初期化が失敗した場合。
- NOT_COMPILED_IN 正しい機能を有効にしてコンパイルされていないビルド。

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){
    // 失敗ケース。
}
```

A.1.2.21 function wolfSSL_CertManagerDisableOCSP

```
int wolfSSL_CertManagerDisableOCSP(
    WOLFSSL_CERT_MANAGER * cm
)
```

OCSP 証明書失効を無効にします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_DisableCRL`

Return:

- `SSL_SUCCESS` `wolfSSL_CertMangerDisableCRL` が `WOLFSSL_CERT_MANAGER` 構造体の `crlEnabled` メンバを正常に無効にしました。
- `BAD_FUNC_ARG` WOLFSSL 構造体が `NULL` でした。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CertManagerDisableOCSP(ssl) != SSL_SUCCESS){
    // 失敗ケース。
}
```

A.1.2.22 function `wolfSSL_CertManagerSetOCSPOverrideURL`

```
int wolfSSL_CertManagerSetOCSPOverrideURL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * url
)
```

この関数は、URL を `WOLFSSL_CERT_MANAGER` 構造体の `ocspOverrideURL` メンバにコピーします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `ocspOverrideURL`
- `wolfSSL_SetOCSP_OverrideURL`

Return:

- `SSL_SUCCESS` 関数が期待通りに実行できた場合。
- `BAD_FUNC_ARG` `WOLFSSL_CERT_MANAGER` 構造体が `NULL` の場合。
- `MEMEORY_E` 証明書マネージャの `ocspOverrideURL` メンバにメモリを割り当てることができなかった場合。

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){
    // 失敗ケース。
}
```

A.1.2.23 function wolfSSL_CertManagerSetOCSP_Cb

```
int wolfSSL_CertManagerSetOCSP_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

この関数は、WOLFSSL_CERT_MANAGER 内の OCSP コールバックを設定します。

Parameters:

- **cm** WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **ioCb** CbOCSPIO 型の関数ポインタ。
- **respFreeCb** CbOCSPRespFree 型の関数ポインタ。
- **ioCbCtx** I/O コールバックユーザ登録コンテキストへの void ポインタ変数。

See:

- [wolfSSL_CertManagerSetOCSPOverrideURL](#)
- [wolfSSL_CertManagerCheckOCSP](#)
- [wolfSSL_CertManagerEnableOCSPStapling](#)
- [wolfSSL_EnableOCSP](#)
- [wolfSSL_DisableOCSP](#)
- [wolfSSL_SetOCSP_Cb](#)

Return:

- SSL_SUCCESS 実行が成功した場合に返されます。引数は WOLFSSL_CERT_MANAGER 構造体に保存されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER が NULL の場合に返されます。

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);
}
```

A.1.2.24 function wolfSSL_CertManagerEnableOCSPStapling

```
int wolfSSL_CertManagerEnableOCSPStapling(
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は、OCSP ステープリングがオンになっていない場合にオンにし、オプションを設定します。

Parameters:

- **cm** WOLFSSL_CTX 構造体のメンバである WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: [wolfSSL_CTX_EnableOCSPStapling](#)

Return:

- SSL_SUCCESS エラーがなく、関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合、またはサブルーチンに許可されていない引数値が渡された場合に返されます。

- MEMORY_E メモリの割り当てに問題があった場合に返されます。
- SSL_FAILURE_OCSP 構造体の初期化が失敗した場合に返されます。
- NOT_COMPILED_IN wolfSSL が HAVE_CERTIFICATE_STATUS_REQUEST オプションでコンパイルされていない場合に返されます。

Example

```
int wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX* ctx){
    ...
    return wolfSSL_CertManagerEnableOCSPStapling(ctx->cm);
}
```

A.2 メモリ処理

A.2.1 Functions

	Name
void *	wolfSSL_Malloc(size_t size, void * heap, int type) この関数は malloc() に似ていますが、wolfSSL が使用するように設定されたメモリ割り当て関数を呼び出します。デフォルトでは、wolfSSL は malloc() を使用します。これは wolfSSL メモリ抽象化レイヤーを使用して変更できます _wolfSSL_SetAllocators() を参照してください。wolfSSL_Malloc は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ XMALLOC によって呼び出されることに注意してください。デフォルトのビルドでは、size パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、heap と type パラメータが含まれます。
void	wolfSSL_Free(void * ptr, void * heap, int type) この関数は free() に似ていますが、wolfSSL が使用するように設定されたメモリ解放関数を呼び出します。デフォルトでは、wolfSSL は free() を使用します。これは wolfSSL メモリ抽象化レイヤーを使用して変更できます _wolfSSL_SetAllocators() を参照してください。wolfSSL_Free は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ XFREE によって呼び出されることに注意してください。デフォルトのビルドでは、ptr パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、heap と type パラメータが含まれます。

	Name
void *	wolfSSL_Realloc(void * ptr, size_t size, void * heap, int type) この関数は realloc() に似ていますが、wolfSSL が使用するよう設定されたメモリ再割り当て関数を呼び出します。デフォルトでは、wolfSSL は realloc() を使用します。これは wolfSSL メモリ抽象化レイヤーを使用して変更できます _wolfSSL_SetAllocators() を参照してください。wolfSSL_Realloc は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ XREALLOC によって呼び出されることに注意してください。デフォルトのビルドでは、size パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、heap と type パラメータが含まれます。
int	wolfSSL_SetAllocators(wolfSSL_Malloc_cb mf, wolfSSL_Free_cb ff, wolfSSL_Realloc_cb rf) この関数は、wolfSSL が使用する割り当て関数を登録します。デフォルトでは、システムがサポートしている場合、malloc/free と realloc が使用されます。この関数を使用すると、ユーザーは実行時に独自のメモリハンドラをインストールできます。
int	wolfSSL_StaticBufferSz(byte * buffer, word32 sz, int flag) この関数は、静的メモリ機能が使用されている場合に利用可能です (-enable-staticmemory)。メモリ「バケット」の最適なバッファサイズを提供します。これにより、パーティション化された後に余分な未使用メモリが残らないようにバッファサイズを計算する方法が提供されます。この関数の非_ex バージョンでは、コンパイル時に設定されたデフォルトのバケットと配布リストが使用されます。返される値が正の場合、使用する計算されたバッファサイズです。
int	wolfSSL_MemoryPaddingSz(void) この関数は、静的メモリ機能が使用されている場合に利用可能です (-enable-staticmemory)。メモリの各パーティションに必要なパディングのサイズを提供します。このパディングサイズは、メモリ管理構造体を含むために必要なサイズと、メモリアライメントのための追加分になります。

	Name
int	wolfSSL_CTX_load_static_memory (WOLFSSL_CTX ** ctx, wolfSSL_method_func method, unsigned char * buf, unsigned int sz, int flag, int max) この関数は、CTX のために静的メモリを確保するために使用されます。確保されたメモリは、CTX の存続期間中および CTX から作成されたすべての SSL オブジェクトに使用されます。NULL の ctx ポインタと wolfSSL_method_func 関数を渡すことにより、CTX 自体の作成も静的メモリを使用します。wolfSSL_method_func は、WOLFSSL_METHOD* (wolfSSL_method_func)(void heap); の関数シグネチャを持ちます。max に 0 を渡すと、設定されていないかのように動作し、最大同時使用制限が適用されません。渡される flag 値は、メモリの使用方法と動作中の動作を決定します。利用可能なフラグは次のとおりです。
int	wolfSSL_CTX_is_static_memory (WOLFSSL_CTX * ctx, WOLFSSL_MEM_STATS * mem_stats) この関数は接続の動作を変更せず、静的メモリ使用に関する情報を収集するためにのみ使用されます。
int	wolfSSL_is_static_memory (WOLFSSL * ssl, WOLFSSL_MEM_CONN_STATS * mem_stats)wolfSSL_is_static_memory は、SSL の静的メモリ使用に関する情報を収集するために使用されます。戻り値は、静的メモリが使用されているかどうかを示し、WOLFSSL_MEM_CONN_STATS は、静的メモリをロードする際に親 CTX に WOLFMEM_TRACK_STATS フラグが渡された場合にのみ入力されます。
int	wc_LoadStaticMemory (WOLFSSL_HEAP_HINT ** pHint, unsigned char * buf, unsigned int sz, int flag, int max) この関数は、wolfCrypt 使用のために静的メモリを確保するために使用されます。作成されたヒープヒントを関数に渡すことでメモリを使用できます。この例は、wc_InitRng_ex を呼び出すときです。渡される flag 値は、メモリの使用方法と動作中の動作を決定します。一般的に、wolfCrypt 操作は WOLFMEM_GENERAL プールからメモリを使用します。利用可能なフラグは次のとおりです。
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) この関数は、カスタムバケットサイズと配分を使用して wolfCrypt 使用のために静的メモリを確保するために使用されます。作成されたヒープヒントを関数に渡すことでメモリを使用できます。この拡張バージョンでは、デフォルトの事前定義されたサイズを使用する代わりに、カスタムバケットサイズと配分を使用できます。

	Name
WOLFSSL_HEAP_HINT *	wolfSSL_SetGlobalHeapHint (WOLFSSL_HEAP_HINT * hint) この関数は、NULL ヒープヒントがメモリ割り当て関数に渡されたときに使用されるグローバルヒープヒントを設定します。これにより、アプリケーション全体で使用されるデフォルトのヒープヒントを設定できます。
WOLFSSL_HEAP_HINT *	wolfSSL_GetGlobalHeapHint (void) この関数は、NULL ヒープヒントがメモリ割り当て関数に渡されたときに使用される現在のグローバルヒープヒントを取得します。
void	wolfSSL_SetDebugMemoryCb (DebugMemoryCb cb) この関数は、静的メモリ割り当て追跡用のデバッグコールバック関数を設定します。WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACKビルドオプションと共に使用されます。コールバック関数は、メモリ割り当ておよび割り当て解除操作中に呼び出され、デバッグ情報を提供します。
void	wc_UnloadStaticMemory (WOLFSSL_HEAP_HINT * heap) この関数は、静的メモリヒープと関連するミューテックスを解放します。静的メモリ割り当ての使用が完了したときに、リソースを適切にクリーンアップするために呼び出す必要があります。
int	wolfSSL_StaticBufferSz_ex (unsigned int listSz, const word32 * sizeList, const word32 * distList, byte * buffer, word32 sz, int flag) この関数は、カスタムバケットサイズと配分を使用した静的メモリ割り当てに必要なバッファサイズを計算します。この拡張バージョンでは、デフォルトの事前定義されたサイズを使用する代わりに、カスタムバケットサイズを使用できます。

	Name
void *	XMALLOC(size_t n, void * heap, int type) これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: 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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: #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)) これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えます: #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) これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: 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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: <pre>#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))</pre> これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えます: <pre>#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))</pre>

	Name
void	XFREE(void * p, void * heap, int type) これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: 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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: #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)) これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えられます: #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))

A.2.2 Functions Documentation

A.2.2.1 function wolfSSL_Malloc

```
void * wolfSSL_Malloc(
    size_t size,
    void * heap,
    int type
)
```

この関数は malloc() に似ていますが、wolfSSL が使用するよう設定されたメモリ割り当て関数を呼び出します。デフォルトでは、wolfSSL は malloc() を使用します。これは wolfSSL メモリ抽象化レイヤーを使用して変更できます - wolfSSL_SetAllocators() を参照してください。wolfSSL_Malloc は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ XMALLOC によって呼び出されることに注意してください。デフォルトのビルドでは、size パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、heap と type パラメータが含まれます。

Parameters:

- **size** 割り当てるメモリのバイト単位のサイズ。
- **heap** メモリに使用するヒープヒント。NULL にできます。
- **type** 動的タイプ (types.h の DYNAMIC_TYPE_ リストを参照)。

See:

- `wolfSSL_Free`
- `wolfSSL_Realloc`
- `wolfSSL_SetAllocators`
- `XMALLOC`
- `XFREE`
- `XREALLOC`

Return:

- `pointer` 成功した場合、この関数は割り当てられたメモリへのポインタを返します。
- `error` エラーがある場合、`NULL` が返されます。

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
```

A.2.2.2 function wolfSSL_Free

```
void wolfSSL_Free(
    void * ptr,
    void * heap,
    int type
)
```

この関数は `free()` に似ていますが、wolfSSL が使用するよう設定されたメモリ解放関数を呼び出します。デフォルトでは、wolfSSL は `free()` を使用します。これは wolfSSL メモリ抽象化レイヤーを使用して変更できます - `wolfSSL_SetAllocators()` を参照してください。wolfSSL_Free は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ `XFREE` によって呼び出されることに注意してください。デフォルトのビルドでは、`ptr` パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、`heap` と `type` パラメータが含まれます。

Parameters:

- **ptr** 解放するメモリへのポインタ。
- **heap** メモリに使用するヒープヒント。NULL にできます。
- **type** 動的タイプ (`types.h` の `DYNAMIC_TYPE_` リストを参照)。

See:

- `wolfSSL_Alloc`
- `wolfSSL_Realloc`
- `wolfSSL_SetAllocators`
- `XMALLOC`
- `XFREE`
- `XREALLOC`

Return: none 戻り値なし。

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
// 必要に応じてデータを処理
...
if(tenInts) {
    wolfSSL_Free(tenInts, NULL, DYNAMIC_TYPE_TMP_BUFFER);
}
```

A.2.2.3 function wolfSSL_Realloc

```
void * wolfSSL_Realloc(
    void * ptr,
```

```

    size_t size,
    void * heap,
    int type
)

```

この関数は `realloc()` に似ていますが、wolfSSL が使用するよう設定されたメモリ再割り当て関数を呼び出します。デフォルトでは、wolfSSL は `realloc()` を使用します。これは wolfSSL メモリ抽象化レイヤーを使用して変更できます - `wolfSSL_SetAllocators()` を参照してください。wolfSSL_Realloc は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ `XREALLOC` によって呼び出されることに注意してください。デフォルトのビルドでは、size パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、heap と type パラメータが含まれます。

Parameters:

- **ptr** 再割り当てする、以前に割り当てられたメモリへのポインタ。
- **size** 割り当てるバイト数。
- **heap** メモリに使用するヒープヒント。NULL にできます。
- **type** 動的タイプ (types.h の DYNAMIC_TYPE_ リストを参照)。

See:

- [wolfSSL_Free](#)
- [wolfSSL_Malloc](#)
- [wolfSSL_SetAllocators](#)
- [XMALLOC](#)
- [XFREE](#)
- [XREALLOC](#)

Return:

- pointer 成功した場合、この関数は再割り当てされたメモリへのポインタを返します。これは ptr と同じポインタである場合もあれば、新しいポインタの場所である場合もあります。
- Null エラーがある場合、NULL が返されます。

Example

```

int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
int* twentyInts = (int*)wolfSSL_Realloc(tenInts, sizeof(int)*20);

```

A.2.2.4 function wolfSSL_SetAllocators

```

int wolfSSL_SetAllocators(
    wolfSSL_Malloc_cb mf,
    wolfSSL_Free_cb ff,
    wolfSSL_Realloc_cb rf
)

```

この関数は、wolfSSL が使用する割り当て関数を登録します。デフォルトでは、システムがサポートしている場合、malloc/free と realloc が使用されます。この関数を使用すると、ユーザーは実行時に独自のメモリハンドラをインストールできます。

Parameters:

- **malloc_function** wolfSSL が使用するメモリ割り当て関数。関数シグネチャは上記の wolfSSL_Malloc_cb プロトタイプと一致する必要があります。
- **free_function** wolfSSL が使用するメモリ解放関数。関数シグネチャは上記の wolfSSL_Free_cb プロトタイプと一致する必要があります。
- **realloc_function** wolfSSL が使用するメモリ再割り当て関数。関数シグネチャは上記の wolfSSL_Realloc_cb プロトタイプと一致する必要があります。

See: none

Return:

- Success 成功した場合、この関数は 0 を返します。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラー。

Example

```
static void* MyMalloc(size_t size)
{
    // カスタム malloc 関数
}

static void MyFree(void* ptr)
{
    // カスタム free 関数
}

static void* MyRealloc(void* ptr, size_t size)
{
    // カスタム realloc 関数
}

// カスタムメモリ関数を wolfSSL に登録
int ret = wolfSSL_SetAllocators(MyMalloc, MyFree, MyRealloc);
if (ret != 0) {
    // メモリ関数の設定に失敗
}
```

A.2.2.5 function wolfSSL_StaticBufferSz

```
int wolfSSL_StaticBufferSz(
    byte * buffer,
    word32 sz,
    int flag
)
```

この関数は、静的メモリ機能が使用されている場合に利用可能です (-enable-staticmemory)。メモリ「バケット」の最適なバッファサイズを提供します。これにより、パーティション化された後に余分な未使用メモリが残らないようにバッファサイズを計算する方法が提供されます。この関数の非_ex バージョンでは、コンパイル時に設定されたデフォルトのバケットと配布リストが使用されます。返される値が正の場合、使用する計算されたバッファサイズです。

Parameters:

- **buffer** バッファへのポインタ。
- **size** バッファのサイズ。
- **type** 希望するメモリタイプ、つまり WOLFMEM_GENERAL または WOLFMEM_IO_POOL。

See:

- [wolfSSL_Malloc](#)
- [wolfSSL_Free](#)

Return:

- Success バッファサイズの計算が正常に完了すると、正の値が返されます。この返される値は最適なバッファサイズです。
- Failure すべての負の値はエラーケースと見なされます。

Example

```

byte buffer[1000];
word32 size = sizeof(buffer);
int optimum;

optimum = wolfSSL_StaticBufferSz(buffer, size, WOLFMEM_GENERAL);
if (optimum < 0) { //エラーケースを処理 }
printf(" すべてのメモリを利用するための最適なバッファサイズは %d です\n",
optimum);
...

```

A.2.2.6 function wolfSSL_MemoryPaddingSz

```

int wolfSSL_MemoryPaddingSz(
    void
)

```

この関数は、静的メモリ機能が使用されている場合に利用可能です（-enable-staticmemory）。メモリの各パーティションに必要なパディングのサイズを提供します。このパディングサイズは、メモリ管理構造体を含むために必要なサイズと、メモリアライメントのための追加分になります。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_Malloc](#)
- [wolfSSL_Free](#)

Return:

- メモリパディングの計算が成功すると、戻り値は正の値になります。
- すべての負の値はエラーケースと見なされます。

Example

```

int padding;
padding = wolfSSL_MemoryPaddingSz();
if (padding < 0) { //エラーケースを処理 }
printf(" メモリの各「バケット」に必要なパディングサイズは %d です\n",
padding);
// IO POOL サイズのバッファの計算は、バケット数
// × (padding + WOLFMEM_IO_SZ)
...

```

A.2.2.7 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
)

```

この関数は、CTX のために静的メモリを確保するために使用されます。確保されたメモリは、CTX の存続期間中および CTX から作成されたすべての SSL オブジェクトに使用されます。NULL の ctx ポ

インタと wolfSSL_method_func 関数を渡すことにより、CTX 自体の作成も静的メモリを使用します。wolfSSL_method_func は、WOLFSSL_METHOD* (wolfSSL_method_func)(void heap); の関数シグネチャを持ちます。max に 0 を渡すと、設定されていないかのように動作し、最大同時使用制限が適用されません。渡される flag 値は、メモリの使用方法と動作中の動作を決定します。利用可能なフラグは次のとおりです。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタのアドレス。
- **method** プロトコルを作成する関数。(ctx も NULL でない場合は NULL である必要があります)
- **buf** すべての操作に使用するメモリ。
- **sz** 渡されるメモリバッファのサイズ。
- **flag** メモリのタイプ。
- **max** 最大同時操作数。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_is_static_memory](#)
- [wolfSSL_is_static_memory](#)

Return:

- 成功した場合、SSL_SUCCESS が返されます。
- すべての失敗した戻り値は 0 未満または SSL_FAILURE と等しくなります。

0 - デフォルトの一般メモリ

WOLFMEM_IO_POOL - メッセージの送受信時の入出力バッファに使用されます。一般メモリをオーバーライドするため、渡されたバッファ内のすべてのメモリが IO に使用されます。WOLFMEM_IO_FIXED - WOLFMEM_IO_POOL と同じですが、各 SSL は存続期間中に 2 つのバッファを保持します。WOLFMEM_TRACK_STATS - 各 SSL は実行中にメモリ統計を追跡します。

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;
...
// 静的メモリを使用して ctx も作成、使用する一般メモリから開始
ctx = NULL;
ret = wolfSSL_CTX_load_static_memory(&ctx, wolfSSLv23_server_method_ex,
    ↪ memory, memorySz, 0,
    MAX_CONCURRENT_HANDSHAKES);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}
// IO で使用するメモリをロード
ret = wolfSSL_CTX_load_static_memory(&ctx, NULL, IO, IOSz, flag,
    ↪ MAX_CONCURRENT_IO);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}
...
```


A.2.2.8 function wolfSSL_CTX_is_static_memory

```
int wolfSSL_CTX_is_static_memory(
    WOLFSSL_CTX * ctx,
    WOLFSSL_MEM_STATS * mem_stats
)
```

この関数は接続の動作を変更せず、静的メモリ使用に関する情報を収集するためにのみ使用されます。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **mem_stats** 静的メモリ使用に関する情報を保持する構造体。

See:

- `wolfSSL_CTX_new`
- `wolfSSL_CTX_load_static_memory`
- `wolfSSL_is_static_memory`

Return:

- CTX に静的メモリを使用している場合は 1 の値が返されます。
- 静的メモリを使用していない場合は 0 が返されます。

Example

```
WOLFSSL_CTX* ctx;
int ret;
WOLFSSL_MEM_STATS mem_stats;
...
//CTX での静的メモリに関する情報を取得

ret = wolfSSL_CTX_is_static_memory(ctx, &mem_stats);

if (ret == 1) {
    // 静的メモリを使用しているケースを処理
    // mem_stats の要素を出力または検査
}

if (ret == 0) {
    //ctx が静的メモリを使用していないケースを処理
}
...
```

A.2.2.9 function wolfSSL_is_static_memory

```
int wolfSSL_is_static_memory(
    WOLFSSL * ssl,
    WOLFSSL_MEM_CONN_STATS * mem_stats
)
```

`wolfSSL_is_static_memory` は、SSL の静的メモリ使用に関する情報を収集するために使用されます。戻り値は、静的メモリが使用されているかどうかを示し、WOLFSSL_MEM_CONN_STATS は、静的メモリをロードする際に親 CTX に WOLFMEM_TRACK_STATS フラグが渡された場合にのみ入力されます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **mem_stats** 静的メモリ使用を含む構造体。

See:

- `wolfSSL_new`
- `wolfSSL_CTX_is_static_memory`

Return:

- CTX に静的メモリを使用している場合は 1 の値が返されます。
- 静的メモリを使用していない場合は 0 が返されます。

Example

```
WOLFSSL* ssl;
int ret;
WOLFSSL_MEM_CONN_STATS mem_stats;

...

ret = wolfSSL_is_static_memory(ssl, mem_stats);

if (ret == 1) {
    // 静的メモリの場合のケースを処理
    // WOLFMEM_TRACK_STATS フラグがある場合は mem_stats の要素を調査
}

...
```

A.2.2.10 function wc_LoadStaticMemory

```
int wc_LoadStaticMemory(
    WOLFSSL_HEAP_HINT ** pHint,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

この関数は、wolfCrypt 使用のために静的メモリを確保するために使用されます。作成されたヒープヒントを関数に渡すことでメモリを使用できます。この例は、`wc_InitRng_ex` を呼び出すときです。渡される `flag` 値は、メモリの使用方法と動作中の動作を決定します。一般的に、wolfCrypt 操作は `WOLFMEM_GENERAL` プールからメモリを使用します。利用可能なフラグは次のとおりです。

Parameters:

- **phint** 使用する `WOLFSSL_HEAP_HINT` 構造体。
- **buf** すべての操作に使用するメモリ。
- **sz** 渡されるメモリバッファのサイズ。
- **flag** メモリのタイプ。
- **max** 最大同時操作数（ハンドシェイク、IO）。

See: none

Return: none この関数は値を返しません。

`WOLFMEM_GENERAL` - デフォルトの一般メモリ

`WOLFMEM_IO_POOL` - メッセージの送受信時の入出力バッファに使用されます。一般メモリをオーバーライドするため、渡されたバッファ内のすべてのメモリが IO に使用されます。`WOLFMEM_IO_FIXED` - `WOLFMEM_IO_POOL` と同じですが、各 SSL は存続期間中に 2 つのバッファを保持します。`WOLFMEM_TRACK_STATS` - 各 SSL は実行中にメモリ統計を追跡します。

Example

```

WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
int flag = WOLFMEM_GENERAL | WOLFMEM_TRACK_STATS;
...

// 使用するメモリをロード

ret = wc_LoadStaticMemory(&hint, memory, memorySz, flag, 0);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}
...

ret = wc_InitRng_ex(&rng, hint, 0);

// ret 値をチェック

```

A.2.2.11 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
)

```

この関数は、カスタムバケットサイズと配分を使用して wolfCrypt 使用のために静的メモリを確保するために使用されます。作成されたヒープヒントを関数に渡すことでメモリを使用できます。この拡張バージョンでは、デフォルトの事前定義されたサイズを使用する代わりに、カスタムバケットサイズと配分を使用できます。

Parameters:

- **pHint** 初期化する WOLFSSL_HEAP_HINT ハンドル。
- **listSz** サイズリストと分布リストのエントリ数。
- **sizeList** 使用するバケットサイズの配列。
- **distList** sizeList に対応する分布リスト。
- **buf** すべての操作に使用するメモリ。
- **sz** 渡されるメモリバッファのサイズ。
- **flag** メモリのタイプ。
- **max** 最大同時操作数（ハンドシェイク、IO）。

See:

- [wc_LoadStaticMemory](#)
- [wc_UnloadStaticMemory](#)

Return: none この関数は値を返しません。

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]));
...

// カスタムバケットサイズで使用するメモリをロード

ret = wc_LoadStaticMemory_ex(&hint, listSz, sizeList, distList,
                             memory, memorySz, flag, 0);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}
...

ret = wc_InitRng_ex(&rng, hint, 0);

// ret 値をチェック

```

A.2.2.12 function wolfSSL_SetGlobalHeapHint

```

WOLFSSL_HEAP_HINT * wolfSSL_SetGlobalHeapHint(
    WOLFSSL_HEAP_HINT * hint
)

```

この関数は、NULL ヒープヒントがメモリ割り当て関数に渡されたときに使用されるグローバルヒープヒントを設定します。これにより、アプリケーション全体で使用するデフォルトのヒープヒントを設定できます。

Parameters:

- **hint** グローバルヒープヒントとして使用する WOLFSSL_HEAP_HINT 構造体。

See:

- [wolfSSL_GetGlobalHeapHint](#)
- [wc_LoadStaticMemory](#)

Return: 設定されていた以前のグローバルヒープヒントを返します。

Example

```

WOLFSSL_HEAP_HINT hint;
WOLFSSL_HEAP_HINT* prev_hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
...

// 使用するメモリをロード
ret = wc_LoadStaticMemory(&hint, memory, memorySz, WOLFMEM_GENERAL, 0);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}

```

```
// グローバルヒープヒントとして設定
prev_hint = wolfSSL_SetGlobalHeapHint(&hint);
if (prev_hint != NULL) {
    // 以前のグローバルヒープヒントがありました
}
```

A.2.2.13 function wolfSSL_GetGlobalHeapHint

```
WOLFSSL_HEAP_HINT * wolfSSL_GetGlobalHeapHint(
    void
)
```

この関数は、NULL ヒープヒントがメモリ割り当て関数に渡されたときに使用される現在のグローバルヒープヒントを取得します。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_SetGlobalHeapHint](#)
- [wc_LoadStaticMemory](#)

Return: 現在のグローバルヒープヒントを返します。設定されていない場合は NULL を返します。

Example

```
WOLFSSL_HEAP_HINT* current_hint;
...

current_hint = wolfSSL_GetGlobalHeapHint();
if (current_hint != NULL) {
    // グローバルヒープヒントが設定されています
    // current_hint を操作に使用できます
}
```

A.2.2.14 function wolfSSL_SetDebugMemoryCb

```
void wolfSSL_SetDebugMemoryCb(
    DebugMemoryCb cb
)
```

この関数は、静的メモリ割り当て追跡用のデバッグコールバック関数を設定します。WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACK ビルドオプションと共に使用されます。コールバック関数は、メモリ割り当ておよび割り当て解除操作中に呼び出され、デバッグ情報を提供します。

Parameters:

- **cb** 設定するデバッグコールバック関数。

See: none

Return:

- 成功した場合、0 が返されます。
- すべての失敗した戻り値は 0 未満になります。

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);
}
...

// デバッグコールバックを設定
int ret = wolfSSL_SetDebugMemoryCb(debug_memory_cb);
if (ret != 0) {
    // エラーケースを処理
}

```

A.2.2.15 function wc_UnloadStaticMemory

```

void wc_UnloadStaticMemory(
    WOLFSSL_HEAP_HINT * heap
)

```

この関数は、静的メモリヒープと関連するミューテックスを解放します。静的メモリ割り当ての使用が完了したときに、リソースを適切にクリーンアップするために呼び出す必要があります。

Parameters:

- **hint** アンロードする WOLFSSL_HEAP_HINT 構造体。

See:

- [wc_LoadStaticMemory](#)
- [wc_LoadStaticMemory_ex](#)

Return:

- 成功した場合、0 が返されます。
- すべての失敗した戻り値は 0 未満になります。

Example

```

WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
...

// 使用するメモリをロード
ret = wc_LoadStaticMemory(&hint, memory, memorySz, WOLFMEM_GENERAL, 0);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}

// 操作にメモリを使用
...

// 完了時にクリーンアップ
wc_UnloadStaticMemory(&hint);

```

A.2.2.16 function wolfSSL_StaticBufferSz_ex

```

int wolfSSL_StaticBufferSz_ex(
    unsigned int listSz,

```

```

    const word32 * sizeList,
    const word32 * distList,
    byte * buffer,
    word32 sz,
    int flag
)

```

この関数は、カスタムバケットサイズと配分を使用した静的メモリ割り当てに必要なバッファサイズを計算します。この拡張バージョンでは、デフォルトの事前定義されたサイズを使用する代わりに、カスタムバケットサイズを使用できます。

Parameters:

- **bucket_sizes** 使用するバケットサイズの配列。
- **bucket_count** 配列内のバケットサイズの数。
- **flag** 希望するメモリタイプ、つまり WOLFMEM_GENERAL または WOLFMEM_IO_POOL。

See:

- [wolfSSL_StaticBufferSz](#)
- [wc_LoadStaticMemory_ex](#)

Return:

- バッファサイズの計算が正常に完了すると、正の値が返されます。
- すべての負の値はエラーケースと見なされます。

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) { //エラーケースを処理 }
printf(" カスタムバケットでの最適なバッファサイズは %d です\n", optimum);
...

```

A.2.2.17 function XMALLOC

```

void * XMALLOC(
    size_t n,
    void * heap,
    int type
)

```

これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: 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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: #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)) これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えます: #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** 割り当てるメモリのサイズ
- **h** (カスタム XMALLOC 関数で使用) 使用するヒープへのポインタ
- **t** ユーザーヒント用のメモリ割り当てタイプ。types.h の列挙型を参照

See:

- [wolfSSL_Malloc](#)
- [wolfSSL_Realloc](#)
- [wolfSSL_Free](#)
- [wolfSSL_SetAllocators](#)

Return:

- pointer 成功時に割り当てられたメモリへのポインタを返します
- NULL 失敗時

Example

```
int* tenInts = XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // スペース割り当てエラー
    return MEMORY_E;
}
```

A.2.2.18 function XREALLOC

```
void * XREALLOC(
    void * p,
    size_t n,
    void * heap,
    int type
)
```

これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: 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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: #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)) これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えます: #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** 再割り当てするアドレスへのポインタ
- **n** 割り当てるメモリのサイズ
- **h** (カスタム XREALLOC 関数で使用) 使用するヒープへのポインタ
- **t** ユーザーヒント用のメモリ割り当てタイプ。types.h の列挙型を参照

See:

- `wolfSSL_Malloc`
- `wolfSSL_Realloc`
- `wolfSSL_Free`
- `wolfSSL_SetAllocators`

Return:

- 成功時に割り当てられたメモリへのポインタを返します
- NULL 失敗時

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);
```

A.2.2.19 function XFREE

```
void XFREE(
    void * p,
    void * heap,
    int type
)
```

これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: 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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: #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)) これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えます: #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** 解放するアドレスへのポインタ
- **h** (カスタム XFREE 関数で使用) 使用するヒープへのポインタ
- **t** ユーザーヒント用のメモリ割り当てタイプ。types.h の列挙型を参照

See:

- `wolfSSL_Malloc`
- `wolfSSL_Realloc`
- `wolfSSL_Free`
- `wolfSSL_SetAllocators`

Return: none 戻り値なし。

Example

```
int* tenInts = XMALLOC(sizeof(int) * 10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // スペース割り当てエラー
    return MEMORY_E;
}
```

A.3 OpenSSL API

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) この関数は次の数学演算を実行します “ $r = (a^p) \% m$ ”。
const WOLFSSL_EVP_CIPHER *	wolfSSL_EVP_des_ede3_ecb (void) それぞれの WOLFSSL_EVP_CIPHER ポインタのゲッター関数。これらの暗号文字列を設定するには、まずプログラムで wolfSSL_EVP_init() を一度呼び出す必要があります。wolfSSL_EVP_des_ede3_ecb() の場合、WOLFSSL_DES_ECB マクロが定義されている必要があります。
const WOLFSSL_EVP_CIPHER *	wolfSSL_EVP_des_cbc (void) それぞれの WOLFSSL_EVP_CIPHER ポインタのゲッター関数。これらの暗号文字列を設定するには、まずプログラムで wolfSSL_EVP_init() を一度呼び出す必要があります。wolfSSL_EVP_des_ecb() の場合、WOLFSSL_DES_ECB マクロが定義されている必要があります。
int	wolfSSL_EVP_DigestInit_ex (WOLFSSL_EVP_MD_CTX * ctx, const WOLFSSL_EVP_MD * type, WOLFSSL_ENGINE * impl) WOLFSSL_EVP_MD_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_DigestInit() のラッパーです。
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) WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_CipherInit() のラッパーです。
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) WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_CipherInit() のラッパーです。暗号化フラグを暗号化に設定します。

	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) WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_CipherInit() のラッパーです。暗号化フラグを復号に設定します。
int	wolfSSL_EVP_CipherUpdate (WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl, const unsigned char * in, int inl) データを暗号化/復号する関数。in バッファが暗号化または復号される対象として追加され、out バッファが結果を保持します。outl は暗号化/復号された情報の長さになります。
int	wolfSSL_EVP_CipherFinal (WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl) この関数は、パディングを追加する最終暗号操作を実行します。WOLFSSL_EVP_CIPHER_CTX 構造体に WOLFSSL_EVP_CIPH_NO_PADDING フラグが設定されている場合、1 が返され、暗号化/復号は実行されません。パディングフラグが設定されている場合、ctx が暗号化に設定されているときにパディングが追加され暗号化され、復号に設定されているときにパディング値がチェックされます。
int	wolfSSL_EVP_CIPHER_CTX_set_key_length (WOLFSSL_EVP_CIPHER_CTX * ctx, int keylen) WOLFSSL_EVP_CIPHER_CTX 構造体の鍵長のセッター関数。
int	wolfSSL_EVP_CIPHER_CTX_block_size (const WOLFSSL_EVP_CIPHER_CTX * ctx) これは ctx ブロックサイズのゲッター関数です。
int	wolfSSL_EVP_CIPHER_block_size (const WOLFSSL_EVP_CIPHER * cipher) これは暗号のブロックサイズのゲッター関数です。
void	wolfSSL_EVP_CIPHER_CTX_set_flags (WOLFSSL_EVP_CIPHER_CTX * ctx, int flags) WOLFSSL_EVP_CIPHER_CTX 構造体のセッター関数。
void	wolfSSL_EVP_CIPHER_CTX_clear_flags (WOLFSSL_EVP_CIPHER_CTX * ctx, int flags) WOLFSSL_EVP_CIPHER_CTX 構造体のクリア関数。
int	wolfSSL_EVP_CIPHER_CTX_set_padding (WOLFSSL_EVP_CIPHER_CTX * c, int pad) パディングを使用するための WOLFSSL_EVP_CIPHER_CTX 構造体のセッター関数。
unsigned long	wolfSSL_EVP_CIPHER_CTX_flags (const WOLFSSL_EVP_CIPHER_CTX * ctx) WOLFSSL_EVP_CIPHER_CTX 構造体のゲッター関数。v1.1.0 で非推奨

	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) この関数は、WOLFSSL_BIO 構造体にキーを PEM 形式で書き込みます。
int	wolfSSL_CTX_use_RSAPrivateKey_file (WOLFSSL_CTX * ctx, const char * file, int format) この関数は、SSL 接続で使用されるプライベート RSA 鍵を SSL コンテキスト (WOLFSSL_CTX) に読み込みます。この関数は、wolfSSL が OpenSSL 互換レイヤーを有効にしてコンパイルされた場合 (-enable_opensslExtra、#define OPENSSL_EXTRA) にのみ利用可能であり、より一般的に使用される wolfSSL_CTX_use_PrivateKey_file() 関数と同一です。file 引数には、format 引数で指定された形式の RSA プライベート鍵ファイルへのポインタが含まれます。
int	wolfSSL_use_certificate_file (WOLFSSL * ssl, const char * file, int format) この関数は、証明書ファイルを SSL セッション (WOLFSSL 構造体) に読み込みます。証明書ファイルは file 引数によって提供されます。format 引数は、ファイルの形式タイプ (SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM) を指定します。
int	wolfSSL_use_PrivateKey_file (WOLFSSL * ssl, const char * file, int format) この関数は、プライベート鍵ファイルを SSL セッション (WOLFSSL 構造体) に読み込みます。鍵ファイルは file 引数によって提供されます。format 引数は、ファイルの形式タイプ (SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM) を指定します。
int	wolfSSL_use_certificate_chain_file (WOLFSSL * ssl, const char * file) この関数は、証明書のチェーンを SSL セッション (WOLFSSL 構造体) にロードします。証明書チェーンを含むファイルは file 引数で指定され、PEM 形式の証明書を含んでいる必要があります。この関数は、MAX_CHAIN_DEPTH(デフォルト = 9、internal.h で定義) 個までの証明書と、サブジェクト証明書を処理します。

	Name
int	wolfSSL_use_RSAPrivateKey_file (WOLFSSL * ssl, const char * file, int format) この関数は、SSL 接続で使用されるプライベート RSA 鍵を SSL セッション (WOLFSSL 構造体) にロードします。この関数は、wolfSSL が OpenSSL 互換レイヤーを有効にしてコンパイルされている場合 (-enable_opensslExtra、#define OPENSSL_EXTRA) にのみ利用可能であり、より一般的に使用される wolfSSL_use_PrivateKey_file() 関数と同一です。file 引数は、format で指定された形式の RSA プライベート鍵ファイルへのポインタを含みます。
long	wolfSSL_set_tlsext_status_type (WOLFSSL * s, int type) この関数は、クライアントアプリケーションがサーバに OSCP ステータスレスポンス (OCSP ステージングとも呼ばれます) を返送するよう要求する際に呼び出されます。現在、サポートされている唯一のタイプは TLSEXT_STATUSTYPE_ocsp です。
WOLFSSL_X509_CHAIN *	wolfSSL_get_peer_chain (WOLFSSL * ssl) ピアの証明書チェーンを取得します。
int	wolfSSL_get_chain_count (WOLFSSL_X509_CHAIN * chain) ピアの証明書チェーン数を取得します。
int	wolfSSL_get_chain_length (WOLFSSL_X509_CHAIN * chain, int idx) インデックス (idx) におけるピアの ASN1.DER 証明書の長さをバイト単位で取得します。
unsigned char *	wolfSSL_get_chain_cert (WOLFSSL_X509_CHAIN * chain, int idx) インデックス (idx) におけるピアの ASN1.DER 証明書を取得します。
int	wolfSSL_get_chain_cert_pem (WOLFSSL_X509_CHAIN * chain, int idx, unsigned char * buf, int inLen, int * outLen) インデックス (idx) におけるピアの PEM 証明書を取得します。
const unsigned char *	wolfSSL_get_sessionID (const WOLFSSL_SESSION * s) セッション ID を取得します。セッション ID は常に 32 バイト長です。
int	wolfSSL_X509_get_serial_number (WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) ピアの証明書シリアル番号を取得します。シリアル番号バッファ (in) は少なくとも 32 バイト長であり、入力として inOutSz 引数として提供される必要があります。関数を呼び出した後、inOutSz には in バッファに書き込まれた実際の長さ (バイト単位) が保持されます。

	Name
WC_PKCS12 *	wolfSSL_d2i_PKCS12_bio (WOLFSSL_BIO * bio, WC_PKCS12 ** pkcs12)wolfSSL_d2i_PKCS12_bio(d2i_PKCS12_bio) は、WOLFSSL_BIO から WC_PKCS12 構造体へ PKCS12 情報をコピーします。情報は、構造体内でコンテンツ情報のリストとして分割され、オプションの MAC 情報を保持する構造体も含まれます。情報が WC_PKCS12 構造体内でチャンク (ただし復号されていない) に分割された後、呼び出すことによって解析および復号できます。
WC_PKCS12 *	wolfSSL_i2d_PKCS12_bio (WOLFSSL_BIO * bio, WC_PKCS12 * pkcs12)wolfSSL_i2d_PKCS12_bio(i2d_PKCS12_bio) は、WC_PKCS12 構造体から WOLFSSL_BIO へ証明書情報をコピーします。
int	wolfSSL_PKCS12_parse (WOLFSSL_X509) ca)PKCS12 は、configure コマンドに <code>-enable_opensslextra</code> を追加することで有効にできます。復号にトリプル DES と RC4 を使用できるため、 <code>opensslextra</code> を有効にする際にこれらの機能も有効にすることを推奨します (<code>-enable_des3 -enable_arc4</code>)。wolfSSL は現在 RC2 をサポートしていないため、RC2 での復号は現在利用できません。これは、OpenSSL コマンドラインで <code>p12</code> ファイルを作成する際に使用されるデフォルトの暗号化スキームで顕著になる可能性があります。 wolfSSL_PKCS12_parse(PKCS12_parse)。この関数が最初に行うことは、存在する場合に MAC が正しいかどうかを確認することです。MAC が失敗した場合、関数は返され、保存されているコンテンツ情報の復号を試みません。この関数は、各コンテンツ情報を解析してバッグタイプを探し、バッグタイプが既知の場合、必要に応じて復号され、構築中の証明書リストまたは見つかった鍵として保存されます。すべてのバッグを解析した後、見つかった鍵は証明書リストと比較され、一致するペアが見つかります。この一致するペアは、鍵と証明書として返されます。オプションで、見つかった証明書リストは <code>STACK_OF</code> の証明書として返されます。現時点では、CRL、Secret、または SafeContents バッグはスキップされ、解析されません。これらまたは他の「不明な」バッグがスキップされているかどうかは、デバッグ出力を表示することで確認できます。フレンドリ名などの追加属性は、PKCS12 ファイルを解析する際にスキップされます。

A.3.2 Functions Documentation

A.3.2.1 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
)

```

この関数は次の数学演算を実行します $r = (a^p) \% m$ 。

Parameters:

- **r** 結果を保持する構造体。
- **a** 累乗される値。
- **p** a を累乗する指数。
- **m** 使用する剰余。
- **ctx** 現在 wolfSSL では使用されていないため、NULL にできます。

See:

- wolfSSL_BN_new
- wolfSSL_BN_free

Return:

- SSL_SUCCESS 数学演算が正常に実行された場合。
- SSL_FAILURE エラーケースが発生した場合。

Example

```

WOLFSSL_BIGNUM r,a,p,m;
int ret;
// big number 値を設定
ret = wolfSSL_BN_mod_exp(r, a, p, m, NULL);
// ret 値を確認

```

A.3.2.2 function wolfSSL_EVP_des_ede3_ecb

```

const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_ede3_ecb(
    void
)

```

それぞれの WOLFSSL_EVP_CIPHER ポインタのゲッター関数。これらの暗号文字列を設定するには、まずプログラムで wolfSSL_EVP_init() を一度呼び出す必要があります。wolfSSL_EVP_des_ede3_ecb() の場合、WOLFSSL_DES_ECB マクロが定義されている必要があります。

Parameters:

- **none** パラメータなし。

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer DES EDE3 操作の WOLFSSL_EVP_CIPHER ポインタを返します。

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()));

```


A.3.2.3 function wolfSSL_EVP_des_cbc

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_cbc(
    void
)
```

それぞれの WOLFSSL_EVP_CIPHER ポインタのゲッター関数。これらの暗号文字列を設定するには、まずプログラムで wolfSSL_EVP_init() を一度呼び出す必要があります。wolfSSL_EVP_des_ecb() の場合、WOLFSSL_DES_ECB マクロが定義されている必要があります。

Parameters:

- **none** パラメータなし。

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer DES 操作用の WOLFSSL_EVP_CIPHER ポインタを返します。

Example

```
WOLFSSL_EVP_CIPHER* cipher;
cipher = wolfSSL_EVP_des_cbc();
...
```

A.3.2.4 function wolfSSL_EVP_DigestInit_ex

```
int wolfSSL_EVP_DigestInit_ex(
    WOLFSSL_EVP_MD_CTX * ctx,
    const WOLFSSL_EVP_MD * type,
    WOLFSSL_ENGINE * impl
)
```

WOLFSSL_EVP_MD_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_DigestInit() のラッパーです。

Parameters:

- **ctx** 初期化する構造体。
- **type** 実行するハッシュのタイプ、例えば SHA。
- **impl** 使用するエンジン。wolfSSL では該当なし、NULL でも可。

See:

- wolfSSL_EVP_MD_CTX_new
- [wolfCrypt_Init](#)
- wolfSSL_EVP_MD_CTX_free

Return:

- SSL_SUCCESS 正常に設定された場合。
- SSL_FAILURE 成功しなかった場合。

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));
//リソースを解放
```

A.3.2.5 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
)
```

WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_CipherInit() のラッパーです。

Parameters:

- **ctx** 初期化する構造体。
- **type** 実行する暗号化/復号のタイプ、例えば AES。
- **impl** 使用するエンジン。wolfSSL では該当なし、NULL でも可。
- **key** 設定する鍵。
- **iv** アルゴリズムが必要とする場合の iv。
- **enc** 暗号化 (1) または復号 (0) フラグ。

See:

- wolfSSL_EVP_CIPHER_CTX_new
- [wolfCrypt_Init](#)
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定された場合。
- SSL_FAILURE 成功しなかった場合。

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_cbc(), e, key, iv, 1));
// リソースを解放
```

A.3.2.6 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
)

```

WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_CipherInit() のラッパーです。暗号化フラグを暗号化に設定します。

Parameters:

- **ctx** 初期化する構造体。
- **type** 実行する暗号化のタイプ、例えば AES。
- **impl** 使用するエンジン。wolfSSL では該当なし、NULL でも可。
- **key** 使用する鍵。
- **iv** 使用する iv。

See:

- wolfSSL_EVP_CIPHER_CTX_new
- [wolfCrypt_Init](#)
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定された場合。
- SSL_FAILURE 成功しなかった場合。

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));
//リソースを解放

```

A.3.2.7 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
)

```

WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_CipherInit() のラッパーです。暗号化フラグを復号に設定します。

Parameters:

- **ctx** 初期化する構造体。
- **type** 実行する暗号化/復号のタイプ、例えば AES。
- **impl** 使用するエンジン。wolfSSL では該当なし、NULL でも可。

- **key** 設定する鍵。
- **iv** アルゴリズムが必要とする場合の iv。
- **enc** 暗号化 (1) または復号 (0) フラグ。

See:

- wolfSSL_EVP_CIPHER_CTX_new
- **wolfCrypt_Init**
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定された場合。
- SSL_FAILURE 成功しなかった場合。

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_cbc(), e, key, iv, 1));
// リソースを解放
```

A.3.2.8 function wolfSSL_EVP_CipherUpdate

```
int wolfSSL_EVP_CipherUpdate(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl,
    const unsigned char * in,
    int inl
)
```

データを暗号化/復号する関数。in バッファが暗号化または復号される対象として追加され、out バッファが結果を保持します。outl は暗号化/復号された情報の長さになります。

Parameters:

- **ctx** 暗号タイプを取得する構造体。
- **out** 出力を保持するバッファ。
- **outl** 出力のサイズに調整されます。
- **in** 操作を実行するバッファ。
- **inl** 入力バッファの長さ。

See:

- wolfSSL_EVP_CIPHER_CTX_new

- `wolfCrypt_Init`
- `wolfSSL_EVP_CIPHER_CTX_free`

Return:

- `SSL_SUCCESS` 成功した場合。
- `SSL_FAILURE` 成功しなかった場合。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
unsigned char out[100];
int outl;
unsigned char in[100];
int inl = 100;

ctx = wolfSSL_EVP_CIPHER_CTX_new();
// ctx をセットアップ
ret = wolfSSL_EVP_CipherUpdate(ctx, out, &outl, in, inl);
// ret 値をチェック
// バッファ out は outl バイトのデータを保持
// リソースを解放
```

A.3.2.9 function wolfSSL_EVP_CipherFinal

```
int wolfSSL_EVP_CipherFinal(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl
)
```

この関数は、パディングを追加する最終暗号操作を実行します。WOLFSSL_EVP_CIPHER_CTX 構造体に WOLFSSL_EVP_CIPH_NO_PADDING フラグが設定されている場合、1 が返され、暗号化/復号は実行されません。パディングフラグが設定されている場合、ctx が暗号化に設定されているときにパディングが追加され暗号化され、復号に設定されているときにパディング値がチェックされます。

Parameters:

- **ctx** 復号/暗号化に使用する構造体。
- **out** 最終復号/暗号化用のバッファ。
- **out1** 関数によってデータが追加されたときの out バッファのサイズ。

See: `wolfSSL_EVP_CIPHER_CTX_new`

Return:

- 1 成功時に返されます。
- 0 失敗が発生した場合。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int outl;
unsigned char out[64];
// ctx を作成
wolfSSL_EVP_CipherFinal(ctx, out, &outl);
```

A.3.2.10 function wolfSSL_EVP_CIPHER_CTX_set_key_length

```
int wolfSSL_EVP_CIPHER_CTX_set_key_length(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
```

```
    int keylen
)
```

WOLFSSL_EVP_CIPHER_CTX 構造体の鍵長のセッター関数。

Parameters:

- **ctx** 鍵長を設定する構造体。
- **keylen** 鍵の長さ。

See: wolfSSL_EVP_CIPHER_flags

Return:

- SSL_SUCCESS 正常に設定された場合。
- SSL_FAILURE 鍵長の設定に失敗した場合。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int keylen;
// ctx を作成
wolfSSL_EVP_CIPHER_CTX_set_key_length(ctx, keylen);
```

A.3.2.11 function wolfSSL_EVP_CIPHER_CTX_block_size

```
int wolfSSL_EVP_CIPHER_CTX_block_size(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

これは ctx ブロックサイズのゲッター関数です。

Parameters:

- **ctx** ブロックサイズを取得する暗号 ctx。

See: wolfSSL_EVP_CIPHER_block_size

Return: size ctx->block_size を返します。

Example

```
const WOLFSSL_EVP_CIPHER_CTX* ctx;
//ctx をセットアップ
printf("block size = %d\n", wolfSSL_EVP_CIPHER_CTX_block_size(ctx));
```

A.3.2.12 function wolfSSL_EVP_CIPHER_block_size

```
int wolfSSL_EVP_CIPHER_block_size(
    const WOLFSSL_EVP_CIPHER * cipher
)
```

これは暗号のブロックサイズのゲッター関数です。

Parameters:

- **cipher** ブロックサイズを取得する暗号。

See: wolfSSL_EVP_aes_256_ctr

Return: size ブロックサイズを返します。

Example

```
printf("block size = %d\n",  
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_aes_256_ecb()));
```

A.3.2.13 function wolfSSL_EVP_CIPHER_CTX_set_flags

```
void wolfSSL_EVP_CIPHER_CTX_set_flags(  
    WOLFSSL_EVP_CIPHER_CTX * ctx,  
    int flags  
)
```

WOLFSSL_EVP_CIPHER_CTX 構造体のセッター関数。

Parameters:

- **ctx** フラグを設定する構造体。
- **flag** 構造体に設定するフラグ。

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none 戻り値なし。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;  
int flag;  
// ctx を作成  
wolfSSL_EVP_CIPHER_CTX_set_flags(ctx, flag);
```

A.3.2.14 function wolfSSL_EVP_CIPHER_CTX_clear_flags

```
void wolfSSL_EVP_CIPHER_CTX_clear_flags(  
    WOLFSSL_EVP_CIPHER_CTX * ctx,  
    int flags  
)
```

WOLFSSL_EVP_CIPHER_CTX 構造体のクリア関数。

Parameters:

- **ctx** フラグをクリアする構造体。
- **flag** 構造体でクリアするフラグ値。

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none 戻り値なし。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;  
int flag;  
// ctx を作成  
wolfSSL_EVP_CIPHER_CTX_clear_flags(ctx, flag);
```

A.3.2.15 function wolfSSL_EVP_CIPHER_CTX_set_padding

```
int wolfSSL_EVP_CIPHER_CTX_set_padding(  
    WOLFSSL_EVP_CIPHER_CTX * c,  
    int pad  
)
```

パディングを使用するための WOLFSSL_EVP_CIPHER_CTX 構造体のセッター関数。

Parameters:

- **ctx** パディングフラグを設定する構造体。
- **padding** パディングを設定しない場合は 0、パディングを設定する場合は 1。

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- SSL_SUCCESS 正常に設定された場合。
- BAD_FUNC_ARG null 引数が渡された場合。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;  
// ctx を作成  
wolfSSL_EVP_CIPHER_CTX_set_padding(ctx, 1);
```

A.3.2.16 function wolfSSL_EVP_CIPHER_CTX_flags

```
unsigned long wolfSSL_EVP_CIPHER_CTX_flags(  
    const WOLFSSL_EVP_CIPHER_CTX * ctx  
)
```

WOLFSSL_EVP_CIPHER_CTX 構造体のゲッター関数。v1.1.0 で非推奨

Parameters:

- **ctx** フラグを取得する構造体。

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfSSL_EVP_CIPHER_flags

Return: unsigned long フラグ/モードの unsigned long。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;  
unsigned long flags;  
ctx = wolfSSL_EVP_CIPHER_CTX_new()  
flags = wolfSSL_EVP_CIPHER_CTX_flags(ctx);
```

A.3.2.17 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  
)
```

この関数は、WOLFSSL_BIO 構造体にキーを PEM 形式で書き込みます。

Parameters:

- **bio** PEM バッファを取得する WOLFSSL_BIO 構造体。
- **key** PEM 形式に変換するキー。
- **cipher** EVP 暗号構造体。
- **passwd** パスワード。
- **len** パスワードの長さ。
- **cb** パスワードコールバック。
- **arg** オプション引数。

See: [wolfSSL_PEM_read_bio_X509_AUX](#)

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE 失敗時。

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_EVP_PKEY* key;
int ret;
// bio を作成してキーをセットアップ
ret = wolfSSL_PEM_write_bio_PrivateKey(bio, key, NULL, NULL, 0, NULL, NULL);
//ret 値を確認
```

A.3.2.18 function wolfSSL_CTX_use_RSAPrivateKey_file

```
int wolfSSL_CTX_use_RSAPrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

この関数は、SSL 接続で使用されるプライベート RSA 鍵を SSL コンテキスト (WOLFSSL_CTX) に読み込みます。この関数は、wolfSSL が OpenSSL 互換レイヤーを有効にしてコンパイルされた場合 (-enable-opensslExtra、#define OPENSSL_EXTRA) にのみ利用可能であり、より一般的に使用される wolfSSL_CTX_use_PrivateKey_file() 関数と同一です。file 引数には、format 引数で指定された形式の RSA プライベート鍵ファイルへのポインタが含まれます。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **file** wolfSSL SSL コンテキストに読み込まれる RSA プライベート鍵を含むファイルの名前へのポインタ。形式は format 引数で指定されます。
- **format** file 引数で指定された RSA プライベート鍵のエンコーディングタイプ。指定可能な値には、SSL_FILETYPE_PEM および SSL_FILETYPE_ASN1 が含まれます。

See:

- [wolfSSL_CTX_use_PrivateKey_buffer](#)
- [wolfSSL_CTX_use_PrivateKey_file](#)
- [wolfSSL_use_RSAPrivateKey_file](#)
- [wolfSSL_use_PrivateKey_buffer](#)
- [wolfSSL_use_PrivateKey_file](#)

Return:

- 成功時は SSL_SUCCESS。

- 関数呼び出しが失敗した場合は SSL_FAILURE。考えられる原因には、入力鍵ファイルの形式が間違っている、または「format」引数を使用して間違った形式が指定されている、ファイルが存在しない、読み取れない、または破損している、メモリ不足状態が発生した、などがあります。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_RSAPrivateKey_file(ctx, "./server-key.pem",
                                         SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // プライベート鍵ファイルの読み込みエラー
}
...
```

A.3.2.19 function wolfSSL_use_certificate_file

```
int wolfSSL_use_certificate_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

この関数は、証明書ファイルを SSL セッション (WOLFSSL 構造体) に読み込みます。証明書ファイルは file 引数によって提供されます。format 引数は、ファイルの形式タイプ (SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM) を指定します。

Parameters:

- ssl** wolfSSL_new() で作成された WOLFSSL 構造体へのポインタ。
- file** wolfSSL SSL セッションに読み込まれる証明書を含むファイルの名前へのポインタ。形式は format 引数で指定されます。
- format** file 引数で指定された証明書のエンコーディングタイプ。指定可能な値には、SSL_FILETYPE_PEM および SSL_FILETYPE_ASN1 が含まれます。

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- 成功時は SSL_SUCCESS。
- 関数呼び出しが失敗した場合は SSL_FAILURE。考えられる原因には、ファイルの形式が間違っている、または「format」引数を使用して間違った形式が指定されている、ファイルが存在しない、読み取れない、または破損している、メモリ不足状態が発生した、ファイルに対する Base16 デコードが失敗した、などがあります。

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_certificate_file(ssl, "./client-cert.pem",
                                  SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // 証明書ファイルの読み込みエラー
}
```

```

}
...

```

A.3.2.20 function wolfSSL_use_PrivateKey_file

```

int wolfSSL_use_PrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)

```

この関数は、プライベート鍵ファイルを SSL セッション (WOLFSSL 構造体) に読み込みます。鍵ファイルは file 引数によって提供されます。format 引数は、ファイルの形式タイプ (SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM) を指定します。

Parameters:

- **ssl** wolfSSL_new() で作成された WOLFSSL 構造体へのポインタ。
- **file** wolfSSL SSL セッションに読み込まれる鍵ファイルを含むファイルの名前へのポインタ。形式は format 引数で指定されます。
- **format** file 引数で指定された鍵のエンコーディングタイプ。指定可能な値には、SSL_FILETYPE_PEM および SSL_FILETYPE_ASN1 が含まれます。

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_SetDevId

Return:

- 成功時は SSL_SUCCESS。
- 関数呼び出しが失敗した場合は SSL_FAILURE。考えられる原因には、ファイルの形式が間違っている、または「format」引数を使用して間違った形式が指定されている、ファイルが存在しない、読み取れない、または破損している、メモリ不足状態が発生した、ファイルに対する Base16 デコードが失敗した、鍵ファイルが暗号化されているがパスワードが提供されていない、などがあります。

外部鍵ストアを使用していてプライベート鍵を持っていない場合は、代わりに公開鍵を提供し、署名を処理するための暗号コールバックを登録することができます。これには、暗号コールバックまたは PK コールバックのいずれかを有効にしてビルドします。暗号コールバックを有効にするには、-enable-cryptocb または WOLF_CRYPT_CB を使用し、wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_SetDevId を使用して関連する devId を設定します。

Example

```

int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_PrivateKey_file(ssl, "./server-key.pem",
                                SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // 鍵ファイルの読み込みエラー
}
...

```

A.3.2.21 function wolfSSL_use_certificate_chain_file

```

int wolfSSL_use_certificate_chain_file(

```

```

    WOLFSSL * ssl,
    const char * file
)

```

この関数は、証明書のチェーンを SSL セッション (WOLFSSL 構造体) にロードします。証明書チェーンを含むファイルは file 引数で指定され、PEM 形式の証明書を含んでいる必要があります。この関数は、MAX_CHAIN_DEPTH(デフォルト = 9、internal.h で定義) 個までの証明書と、サブジェクト証明書を処理します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ
- **file** wolfSSL SSL セッションにロードする証明書のチェーンを含むファイルの名前へのポインタ。証明書は PEM 形式でなければなりません。

See:

- `wolfSSL_CTX_use_certificate_chain_file`
- `wolfSSL_CTX_use_certificate_chain_buffer`
- `wolfSSL_use_certificate_chain_buffer`

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE 関数呼び出しが失敗した場合、考えられる原因は次のとおりです: ファイルの形式が間違っている、または "format" 引数で与えられた形式が間違っている、ファイルが存在しない、読み取れない、または破損している、メモリ不足の状態が発生した

Example

```

int ret = 0;
WOLFSSL* ctx;
...
ret = wolfSSL_use_certificate_chain_file(ssl, "./cert-chain.pem");
if (ret != SSL_SUCCESS) {
    // 証明書ファイルのロードエラー
}
...

```

A.3.2.22 function wolfSSL_use_RSAPrivateKey_file

```

int wolfSSL_use_RSAPrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)

```

この関数は、SSL 接続で使用されるプライベート RSA 鍵を SSL セッション (WOLFSSL 構造体) にロードします。この関数は、wolfSSL が OpenSSL 互換レイヤーを有効にしてコンパイルされている場合 (-enable-opensslExtra、#define OPENSSL_EXTRA) にのみ利用可能であり、より一般的に使用される wolfSSL_use_PrivateKey_file() 関数と同一です。file 引数は、format で指定された形式の RSA プライベート鍵ファイルへのポインタを含みます。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ
- **file** wolfSSL SSL セッションにロードする RSA プライベート鍵を含むファイルの名前へのポインタ、形式は format で指定されます。
- **format** file で指定された RSA プライベート鍵のエンコーディングタイプ。指定可能な値は SSL_FILETYPE_PEM と SSL_FILETYPE_ASN1 です。

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 成功時
- SSL_FAILURE 関数呼び出しが失敗した場合、考えられる原因は次のとおりです: 入力鍵ファイルの形式が間違っている、または "format" 引数で与えられた形式が間違っている、ファイルが存在しない、読み取れない、または破損している、メモリ不足の状態が発生した

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_RSAPrivateKey_file(ssl, "./server-key.pem",
                                     SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // プライベート鍵ファイルのロードエラー
}
...
```

A.3.2.23 function wolfSSL_set_tlsext_status_type

```
long wolfSSL_set_tlsext_status_type(
    WOLFSSL * s,
    int type
)
```

この関数は、クライアントアプリケーションがサーバに OCSP ステータスレスポンス (OCSP ステープリングとも呼ばれます) を返送するよう要求する際に呼び出されます。現在、サポートされている唯一のタイプは TLSEXT_STATUSTYPE_ocsp です。

Parameters:

- **s** SSL_new() 関数によって作成された WOLFSSL 構造体へのポインタ。
- **type** TLSEXT_STATUSTYPE_ocsp のみがサポートされている SSL 拡張タイプ。

See:

- [wolfSSL_new](#)
- [wolfSSL_CTX_new](#)
- [wolfSSL_free](#)
- [wolfSSL_CTX_free](#)

Return:

- 1 成功時。
- 0 エラー時。

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);
```

A.3.2.24 function wolfSSL_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_get_peer_chain(  
    WOLFSSL * ssl  
)
```

ピアの証明書チェーンを取得します。

Parameters:

- **ssl** 有効な WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_get_chain_count](#)
- [wolfSSL_get_chain_length](#)
- [wolfSSL_get_chain_cert](#)
- [wolfSSL_get_chain_cert_pem](#)

Return:

- chain 成功した場合、呼び出しはピアの証明書チェーンを返します。
- 0 無効な WOLFSSL ポインタが関数に渡された場合に返されます。

Example

none

A.3.2.25 function wolfSSL_get_chain_count

```
int wolfSSL_get_chain_count(  
    WOLFSSL_X509_CHAIN * chain  
)
```

ピアの証明書チェーン数を取得します。

Parameters:

- **chain** 有効な WOLFSSL_X509_CHAIN 構造体へのポインタ。

See:

- [wolfSSL_get_peer_chain](#)
- [wolfSSL_get_chain_length](#)
- [wolfSSL_get_chain_cert](#)
- [wolfSSL_get_chain_cert_pem](#)

Return:

- Success 成功した場合、呼び出しはピアの証明書チェーン数を返します。
- 0 無効な chain ポインタが関数に渡された場合に返されます。

Example

none

A.3.2.26 function wolfSSL_get_chain_length

```
int wolfSSL_get_chain_length(  
    WOLFSSL_X509_CHAIN * chain,  
    int idx  
)
```

インデックス (idx) におけるピアの ASN1.DER 証明書の長さをバイト単位で取得します。

Parameters:

- **chain** 有効な WOLFSSL_X509_CHAIN 構造体へのポインタ。
- **idx** チェーンの開始インデックス。

See:

- [wolfSSL_get_peer_chain](#)
- [wolfSSL_get_chain_count](#)
- [wolfSSL_get_chain_cert](#)
- [wolfSSL_get_chain_cert_pem](#)

Return:

- Success 成功した場合、呼び出しはインデックスによるピアの証明書の長さをバイト単位で返します。
- 0 無効な chain ポインタが関数に渡された場合に返されます。

Example

none

A.3.2.27 function wolfSSL_get_chain_cert

```
unsigned char * wolfSSL_get_chain_cert(  
    WOLFSSL_X509_CHAIN * chain,  
    int idx  
)
```

インデックス (idx) におけるピアの ASN1.DER 証明書を取得します。

Parameters:

- **chain** 有効な WOLFSSL_X509_CHAIN 構造体へのポインタ。
- **idx** チェーンの開始インデックス。

See:

- [wolfSSL_get_peer_chain](#)
- [wolfSSL_get_chain_count](#)
- [wolfSSL_get_chain_length](#)
- [wolfSSL_get_chain_cert_pem](#)

Return:

- Success 成功した場合、呼び出しはインデックスによるピアの証明書を返します。
- 0 無効な chain ポインタが関数に渡された場合に返されます。

Example

none

A.3.2.28 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  
)
```

インデックス (idx) におけるピアの PEM 証明書を取得します。

Parameters:

- **chain** 有効な WOLFSSL_X509_CHAIN 構造体へのポインタ。
- **idx** チェーンの開始インデックス。

See:

- [wolfSSL_get_peer_chain](#)
- [wolfSSL_get_chain_count](#)
- [wolfSSL_get_chain_length](#)
- [wolfSSL_get_chain_cert](#)

Return:

- Success 成功した場合、呼び出しはインデックスによるピアの証明書を返します。
- 0 無効な chain ポインタが関数に渡された場合に返されます。

Example

none

A.3.2.29 function wolfSSL_get_sessionID

```
const unsigned char * wolfSSL_get_sessionID(  
    const WOLFSSL_SESSION * s  
)
```

セッション ID を取得します。セッション ID は常に 32 バイト長です。

Parameters:

- **session** 有効な wolfssl セッションへのポインタ。

See: [SSL_get_session](#)**Return:** id セッション ID。*Example*

none

A.3.2.30 function wolfSSL_X509_get_serial_number

```
int wolfSSL_X509_get_serial_number(  
    WOLFSSL_X509 * x509,  
    unsigned char * in,  
    int * inOutSz  
)
```

ピアの証明書シリアル番号を取得します。シリアル番号バッファ (in) は少なくとも 32 バイト長であり、入力として *inOutSz* 引数として提供される必要があります。関数を呼び出した後、*inOutSz* には in バッファに書き込まれた実際の長さ (バイト単位) が保持されます。

Parameters:

- **in** シリアル番号バッファで、少なくとも 32 バイト長である必要があります。
- **inOutSz** in バッファに書き込まれた実際の長さ (バイト単位) を保持します。

See: `SSL_get_peer_certificate`

Return:

- `SSL_SUCCESS` 成功時。
- `BAD_FUNC_ARG` 不正な関数引数が検出された場合に返されます。

Example

none

A.3.2.31 function wolfSSL_d2i_PKCS12_bio

```
WC_PKCS12 * wolfSSL_d2i_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 ** pkcs12
)
```

`wolfSSL_d2i_PKCS12_bio(d2i_PKCS12_bio)` は、`WOLFSSL_BIO` から `WC_PKCS12` 構造体へ PKCS12 情報をコピーします。情報は、構造体内でコンテンツ情報のリストとして分割され、オプションの MAC 情報を保持する構造体も含まれます。情報が `WC_PKCS12` 構造体内でチャンク (ただし復号されていない) に分割された後、呼び出すことによって解析および復号できます。

Parameters:

- **bio** PKCS12 バッファを読み取る `WOLFSSL_BIO` 構造体。
- **pkcs12** 作成された新しい PKCS12 構造体用の `WC_PKCS12` 構造体ポインタ。NULL でも可。

See:

- `wolfSSL_PKCS12_parse`
- `wc_PKCS12_free`

Return:

- `WC_PKCS12` `WC_PKCS12` 構造体へのポインタ。
- Failure 関数が失敗した場合、NULL を返します。

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio は PKCS12 ファイルを読み込みます。
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, "a password", &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//cert、pkey、およびオプションで certs スタックを使用します。
```

A.3.2.32 function wolfSSL_i2d_PKCS12_bio

```
WC_PKCS12 * wolfSSL_i2d_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 * pkcs12
)
```


wolfSSL_i2d_PKCS12_bio(i2d_PKCS12_bio) は、WC_PKCS12 構造体から WOLFSSL_BIO へ証明書情報をコピーします。

Parameters:

- **bio** PKCS12 バッファを書き込む WOLFSSL_BIO 構造体。
- **pkcs12** 入力としての PKCS12 構造体用の WC_PKCS12 構造体。

See:

- [wolfSSL_PKCS12_parse](#)
- [wc_PKCS12_free](#)

Return:

- 1 成功時。
- 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);
//DER ファイルを内部構造に変換します。
wc_d2i_PKCS12(buffer, bytes, pkcs12);
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
//PKCS12 構造体を bio に変換します。
wolfSSL_i2d_PKCS12_bio(bio, pkcs12);
wc_PKCS12_free(pkcs12);
//bio を使用します。
```

A.3.2.33 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 は、configure コマンドに `-enable-opensslextra` を追加することで有効にできます。復号にトリプル DES と RC4 を使用できるため、`opensslextra` を有効にする際にこれらの機能も有効にすることを推奨します (`-enable-des3 -enable-arc4`)。wolfSSL は現在 RC2 をサポートしていないため、RC2 での復号は現在利用できません。これは、OpenSSL コマンドラインで `p12` ファイルを作成する際に使用されるデフォルトの暗号化スキームで顕著になる可能性があります。wolfSSL_PKCS12_parse(PKCS12_parse)。この関数が最初に行うことは、存在する場合に MAC が正しいかどうかを確認することです。MAC が失敗した場合、関数は返され、保存されているコンテンツ情報の復号を試みません。この関数は、各コンテンツ情報を解析してバッグタイプを探し、バッグタイプが既知の場合、必要に応じて復号され、構築中の証明書リストまたは見つかった鍵として保存されます。すべてのバッグを解析した後、見つかった鍵は証明書リストと比較され、一致するペアが見つかります。この一致するペアは、鍵と証明書として返されます。オプションで、見つかった証明書リストは STACK_OF の証明書として返されます。現時点では、CRL、Secret、ま

たは SafeContents バグはスキップされ、解析されません。これらまたは他の「不明な」バグがスキップされているかどうかは、デバッグ出力を表示することで確認できます。フレンドリ名などの追加属性は、PKCS12 ファイルを解析する際にスキップされます。

Parameters:

- **pkcs12** 解析する WC_PKCS12 構造体。
- **passwd** PKCS12 を復号するためのパスワード。
- **pkey** PKCS12 から復号された秘密鍵を保持する構造体。
- **cert** PKCS12 から復号された証明書を保持する構造体。
- **stack** 追加の証明書のオプションスタック。

See:

- [wolfSSL_d2i_PKCS12_bio](#)
- [wc_PKCS12_free](#)

Return:

- SSL_SUCCESS PKCS12 の解析が成功した場合。
- SSL_FAILURE エラーケースに遭遇した場合。

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio は PKCS12 ファイルを読み込みます。
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, "a password", &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//cert、pkey、およびオプションで certs スタックを使用します。
```

A.4 wolfSSL 証明書と鍵

A.4.1 Functions

	Name
int	wc_KeyPemToDer (const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, const char * pass) PEM 形式のキーを DER 形式に変換します。
int	wc_CertPemToDer (const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, int type) この関数は、PEM 形式の証明書を DER 形式に変換します。OpenSSL 関数 PemToDer を呼び出します。

	Name
int	wc_GetPubKeyDerFromCert (struct DecodedCert * cert, byte * derKey, word32 * derKeySz) この関数は、入力された DecodedCert 構造体から DER 形式の公開鍵を取得します。この API を呼び出す前に、ユーザーは wc_InitDecodedCert() と wc_ParseCert() を呼び出す必要があります。wc_InitDecodedCert() は DER/ASN.1 エンコードされた証明書を受け入れます。PEM 証明書を DER に変換するには、wc_InitDecodedCert() を呼び出す前にまず wc_CertPemToDer() を使用してください。
int	wolfSSL_CTX_use_certificate_file (WOLFSSL_CTX * ctx, const char * file, int format) この関数は、証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは file 引数によって提供されます。format 引数はファイルのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM のいずれかです。適切な使用方法については、example を参照してください。
int	wolfSSL_CTX_use_PrivateKey_file (WOLFSSL_CTX * ctx, const char * file, int format) この関数は、秘密鍵ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは file 引数によって提供されます。format 引数はファイルのフォーマットタイプを指定します - SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用方法については、example を参照してください。
int	wolfSSL_CTX_load_verify_locations (WOLFSSL_CTX * ctx, const char * file, const char * path) この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである場合があります。同じファイルに複数の CA 証明書が含まれている場合、wolfSSL はファイル内に提示された順序でそれらをロードします。path 引数は、信頼されたルート CA の証明書を含むディレクトリの名前へのポインタです。file の値が NULL でない場合、必要でなければ path を NULL として指定できます。path が指定され、ライブラリのビルド時に NO_WOLFSSL_DIR が定義されていない場合、wolfSSL は指定されたディレクトリにあるすべての CA 証明書をロードします。この関数はディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダー "-----BEGIN CERTIFICATE-----" を持つ PEM 形式の CERT_TYPE ファイルを想定しています。

	Name
int	wolfSSL_CTX_load_verify_locations_ex (WOLFSSL_CTX * ctx, const char * file, const char * path, word32 flags) この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである場合があります。同じファイルに複数の CA 証明書が含まれている場合、wolfSSL はファイル内に提示された順序でそれらをロードします。path 引数は、信頼されたルート CA の証明書を含むディレクトリの名前へのポインタです。file の値が NULL でない場合、必要でなければ path を NULL として指定できます。path が指定され、ライブラリのビルド時に NO_WOLFSSL_DIR が定義されていない場合、wolfSSL は指定されたディレクトリにあるすべての CA 証明書をロードします。この関数は、指定されたフラグに基づいてディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダー “—BEGIN CERTIFICATE—” を持つ PEM 形式の CERT_TYPE ファイルを想定しています。
const char **	wolfSSL_get_system_CA_dirs (word32 * num) この関数は、wolfSSL_CTX_load_system_CA_certs が呼び出されたときに wolfSSL がシステム CA 証明書を検索するディレクトリを表す文字列の配列へのポインタを返します。証明書をアクセス可能なシステムディレクトリに保存しないシステム (Apple プラットフォームなど) では、この関数は常に NULL を返します。
int	wolfSSL_CTX_load_system_CA_certs (WOLFSSL_CTX * ctx) ほとんどのプラットフォーム (Linux および Windows を含む) において、この関数は OS 依存の CA 証明書ストアから WOLFSSL_CTX へ CA 証明書を読み込もうと試みます。読み込まれた証明書は信頼されます。
int	wolfSSL_CTX_use_certificate_chain_file (WOLFSSL_CTX * ctx, const char * file) この関数は、証明書チェーンを SSL コンテキスト (WOLFSSL_CTX) に読み込みます。証明書チェーンを含むファイルは file 引数によって提供され、PEM 形式の証明書を含む必要があります。この関数は、MAX_CHAIN_DEPTH(デフォルト = 9、internal.h で定義) 個までの証明書と、サブジェクト証明書を処理します。

	Name
int	wolfSSL_CTX_der_load_verify_locations (WOLFSSL_CTX * ctx, const char * file, int format) この関数は wolfSSL_CTX_load_verify_locations に似ていますが、DER 形式の CA ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードできます。PEM 形式の CA ファイルをロードするためにも使用できます。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数で指定されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである可能性があります。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイル内で提示された順序でそれらをロードします。format 引数は、証明書の形式が SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1(DER) のいずれかであることを指定します。 wolfSSL_CTX_load_verify_locations とは異なり、この関数は指定されたディレクトリパスからの CA 証明書のロードを許可しません。この関数は、wolfSSL ライブラリが WOLFSSL_DER_LOAD を定義してコンパイルされている場合にのみ利用可能です。
void	wolfSSL_SetCertCbCtx (WOLFSSL * ssl, void * ctx) この関数は検証コールバック用のユーザー CTX オブジェクト情報を格納します。
void	wolfSSL_CTX_SetCertCbCtx (WOLFSSL_CTX * ctx, void * userCtx) この関数は検証コールバック用のユーザー CTX オブジェクト情報を格納します。
int	wolfSSL_CTX_save_cert_cache (WOLFSSL_CTX * ctx, const char * fname) この関数は、証明書キャッシュをメモリからファイルに書き込みます。
int	wolfSSL_CTX_restore_cert_cache (WOLFSSL_CTX * ctx, const char * fname) この関数は、証明書キャッシュをファイルから永続化します。
int	wolfSSL_CTX_memsave_cert_cache (WOLFSSL_CTX * ctx, void * mem, int sz, int * used) この関数は、証明書キャッシュをメモリに永続化します。
int	wolfSSL_CTX_get_cert_cache_memsize (WOLFSSL_CTX * ctx) 証明書キャッシュの保存バッファに必要なサイズを返します。
char *	wolfSSL_X509_NAME_online (WOLFSSL_X509_NAME * name, char * in, int sz) この関数は x509 の名前をバッファにコピーします。
WOLFSSL_X509_NAME *	wolfSSL_X509_get_issuer_name (WOLFSSL_X509 * cert) この関数は証明書の発行者名を返します。
WOLFSSL_X509_NAME *	wolfSSL_X509_get_subject_name (WOLFSSL_X509 * cert) この関数は WOLFSSL_X509 構造体の subject メンバーを返します。

	Name
int	wolfSSL_X509_get_isCA (WOLFSSL_X509 * x509) WOLFSSL_X509 構造体の isCa メンバーをチェックし、その値を返します。
int	wolfSSL_X509_NAME_get_text_by_NID (WOLFSSL_X509_NAME * name, int nid, char * buf, int len) この関数は渡された NID 値に関連するテキストを取得します。
int	wolfSSL_X509_get_signature_type (WOLFSSL_X509 * x509) この関数は WOLFSSL_X509 構造体の sigOID メンバーに格納されている値を返します。
int	wolfSSL_X509_get_signature (WOLFSSL_X509 * x509, unsigned char * buf, int * bufSz) X509 署名を取得し、バッファに格納します。
int	wolfSSL_X509_STORE_add_cert (WOLFSSL_X509_STORE * store, WOLFSSL_X509 * x509) この関数は WOLFSSL_X509_STORE 構造体に証明書を追加します。
WOLFSSL_STACK *	wolfSSL_X509_STORE_CTX_get_chain (WOLFSSL_X509_STORE_CTX * ctx) この関数は WOLFSSL_X509_STORE_CTX 構造体の chain 変数の getter 関数です。現在、chain は入力されていません。
int	wolfSSL_X509_STORE_set_flags (WOLFSSL_X509_STORE * store, unsigned long flag) この関数は渡された WOLFSSL_X509_STORE 構造体の動作を変更するためのフラグを受け取ります。使用されるフラグの例として WOLFSSL_CRL_CHECK があります。
const byte *	wolfSSL_X509_notBefore (WOLFSSL_X509 * x509) この関数は、バイト配列としてエンコードされた証明書の「not before」有効期限を返します。
const byte *	wolfSSL_X509_notAfter (WOLFSSL_X509 * x509) この関数は、バイト配列としてエンコードされた証明書の「not after」有効期限を返します。
void	wolfSSL_CTX_set_client_cert_cb (WOLFSSL_CTX * ctx, client_cert_cb cb) クライアント証明書と秘密鍵を選択するコールバックを設定します。
void	wolfSSL_CTX_set_cert_cb (WOLFSSL_CTX * ctx, CertSetupCallback cb, void * arg) 汎用証明書セットアップコールバックを設定します。
const char *	wolfSSL_get_psk_identity_hint (const WOLFSSL *) この関数は、PSK アイデンティティヒントを返します。
const char *	wolfSSL_get_psk_identity (const WOLFSSL *) この関数は、Arrays 構造体の client_identity メンバへの定数ポインタを返します。
int	wolfSSL_CTX_use_psk_identity_hint (WOLFSSL_CTX * ctx, const char * hint) この関数は、hint 引数を WOLFSSL_CTX 構造体の server_hint メンバに格納します。
int	wolfSSL_use_psk_identity_hint (WOLFSSL * ssl, const char * hint) この関数は、hint 引数を WOLFSSL 構造体内の Arrays 構造体の server_hint メンバに格納します。

	Name
WOLFSSL_X509 *	wolfSSL_get_peer_certificate (WOLFSSL * ssl) この関数は、ピアの証明書を取得します。
WOLFSSL_X509 *	wolfSSL_get_chain_X509 (WOLFSSL_X509_CHAIN * chain, int idx) この関数は証明書チェーンからインデックス (idx) にあるピアの
char *	wolfSSL_X509_certificate を取得します。 wolfSSL_X509_get_subjectCN (WOLFSSL_X509 *) 証明書からサブジェクトのコモンネームを返します。
const unsigned char *	wolfSSL_X509_get_der (WOLFSSL_X509 * x509, int * outSz) この関数は、WOLFSSL_X509 構造体内の DER エンコードされた証明書を取得します。
WOLFSSL_ASN1_TIME *	wolfSSL_X509_get_notAfter (WOLFSSL_X509 *) この関数は、x509 が NULL であるかどうかをチェックし、そうでない場合は x509 構造体の notAfter メンバを返します。
int	wolfSSL_X509_version (WOLFSSL_X509 * x509) この関数は、X509 証明書のバージョンを取得します。
WOLFSSL_X509 *	wolfSSL_X509_d2i_fp (WOLFSSL_X509 ** x509, FILE * file) NO_STDIO_FILESYSTEM が定義されている場合、この関数はヒープメモリを割り当て、WOLFSSL_X509 構造体を初期化し、そのポインタを返します。
WOLFSSL_X509 *	wolfSSL_X509_load_certificate_file (const char * fname, int format) この関数は、x509 証明書をメモリにロードします。
unsigned char *	wolfSSL_X509_get_device_type (WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は、x509 構造体からデバイスタイプをバッファにコピーします。
unsigned char *	wolfSSL_X509_get_hw_type (WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は、WOLFSSL_X509 構造体の hwType メンバをバッファにコピーします。
unsigned char *	wolfSSL_X509_get_hw_serial_number (WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は、x509 オブジェクトの hwSerialNum メンバを返します。
int	wolfSSL_SetTmpDH (WOLFSSL * ssl, const unsigned char * p, int pSz, const unsigned char * g, int gSz) サーバ Diffie-Hellman エフェメラルパラメータの設定。この関数は、サーバが DHE を使用する暗号スイートをネゴシエートする場合に使用されるグループパラメータを設定します。
int	wolfSSL_SetTmpDH_buffer (WOLFSSL * ssl, const unsigned char * b, long sz, int format) この関数は、Diffie-Hellman パラメータのラッパーである wolfSSL_SetTMpDH_buffer_wrapper を呼び出します。

	Name
int	wolfSSL_SetTmpDH_file (WOLFSSL * ssl, const char * f, int format) この関数は、wolfSSL_SetTmpDH_file_wrapper を呼び出してサーバ Diffie-Hellman パラメータを設定します。
int	wolfSSL_CTX_SetTmpDH (WOLFSSL_CTX * ctx, const unsigned char * p, int pSz, const unsigned char * g, int gSz) サーバ CTX Diffie-Hellman のパラメータを設定します。
int	wolfSSL_CTX_SetTmpDH_buffer (WOLFSSL_CTX * ctx, const unsigned char * b, long sz, int format)wolfSSL_SetTmpDH_buffer_wrapper を呼び出すラッパー関数です。
int	wolfSSL_CTX_SetTmpDH_file (WOLFSSL_CTX * ctx, const char * f, int format) この関数は、wolfSSL_SetTmpDH_file_wrapper を呼び出してサーバ Diffie-Hellman パラメータを設定します。
int	wolfSSL_CTX_SetMinDhKey_Sz (WOLFSSL_CTX * ctx, word16 keySz_bits) この関数は、WOLFSSL_CTX 構造体の minDhKeySz メンバにアクセスして、Diffie_Hellman 鍵サイズの最小サイズ (ビット単位) を設定します。
int	wolfSSL_SetMinDhKey_Sz (WOLFSSL * ssl, word16 keySz_bits)WOLFSSL 構造体内の Diffie_Hellman 鍵の最小サイズ (ビット単位) を設定します。
int	wolfSSL_CTX_SetMaxDhKey_Sz (WOLFSSL_CTX * ctx, word16 keySz_bits) この関数は、WOLFSSL_CTX 構造体の maxDhKeySz メンバにアクセスして、Diffie_Hellman 鍵サイズの最大サイズ (ビット単位) を設定します。
int	wolfSSL_SetMaxDhKey_Sz (WOLFSSL * ssl, word16 keySz_bits)WOLFSSL 構造体内の Diffie_Hellman 鍵の最大サイズ (ビット単位) を設定します。
int	wolfSSL_GetDhKey_Sz (WOLFSSL * ssl)options 構造体のメンバである dhKeySz の値 (ビット単位) を返します。この値は、Diffie-Hellman 鍵サイズをバイト単位で表します。
int	wolfSSL_CTX_SetMinRsaKey_Sz (WOLFSSL_CTX * ctx, short keySz)WOLFSSL_CTX 構造体と WOLFSSL_CERT_MANAGER 構造体の両方で最小 RSA 鍵サイズを設定します。
int	wolfSSL_SetMinRsaKey_Sz (WOLFSSL * ssl, short keySz)WOLFSSL 構造体にある RSA の最小許容鍵サイズをビット単位で設定します。
int	wolfSSL_CTX_SetMinEccKey_Sz (WOLFSSL_CTX * ctx, short keySz)WOLF_CTX 構造体と WOLFSSL_CERT_MANAGER 構造体で ECC 鍵の最小サイズをビット単位で設定します。

	Name
int	wolfSSL_SetMinEccKey_Sz (WOLFSSL * ssl, short keySz) options 構造体の minEccKeySz メンバの値を設定します。options 構造体は WOLFSSL 構造体のメンバであり、ssl パラメータを通じてアクセスされます。
int	wolfSSL_make_eap_keys (WOLFSSL * ssl, void * key, unsigned int len, const char * label) この関数は、マスターシークレットから鍵材料を導出するために EAP-TLS と EAP-TTLS によって使用されます。
int	wolfSSL_CTX_load_verify_buffer (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は CA 証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。適切な使用方法については例を参照してください。
int	wolfSSL_CTX_load_verify_buffer_ex (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format, int userChain, word32 flags) この関数は CA 証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。_ex 版は PR 2413 で追加され、userChain と flags の追加引数をサポートします。
int	wolfSSL_CTX_load_verify_chain_buffer_format (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は CA 証明書チェーンバッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。適切な使用方法については例を参照してください。

	Name
int	wolfSSL_CTX_use_certificate_buffer (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。
int	wolfSSL_CTX_use_PrivateKey_buffer (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は秘密鍵バッファを SSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。
int	wolfSSL_CTX_use_certificate_chain_buffer (WOLFSSL_CTX * ctx, const unsigned char * in, long sz) この関数は証明書チェーンバッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。バッファは PEM 形式で、サブジェクトの証明書から始まり、ルート証明書で終わる必要があります。適切な使用方法については例を参照してください。
int	wolfSSL_use_certificate_buffer (WOLFSSL * ssl, const unsigned char * in, long sz, int format) この関数は、証明書バッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。

	Name
int	wolfSSL_use_PrivateKey_buffer (WOLFSSL * ssl, const unsigned char * in, long sz, int format) この関数は、秘密鍵バッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。
int	wolfSSL_use_certificate_chain_buffer (WOLFSSL * ssl, const unsigned char * in, long sz) この関数は、証明書チェーンバッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができる点が異なります。バッファはサイズ sz の in 引数によって提供されます。バッファは PEM フォーマットである必要があり、サブジェクトの証明書から始まり、ルート証明書で終わる必要があります。適切な使用方法については例を参照してください。
int	wolfSSL_UnloadCertsKeys (WOLFSSL * ssl) この関数は、SSL が所有する証明書または鍵をアンロードします。
int	wolfSSL_GetIVSize (WOLFSSL * ssl) WOLFSSL 構造体に保持されている specs 構造体の iv_size メンバを返します。
void	wolfSSL_KeepArrays (WOLFSSL * ssl) 通常、SSL ハンドシェイクの終了時に、wolfSSL は一時配列を解放します。ハンドシェイクが始まる前にこの関数を呼び出すと、wolfSSL は一時配列を解放しなくなります。一時配列は wolfSSL_get_keys() や PSK ヒントなどに必要になる場合があります。ユーザーが一時配列を使い終わったら、wolfSSL_FreeArrays() を呼び出してリソースを即座に解放するか、あるいは関連する SSL オブジェクトが解放されるときにリソースが解放されます。
void	wolfSSL_FreeArrays (WOLFSSL * ssl) 通常、SSL ハンドシェイクの終了時に、wolfSSL は一時配列を解放します。ハンドシェイクの前に wolfSSL_KeepArrays() が呼び出されていた場合、wolfSSL は一時配列を解放しません。この関数は一時配列を明示的に解放し、ユーザーが一時配列を使い終わり、SSL オブジェクトが解放されるのを待たずにこれらのリソースを解放したい場合に呼び出す必要があります。

	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) TLS 鍵を導出するための外部向けラッパー。
int	wolfSSL_X509_get_ext_by_NID (const WOLFSSL_X509 * x, int nid, int lastpos) この関数は、渡された NID 値に一致する拡張インデックスを検索して返します。
void *	wolfSSL_X509_get_ext_d2i (const WOLFSSL_X509 * x509, int nid, int * c, int * idx) この関数は渡された NID 値に一致する拡張を検索して返します。
int	wolfSSL_X509_digest (const WOLFSSL_X509 * x509, const WOLFSSL_EVP_MD * digest, unsigned char * buf, unsigned int * len) この関数は DER 証明書のハッシュを返します。
int	wolfSSL_use_PrivateKey (WOLFSSL * ssl, WOLFSSL_EVP_PKEY * pkey) WOLFSSL 構造体の秘密鍵を設定するために使用されます。
int	wolfSSL_use_PrivateKey_ASN1 (int pri, WOLFSSL * ssl, const unsigned char * der, long derSz) WOLFSSL 構造体の秘密鍵を設定するために使用されます。DER 形式の鍵バッファが必要です。
int	wolfSSL_use_RSAPrivateKey_ASN1 (WOLFSSL * ssl, unsigned char * der, long derSz) WOLFSSL 構造体の秘密鍵を設定するために使用されます。DER 形式の RSA 鍵バッファが必要です。
WOLFSSL_DH *	wolfSSL_DSA_dup_DH (const WOLFSSL_DSA * r) この関数は dsa のパラメータを新しく作成された WOLFSSL_DH 構造体に複製します。
WOLFSSL_X509 *	wolfSSL_d2i_X509_bio (WOLFSSL_BIO * bio, WOLFSSL_X509 ** x509) この関数は bio から DER バッファを取得し、それを WOLFSSL_X509 構造体に変換します。
WOLFSSL_X509 *	wolfSSL_PEM_read_bio_X509_AUX (WOLFSSL_BIO * bp, WOLFSSL_X509 ** x, wc_pem_password_cb * cb, void * u) この関数は wolfSSL_PEM_read_bio_X509 と同じように動作します。AUX は、信頼された/拒否されたユースケースや人間が読みやすいフレンドリ名などの追加情報を含むことを意味します。
long	wolfSSL_CTX_set_tmp_dh (WOLFSSL_CTX * ctx, WOLFSSL_DH * dh) WOLFSSL_CTX 構造体の dh メンバを Diffie-Hellman パラメータで初期化します。
WOLFSSL_DSA *	wolfSSL_PEM_read_bio_DSAParams (WOLFSSL_BIO * bp, WOLFSSL_DSA ** x, wc_pem_password_cb * cb, void * u) この関数は、bio 内の PEM バッファから DSA パラメータを取得します。

	Name
char *	WOLF_STACK_OF(WOLFSSL_X509) const この関数は、ピアの証明書チェーンを取得します。
WOLFSSL_ASN1_TIME *	wolfSSL_X509_get_next_altname(WOLFSSL_X509 *) この関数は、ピア証明書から次の代替名 (もしあれば) を返します。
	wolfSSL_X509_get_notBefore(WOLFSSL_X509 *) この関数は、x509 が NULL かどうかを確認し、NULL でない場合は x509 構造体の notBefore メンバを返します。

A.4.2 Functions Documentation

A.4.2.1 function wc_KeyPemToDer

```
int wc_KeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    const char * pass
)
```

PEM 形式のキーを DER 形式に変換します。

Parameters:

- **pem** PEM エンコードされた証明書へのポインタ。
- **pemSz** PEM バッファ (pem) のサイズ
- **buff** DerBuffer 構造体の buffer メンバーのコピーへのポインタ。
- **buffSz** DerBuffer 構造体に割り当てられたバッファスペースのサイズ。
- **pass** 関数に渡されるパスワード。

See: wc_PemToDer

Return:

- int 関数は正常実行時にバッファに書き込まれたバイト数を返します。
- int エラーを示す負の整数が返されます。

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){
    // バイトがバッファに書き込まれました。
}
```

A.4.2.2 function wc_CertPemToDer

```
int wc_CertPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    int type
)
```

この関数は、PEM 形式の証明書を DER 形式に変換します。OpenSSL 関数 PemToDer を呼び出します。

Parameters:

- **pem** PEM 形式の証明書へのポインタ。
- **pemSz** 証明書のサイズ。
- **buff** DER 形式にコピーされるバッファ。
- **buffSz** バッファのサイズ。
- **type** [asn_public.h](#) enum CertType にある証明書ファイルタイプ。

See: wc_PemToDer

Return: buffer バッファに書き込まれたバイトを返します。

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) {
    // バッファにバイトが書き込まれました
}
```

A.4.2.3 function wc_GetPubKeyDerFromCert

```
int wc_GetPubKeyDerFromCert(
    struct DecodedCert * cert,
    byte * derKey,
    word32 * derKeySz
)
```

この関数は、入力された DecodedCert 構造体から DER 形式の公開鍵を取得します。この API を呼び出す前に、ユーザーは wc_InitDecodedCert() と wc_ParseCert() を呼び出す必要があります。wc_InitDecodedCert() は DER/ASN.1 エンコードされた証明書を受け入れます。PEM 証明書を DER に変換するには、wc_InitDecodedCert() を呼び出す前にまず wc_CertPemToDer() を使用してください。

Parameters:

- **cert** X.509 証明書を保持する入力された DecodedCert 構造体
- **derKey** DER エンコードされた公開鍵を配置する出力バッファ
- **derKeySz** [IN/OUT] 入力時の derKey バッファのサイズ、返却時の公開鍵のサイズ。derKey が NULL として渡された場合、derKeySz は公開鍵に必要なバッファサイズに設定され、関数から LENGTH_ONLY_E が返されます。

See: wc_GetPubKeyDerFromCert

Return: 0 成功時、エラー時は負の値。derKey が NULL で長さのみを返す場合は LENGTH_ONLY_E。

A.4.2.4 function wolfSSL_CTX_use_certificate_file

```
int wolfSSL_CTX_use_certificate_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

この関数は、証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは file 引数によって提供されます。format 引数はファイルのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM のいずれかです。適切な使用方法については、example を参照してください。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインタ
- **file** wolfSSL SSL コンテキストにロードされる証明書を含むファイルの名前へのポインタ。
- **format** - file が指す証明書のフォーマット。可能なオプションは SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE 関数呼び出しが失敗した場合、考えられる原因には、ファイルが間違ったフォーマットである、または "format" 引数を使用して間違ったフォーマットが指定されている、ファイルが存在しない、読み取れない、または破損している、メモリ不足の状態が発生している、ファイルの Base16 デコードが失敗したことなどが含まれます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_file(ctx, "./client-cert.pem",
                                     SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // 証明書ファイルのロードエラー
}
...
```

A.4.2.5 function wolfSSL_CTX_use_PrivateKey_file

```
int wolfSSL_CTX_use_PrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

この関数は、秘密鍵ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは file 引数によって提供されます。format 引数はファイルのフォーマットタイプを指定します - SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用方法については、example を参照してください。

Parameters:

- **none** パラメータなし。

See:

- `wolfSSL_CTX_use_PrivateKey_buffer`
- `wolfSSL_use_PrivateKey_file`
- `wolfSSL_use_PrivateKey_buffer`
- `wc_CryptoCb_RegisterDevice`
- `wolfSSL_CTX_SetDevId`

Return:

- `SSL_SUCCESS` 成功時。
- `SSL_FAILURE` ファイルが間違ったフォーマットである、または “format” 引数を使用して間違ったフォーマットが指定されている。ファイルが存在しない、読み取れない、または破損している。メモリ不足の状態が発生している。ファイルの Base16 デコードが失敗した。キーファイルが暗号化されているが、パスワードが提供されていない。

外部キーストアを使用していて秘密鍵を持っていない場合、代わりに公開鍵を提供し、署名を処理するための暗号コールバックを登録できます。これには、暗号コールバックまたは PK コールバックのいずれかでビルドできます。暗号コールバックを有効にするには、`-enable-cryptocb` または `WOLF_CRYPTO_CB` でビルドし、`wc_CryptoCb_RegisterDevice` を使用して暗号コールバックを登録し、`wolfSSL_CTX_SetDevId` を使用して関連する `devId` を設定します。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_PrivateKey_file(ctx, "./server-key.pem",
                                     SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // 鍵ファイルのロードエラー
}
...
```

A.4.2.6 function wolfSSL_CTX_load_verify_locations

```
int wolfSSL_CTX_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    const char * path
)
```

この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (`WOLFSSL_CTX`) にロードします。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである場合があります。同じファイルに複数の CA 証明書が含まれている場合、wolfSSL はファイル内に提示された順序でそれらをロードします。path 引数は、信頼されたルート CA の証明書をを含むディレクトリの名前へのポインタです。file の値が NULL でない場合、必要であれば path を NULL として指定できます。path が指定され、ライブラリのビルド時に `NO_WOLFSSL_DIR` が定義されていない場合、wolfSSL は指定されたディレクトリにあるすべての CA 証明書をロードします。この関数はディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダー “`---BEGIN CERTIFICATE---`” を持つ PEM 形式の `CERT_TYPE` ファイルを想定しています。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **file** PEM 形式の CA 証明書をを含むファイルの名前へのポインタ。
- **path** PEM 形式の証明書をロードするディレクトリの名前へのポインタ。

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` 成功時。
- `SSL_FAILURE` ctx が NULL の場合、または file と path の両方が NULL の場合に返されます。
- `SSL_BAD_FILETYPE` ファイルが間違ったフォーマットの場合に返されます。
- `SSL_BAD_FILE` ファイルが存在しない、読み取れない、または破損している場合に返されます。
- `MEMORY_E` メモリ不足の状態が発生した場合に返されます。
- `ASN_INPUT_E` ファイルの Base16 デコードが失敗した場合に返されます。
- `ASN_BEFORE_DATE_E` 現在の日付が before date より前の場合に返されます。
- `ASN_AFTER_DATE_E` 現在の日付が after date より後の場合に返されます。
- `BUFFER_E` チェーンバッファが受信バッファより大きい場合に返されます。
- `BAD_PATH_ERROR` path を開こうとしたときに `opendir()` が失敗した場合に返されます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_verify_locations(ctx, "./ca-cert.pem", NULL);
if (ret != WOLFSSL_SUCCESS) {
    // CA 証明書のロードエラー
}
...
```

A.4.2.7 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
)
```

この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである場合があります。同じファイルに複数の CA 証明書が含まれている場合、wolfSSL はファイル内に提示された順序でそれらをロードします。path 引数は、信頼されたルート CA の証明書を含むディレクトリの名前へのポインタです。file の値が NULL でない場合、必要であれば path を NULL として指定できます。path が指定され、ライブラリのビルド時に `NO_WOLFSSL_DIR` が定義されていない場合、wolfSSL は指定されたディレクトリにあるすべての CA 証明書をロードします。この関数は、指定されたフラグに基づいてディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダー “—BEGIN CERTIFICATE—” を持つ PEM 形式の `CERT_TYPE` ファイルを想定しています。

Parameters:

- `ctx` `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。

- **file** PEM 形式の CA 証明書を含むファイル名へのポインタ。
- **path** PEM 形式の証明書を読み込むディレクトリ名へのポインタ。
- **flags** 指定可能なマスク値: WOLFSSL_LOAD_FLAG_IGNORE_ERR、WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY、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 成功時。
- SSL_FAILURE ctx が NULL の場合、または file と path の両方が NULL の場合に返されます。少なくとも 1 つの証明書が正常にロードされたが、1 つ以上が失敗した場合もこれが返されます。理由についてはエラースタックを確認してください。
- SSL_BAD_FILETYPE ファイルが間違ったフォーマットの場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファより大きい場合に返されます。
- BAD_PATH_ERROR path を開こうとしたときに opendir() が失敗した場合に返されます。

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) {
    // CA 証明書の読み込みエラー
}
...
```

A.4.2.8 function wolfSSL_get_system_CA_dirs

```
const char ** wolfSSL_get_system_CA_dirs(
    word32 * num
)
```

この関数は、wolfSSL_CTX_load_system_CA_certs が呼び出されたときに wolfSSL がシステム CA 証明書を検索するディレクトリを表す文字列の配列へのポインタを返します。証明書をアクセス可能なシステムディレクトリに保存しないシステム (Apple プラットフォームなど) では、この関数は常に NULL を返します。

Parameters:

- **num** 文字列配列の長さが格納される word32 へのポインタ。

See:

- [wolfSSL_CTX_load_system_CA_certs](#)
- [wolfSSL_CTX_load_verify_locations](#)
- [wolfSSL_CTX_load_verify_locations_ex](#)

Return:

- 成功時は有効なポインタ。
- 失敗時は NULL ポインタ。

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]);
}
...
```

A.4.2.9 function wolfSSL_CTX_load_system_CA_certs

```
int wolfSSL_CTX_load_system_CA_certs(
    WOLFSSL_CTX * ctx
)
```

ほとんどのプラットフォーム (Linux および Windows を含む) において、この関数は OS 依存の CA 証明書ストアから WOLFSSL_CTX へ CA 証明書を読み込もうと試みます。読み込まれた証明書は信頼されます。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された SSL コンテキストへのポインタ。

See:

- `wolfSSL_get_system_CA_dirs`
- `wolfSSL_CTX_load_verify_locations`
- `wolfSSL_CTX_load_verify_locations_ex`

Return:

- 成功時は WOLFSSL_SUCCESS。
- システム CA 証明書が読み込まれなかった場合は WOLFSSL_BAD_PATH。
- その他の失敗タイプの場合は WOLFSSL_FAILURE(例: Windows 証明書ストアが適切に閉じられなかった場合)。

Apple プラットフォーム (macOS を除く) では、証明書をシステムから取得できないため、wolfSSL 証明書マネージャに読み込むことができません。これらのプラットフォームでは、この関数は、WOLFSSL_CTX にバインドされた TLS 接続において、ユーザーによって読み込まれた証明書に対してピア証明書の信頼性を最初に認証できない場合に、ネイティブシステムの信頼 API を使用してピア証明書チェーンの信頼性を検証できるようにします。

サポートおよびテストされているプラットフォームは、Linux(Debian、Ubuntu、Gentoo、Fedora、RHEL)、Windows 10/11、Android、macOS、および iOS です。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_system_CA_certs(ctx,);
if (ret != WOLFSSL_SUCCESS) {
    // システム CA 証明書の読み込みエラー
}
```

```
}
...
```

A.4.2.10 function wolfSSL_CTX_use_certificate_chain_file

```
int wolfSSL_CTX_use_certificate_chain_file(
    WOLFSSL_CTX * ctx,
    const char * file
)
```

この関数は、証明書チェーンを SSL コンテキスト (WOLFSSL_CTX) に読み込みます。証明書チェーンを含むファイルは file 引数によって提供され、PEM 形式の証明書を含む必要があります。この関数は、MAX_CHAIN_DEPTH(デフォルト = 9、internal.h で定義) 個までの証明書と、サブジェクト証明書を処理します。

Parameters:

- **ctx** `wolfSSL_CTX_new()` を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **file** wolfSSL SSL コンテキストに読み込まれる証明書チェーンを含むファイルの名前へのポインタ。証明書は PEM 形式である必要があります。

See:

- `wolfSSL_CTX_use_certificate_file`
- `wolfSSL_CTX_use_certificate_buffer`
- `wolfSSL_use_certificate_file`
- `wolfSSL_use_certificate_buffer`

Return:

- 成功時は SSL_SUCCESS。
- 関数呼び出しが失敗した場合は SSL_FAILURE。考えられる原因には、ファイルの形式が間違っている、または「format」引数を使用して間違った形式が指定されている、ファイルが存在しない、読み取れない、または破損している、メモリ不足状態が発生した、などがあります。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_chain_file(ctx, "./cert-chain.pem");
if (ret != SSL_SUCCESS) {
    // 証明書ファイルの読み込みエラー
}
...
```

A.4.2.11 function wolfSSL_CTX_der_load_verify_locations

```
int wolfSSL_CTX_der_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

この関数は `wolfSSL_CTX_load_verify_locations` に似ていますが、DER 形式の CA ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードできます。PEM 形式の CA ファイルをロードするためにも使用できます。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数で指定されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである可能性があります。複数の CA 証明書が同じファイルに含

まれている場合、wolfSSL はファイル内で提示された順序でそれらをロードします。format 引数は、証明書の形式が SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1(DER) のいずれかであることを指定します。wolfSSL_CTX_load_verify_locations とは異なり、この関数は指定されたディレクトリパスからの CA 証明書のロードを許可しません。この関数は、wolfSSL ライブラリが WOLFSSL_DER_LOAD を定義してコンパイルされている場合にのみ利用可能です。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインタ
- **file** wolfSSL SSL コンテキストにロードする CA 証明書を含むファイルの名前へのポインタ、形式は format で指定されます。
- **format** file で指定された証明書のエンコーディングタイプ。指定可能な値は SSL_FILETYPE_PEM と SSL_FILETYPE_ASN1 です。

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE 失敗時。

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) {
    // CA 証明書のロードエラー
}
...
```

A.4.2.12 function wolfSSL_SetCertCbCtx

```
void wolfSSL_SetCertCbCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

この関数は検証コールバック用のユーザー CTX オブジェクト情報を格納します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **ctx** WOLFSSL 構造体の verifyCbCtx メンバの値に設定される void ポインタ。

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: none 返り値はありません。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
```

```

(void*)ctx;
...
if(ssl != NULL){
    wolfSSL_SetCertCbCtx(ssl, ctx);
} else {
    // エラーケース、SSL が適切に初期化されていません。
}

```

A.4.2.13 function wolfSSL_CTX_SetCertCbCtx

```

void wolfSSL_CTX_SetCertCbCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)

```

この関数は検証コールバック用のユーザー CTX オブジェクト情報を格納します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ。
- **userCtx** WOLFSSL_CTX 構造体の verifyCbCtx メンバの値を設定するために使用される void ポインタ。

See:

- [wolfSSL_CTX_save_cert_cache](#)
- [wolfSSL_CTX_restore_cert_cache](#)
- [wolfSSL_CTX_set_verify](#)

Return: none 返り値はありません。

Example

```

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
void* userCtx = NULL; // ユーザー定義のコンテキストを割り当て
...
if(ctx != NULL){
    wolfSSL_SetCertCbCtx(ctx, userCtx);
} else {
    // エラーケース、SSL が適切に初期化されていません。
}

```

A.4.2.14 function wolfSSL_CTX_save_cert_cache

```

int wolfSSL_CTX_save_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)

```

この関数は、証明書キャッシュをメモリからファイルに書き込みます。

Parameters:

- **ctx** 証明書情報を保持する WOLFSSL_CTX 構造体へのポインタ。
- **fname** 書き込み用ファイルを指す定数 char 型ポインタ。

See:

- [CM_SaveCertCache](#)
- [DoMemSaveCertCache](#)

Return:

- SSL_SUCCESS CM_SaveCertCache が正常に終了した場合に返されます。
- BAD_FUNC_ARG いずれかの引数が NULL の場合に返されます。
- SSL_BAD_FILE 証明書キャッシュ保存ファイルを開くことができなかった場合に返されます。
- BAD_MUTEX_E ロック mutex が失敗した場合に返されます。
- MEMORY_E メモリの割り当てに失敗した場合に返されます。
- FWRITE_ERROR 証明書キャッシュファイルの書き込みに失敗しました。

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol def );
const char* fname;
...
if(wolfSSL_CTX_save_cert_cache(ctx, fname)){
    // ファイルが書き込まれました。
}
```

A.4.2.15 function wolfSSL_CTX_restore_cert_cache

```
int wolfSSL_CTX_restore_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

この関数は、証明書キャッシュをファイルから永続化します。

Parameters:

- **ctx** 証明書情報を保持する WOLFSSL_CTX 構造体へのポインタ。
- **fname** 読み取り用ファイルを指す定数 char 型ポインタ。

See:

- CM_RestoreCertCache
- XFOPEN

Return:

- SSL_SUCCESS 関数 CM_RestoreCertCache が正常に実行された場合に返されます。
- SSL_BAD_FILE XFOPEN が XBADFILE を返した場合に返されます。ファイルが破損しています。
- MEMORY_E 一時バッファに割り当てられたメモリが失敗した場合に返されます。
- BAD_FUNC_ARG fname または ctx が NULL 値を持つ場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* fname = " ファイルへのパス";
...
if(wolfSSL_CTX_restore_cert_cache(ctx, fname)){
    // 実行が成功したかどうかを確認します。
}
```

A.4.2.16 function wolfSSL_CTX_memsave_cert_cache

```
int wolfSSL_CTX_memsave_cert_cache(
    WOLFSSL_CTX * ctx,
    void * mem,
    int sz,
    int * used
)
```


)

この関数は、証明書キャッシュをメモリに永続化します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **mem** 宛先 (出力バッファ) への void 型ポインタ。
- **sz** 出力バッファのサイズ。
- **used** 証明書キャッシュヘッダのサイズへのポインタ。

See:

- DoMemSaveCertCache
- GetCertCacheMemSize
- CM_MemRestoreCertCache
- CM_GetCertCacheMemSize

Return:

- SSL_SUCCESS 関数の実行が成功し、エラーが発生しなかった場合に返されます。
- BAD_MUTEX_E WOLFSSL_CERT_MANAGER メンバ caLock が 0(ゼロ) でなかった mutex エラー。
- BAD_FUNC_ARG ctx、mem、または used が NULL の場合、または sz が 0(ゼロ) 以下の場合に返されます。
- BUFFER_E 出力バッファ mem が小さすぎました。

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){
    // 関数がエラーを返しました。
}
```

A.4.2.17 function wolfSSL_CTX_get_cert_cache_memsize

```
int wolfSSL_CTX_get_cert_cache_memsize(
    WOLFSSL_CTX * ctx
)
```

証明書キャッシュの保存バッファに必要なサイズを返します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された wolfSSL_CTX 構造体へのポインタ。

See: CM_GetCertCacheMemSize

Return:

- int 成功時にメモリサイズを表す整数値が返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合に返されます。
- BAD_MUTEX_E mutex ロックエラーがあった場合に返されます。

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){
    // メモリサイズの取得に成功しました。
}

```

A.4.2.18 function wolfSSL_X509_NAME_online

```

char * wolfSSL_X509_NAME_online(
    WOLFSSL_X509_NAME * name,
    char * in,
    int sz
)

```

この関数は x509 の名前をバッファにコピーします。

Parameters:

- **name** WOLFSSL_X509 構造体へのポインタ。
- **in** WOLFSSL_X509_NAME 構造体からコピーされた名前を保持するバッファ。
- **sz** バッファの最大サイズ。

See:

- [wolfSSL_X509_get_subject_name](#)
- [wolfSSL_X509_get_issuer_name](#)
- [wolfSSL_X509_get_isCA](#)
- [wolfSSL_get_peer_certificate](#)
- [wolfSSL_X509_version](#)

Return: 関数が正常に実行された場合、WOLFSSL_X509_NAME 構造体の name メンバーのデータを持つバッファへの char ポインタが返されます。

Example

```

WOLFSSL_X509 x509;
char* name;
...
name = wolfSSL_X509_NAME_online(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(name <= 0){
    // バッファに何もありません
}

```

A.4.2.19 function wolfSSL_X509_get_issuer_name

```

WOLFSSL_X509_NAME * wolfSSL_X509_get_issuer_name(
    WOLFSSL_X509 * cert
)

```

この関数は証明書の発行者名を返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ。

See:

- [wolfSSL_X509_get_subject_name](#)
- [wolfSSL_X509_get_isCA](#)
- [wolfSSL_get_peer_certificate](#)
- [wolfSSL_X509_NAME_online](#)

Return:

- point 渡された cert が NULL でない場合、WOLFSSL_X509 構造体の issuer メンバーへのポインタが返されます。
- NULL 渡された cert が NULL の場合。

Example

```
WOLFSSL_X509* x509;
WOLFSSL_X509_NAME issuer;
...
issuer = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(!issuer){
    // NULL が返されました
} else {
    // issuer は証明書の発行者名を保持しています
}
```

A.4.2.20 function wolfSSL_X509_get_subject_name

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_subject_name(
    WOLFSSL_X509 * cert
)
```

この関数は WOLFSSL_X509 構造体の subject メンバーを返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ。

See:

- [wolfSSL_X509_get_issuer_name](#)
- [wolfSSL_X509_get_isCA](#)
- [wolfSSL_get_peer_certificate](#)

Return: pointer WOLFSSL_X509_NAME 構造体へのポインタ。WOLFSSL_X509 構造体が NULL の場合、または構造体の subject メンバーが NULL の場合、ポインタは NULL になる可能性があります。

Example

```
WOLFSSL_X509* cert;
WOLFSSL_X509_NAME name;
...
name = wolfSSL_X509_get_subject_name(cert);
if(name == NULL){
    // NULL ケースを処理
}
```

A.4.2.21 function wolfSSL_X509_get_isCA

```
int wolfSSL_X509_get_isCA(
    WOLFSSL_X509 * x509
)
```

WOLFSSL_X509 構造体の isCa メンバーをチェックし、その値を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

See:

- [wolfSSL_X509_get_issuer_name](#)
- [wolfSSL_X509_get_isCA](#)

Return:

- isCA WOLFSSL_X509 構造体の isCa メンバーの値が返されます。
- 0 有効な x509 構造体が渡されなかった場合に返されます。

Example

```
WOLFSSL* ssl;
...
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_X509_get_isCA(ssl)){
    // これは CA です
}else {
    // 失敗ケース
}
```

A.4.2.22 function wolfSSL_X509_NAME_get_text_by_NID

```
int wolfSSL_X509_NAME_get_text_by_NID(
    WOLFSSL_X509_NAME * name,
    int nid,
    char * buf,
    int len
)
```

この関数は渡された NID 値に関連するテキストを取得します。

Parameters:

- **name** テキストを検索する WOLFSSL_X509_NAME。
- **nid** 検索する NID。
- **buf** 見つかったテキストを保持するバッファ。
- **len** バッファの長さ。

See: none

Return: int テキストバッファのサイズを返します。

Example

```
WOLFSSL_X509_NAME* name;
char buffer[100];
int bufferSz;
int ret;
// WOLFSSL_X509_NAME を取得
ret = wolfSSL_X509_NAME_get_text_by_NID(name, NID_commonName,
buffer, bufferSz);

// ret 値を確認
```

A.4.2.23 function wolfSSL_X509_get_signature_type

```
int wolfSSL_X509_get_signature_type(
    WOLFSSL_X509 * x509
```

)

この関数は WOLFSSL_X509 構造体の sigOID メンバーに格納されている値を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- int x509 オブジェクトから取得された整数値が返されます。

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){
    // 予期しない値を処理
}
```

A.4.2.24 function wolfSSL_X509_get_signature

```
int wolfSSL_X509_get_signature(
    WOLFSSL_X509 * x509,
    unsigned char * buf,
    int * bufSz
)
```

X509 署名を取得し、バッファに格納します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。
- **buf** バッファへの char ポインタ。
- **bufSz** バッファのサイズへの整数ポインタ。

See:

- `wolfSSL_X509_get_serial_number`
- `wolfSSL_X509_get_signature_type`
- `wolfSSL_X509_get_device_type`

Return:

- SSL_SUCCESS 関数が正常に実行された場合に返されます。署名はバッファに読み込まれます。
- SSL_FATAL_ERROR x509 構造体または bufSz メンバーが NULL の場合に返されます。sig 構造体の length メンバー (sig は x509 のメンバー) のチェックもあります。

Example

```

WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
unsigned char* buf; // 初期化
int* bufSz = sizeof(buf)/sizeof(unsigned char);
...
if(wolfSSL_X509_get_signature(x509, buf, bufSz) != SSL_SUCCESS){
    // 関数は正常に実行されませんでした
} else{
    // バッファは正しく書き込まれました
}

```

A.4.2.25 function wolfSSL_X509_STORE_add_cert

```

int wolfSSL_X509_STORE_add_cert(
    WOLFSSL_X509_STORE * store,
    WOLFSSL_X509 * x509
)

```

この関数は WOLFSSL_X509_STORE 構造体に証明書を追加します。

Parameters:

- **str** 証明書を追加する証明書ストア。
- **x509** 追加する証明書。

See: [wolfSSL_X509_free](#)

Return:

- SSL_SUCCESS 証明서가正常に追加された場合。
- SSL_FATAL_ERROR 証明서가正常に追加されなかった場合。

Example

```

WOLFSSL_X509_STORE* str;
WOLFSSL_X509* x509;
int ret;
ret = wolfSSL_X509_STORE_add_cert(str, x509);
// ret 値を確認

```

A.4.2.26 function wolfSSL_X509_STORE_CTX_get_chain

```

WOLFSSL_STACK * wolfSSL_X509_STORE_CTX_get_chain(
    WOLFSSL_X509_STORE_CTX * ctx
)

```

この関数は WOLFSSL_X509_STORE_CTX 構造体の chain 変数の getter 関数です。現在、chain は入力されていません。

Parameters:

- **ctx** 解析チェーンを取得する証明書ストアコンテキスト。

See: [wolfSSL_sk_X509_free](#)

Return:

- pointer 成功した場合、WOLFSSL_STACK(STACK_OF(WOLFSSL_X509) と同じ) ポインタを返します。
- Null 失敗時。

Example

```
WOLFSSL_STACK* sk;
WOLFSSL_X509_STORE_CTX* ctx;
sk = wolfSSL_X509_STORE_CTX_get_chain(ctx);
// sk が NULL でないか確認してから使用。使用後は sk を解放する必要があります
```

A.4.2.27 function wolfSSL_X509_STORE_set_flags

```
int wolfSSL_X509_STORE_set_flags(
    WOLFSSL_X509_STORE * store,
    unsigned long flag
)
```

この関数は渡された WOLFSSL_X509_STORE 構造体の動作を変更するためのフラグを受け取ります。使用されるフラグの例として WOLFSSL_CRL_CHECK があります。

Parameters:

- **str** フラグを設定する証明書ストア。
- **flag** 動作のためのフラグ。

See:

- wolfSSL_X509_STORE_new
- wolfSSL_X509_STORE_free

Return:

- SSL_SUCCESS フラグの設定時にエラーが発生しなかった場合。
- <0 失敗時に負の値が返されます。

Example

```
WOLFSSL_X509_STORE* str;
int ret;
// str を作成して設定
ret = wolfSSL_X509_STORE_set_flags(str, WOLFSSL_CRL_CHECKALL);
if (ret != SSL_SUCCESS) {
    //ret 値を確認してエラーケースを処理する
}
```

A.4.2.28 function wolfSSL_X509_notBefore

```
const byte * wolfSSL_X509_notBefore(
    WOLFSSL_X509 * x509
)
```

この関数は、バイト配列としてエンコードされた証明書の「not before」有効期限を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- byte notBeforeData を含むバイトが返されます。

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                             DYNAMIC_TYPE_X509);
...
byte* notBeforeData = wolfSSL_X509_notBefore(x509);
```

A.4.2.29 function wolfSSL_X509_notAfter

```
const byte * wolfSSL_X509_notAfter(
    WOLFSSL_X509 * x509
)
```

この関数は、バイト配列としてエンコードされた証明書の「not after」有効期限を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- byte notAfterData を含むバイトが返されます。

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                             DYNAMIC_TYPE_X509);
...
byte* notAfterData = wolfSSL_X509_notAfter(x509);
```

A.4.2.30 function wolfSSL_CTX_set_client_cert_cb

```
void wolfSSL_CTX_set_client_cert_cb(
    WOLFSSL_CTX * ctx,
    client_cert_cb cb
)
```

クライアント証明書と秘密鍵を選択するコールバックを設定します。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクト。
- **cb** クライアント証明書と鍵を選択するコールバック関数。

See: [wolfSSL_CTX_set_cert_cb](#)

Return: void

この関数は、ハンドシェイク中にクライアント証明書が要求されたときに呼び出されるコールバックをアプリケーションが登録できるようにします。コールバックは、使用する証明書と鍵を選択して提供できます。

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);
```

A.4.2.31 function wolfSSL_CTX_set_cert_cb

```
void wolfSSL_CTX_set_cert_cb(
    WOLFSSL_CTX * ctx,
    CertSetupCallback cb,
    void * arg
)
```

汎用証明書セットアップコールバックを設定します。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクト。
- **cb** 証明書セットアップ用のコールバック関数。
- **arg** コールバックに渡すユーザー引数。

See: [wolfSSL_CTX_set_client_cert_cb](#)

Return: void

この関数は、証明書セットアップ中に呼び出されるコールバックをアプリケーションが登録できるようにします。コールバックは、カスタム証明書選択またはロードロジックを実行できます。

Example

```
int my_cert_setup_cb(WOLFSSL* ssl, void* arg) { ... }
wolfSSL_CTX_set_cert_cb(ctx, my_cert_setup_cb, NULL);
```

A.4.2.32 function wolfSSL_get_psk_identity_hint

```
const char * wolfSSL_get_psk_identity_hint(
    const WOLFSSL *
)
```

この関数は、PSK アイデンティティヒントを返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See: [wolfSSL_get_psk_identity](#)

Return:

- pointer WOLFSSL 構造体の arrays メンバに格納されていた値への const char 型ポインタが返されます。
- NULL WOLFSSL または Arrays 構造体が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char* idHint;
...
idHint = wolfSSL_get_psk_identity_hint(ssl);
```



```

if(idHint){
    // ヒントが取得されました。
    return idHint;
} else {
    // ヒントの取得に成功しませんでした。
}

```

A.4.2.33 function wolfSSL_get_psk_identity

```

const char * wolfSSL_get_psk_identity(
    const WOLFSSL *
)

```

この関数は、Arrays 構造体の client_identity メンバへの定数ポインタを返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_get_psk_identity_hint](#)
- [wolfSSL_use_psk_identity_hint](#)

Return:

- string Arrays 構造体の client_identity メンバの文字列値。
- NULL WOLFSSL 構造体が NULL、または WOLFSSL 構造体の Arrays メンバが NULL の場合。

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){
    // pskID に値がありません
}

```

A.4.2.34 function wolfSSL_CTX_use_psk_identity_hint

```

int wolfSSL_CTX_use_psk_identity_hint(
    WOLFSSL_CTX * ctx,
    const char * hint
)

```

この関数は、hint 引数を WOLFSSL_CTX 構造体の server_hint メンバに格納します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **hint** WOLFSSL_CTX 構造体にコピーされる定数 char ポインタ。

See: [wolfSSL_use_psk_identity_hint](#)

Return: SSL_SUCCESS 関数の実行が成功した場合に返されます。

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){
    // 関数は成功しました
return ret;
} else {
    // 失敗ケース
}

```

A.4.2.35 function wolfSSL_use_psk_identity_hint

```

int wolfSSL_use_psk_identity_hint(
    WOLFSSL * ssl,
    const char * hint
)

```

この関数は、hint 引数を WOLFSSL 構造体内の Arrays 構造体の server_hint メンバに格納します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **hint** メモリに保存されるヒントを保持する定数文字ポインタ。

See: [wolfSSL_CTX_use_psk_identity_hint](#)

Return:

- SSL_SUCCESS hint が WOLFSSL 構造体に正常に格納された場合に返されます。
- SSL_FAILURE WOLFSSL または Arrays 構造体が NULL の場合に返されます。

Example

```

WOLFSSL* ssl = wolfSSL_new(ctx);
const char* hint;
...
if(wolfSSL_use_psk_identity_hint(ssl, hint) != SSL_SUCCESS){
    // 失敗ケースを処理
}

```

A.4.2.36 function wolfSSL_get_peer_certificate

```

WOLFSSL_X509 * wolfSSL_get_peer_certificate(
    WOLFSSL * ssl
)

```

この関数は、ピアの証明書を取得します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_X509_get_issuer_name](#)
- [wolfSSL_X509_get_subject_name](#)
- [wolfSSL_X509_get_isCA](#)

Return:

- pointer 存在する場合、WOLFSSL_X509 構造体の peerCert メンバへのポインタ。
- 0 ピア証明書発行者のサイズが定義されていない場合に返されます。

Example

```

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
WOLFSSL_X509* peerCert = wolfSSL_get_peer_certificate(ssl);

if(peerCert){
    // ピア証明書へのポインタ peerCert があります
}

```

A.4.2.37 function wolfSSL_get_chain_X509

```

WOLFSSL_X509 * wolfSSL_get_chain_X509(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)

```

この関数は証明書チェーンからインデックス (idx) にあるピアの wolfSSL_X509_certificate を取得します。

Parameters:

- **chain** 動的メモリを使用しない SESSION_CACHE に使用される WOLFSSL_X509_CHAIN へのポインタ。
- **idx** WOLFSSL_X509 証明書のインデックス。

See:

- InitDecodedCert
- ParseCertRelative
- CopyDecodedToX509

Return: pointer WOLFSSL_X509 構造体へのポインタを返します。

返されたメモリを wolfSSL_FreeX509() を呼び出して解放することはユーザーの責任です。

Example

```

WOLFSSL_X509_CHAIN* chain = &session->chain;
int idx = 999; // idx を設定
...
WOLFSSL_X509_CHAIN ptr;
prt = wolfSSL_get_chain_X509(chain, idx);

if(ptr != NULL){
    // ptr は指定されたインデックスの証明書を含まます
    wolfSSL_FreeX509(ptr);
} else {
    // ptr は NULL です
}

```

A.4.2.38 function wolfSSL_X509_get_subjectCN

```

char * wolfSSL_X509_get_subjectCN(
    WOLFSSL_X509 *
)

```

証明書からサブジェクトのコモンネームを返します。

Parameters:

- **x509** 証明書情報を含む WOLFSSL_X509 構造体へのポインタ。

See:

- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- NULL x509 構造体が null の場合に返されます。
- string 成功時にはサブジェクトのコモンネームの文字列表現が返されます。

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                           DYNAMIC_TYPE_X509);

...
int x509Cn = wolfSSL_X509_get_subjectCN(x509);
if(x509Cn == NULL){
    // NULL ケースを処理
} else {
    // x509Cn にはコモンネームが含まれる
}
```

A.4.2.39 function wolfSSL_X509_get_der

```
const unsigned char * wolfSSL_X509_get_der(
    WOLFSSL_X509 * x509,
    int * outSz
)
```

この関数は、WOLFSSL_X509 構造体内の DER エンコードされた証明書を取得します。

Parameters:

- **x509** 証明書情報を含む WOLFSSL_X509 構造体へのポインタ。
- **outSz** WOLFSSL_X509 構造体の derBuffer メンバの長さ。

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 この関数は、DerBuffer 構造体の buffer メンバを返します。これは byte 型です。
- NULL x509 または outSz パラメータが NULL の場合に返されます。

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                           DYNAMIC_TYPE_X509);

int* outSz; // 初期化
...
byte* x509Der = wolfSSL_X509_get_der(x509, outSz);
if(x509Der == NULL){
```

```
// 失敗ケース、パラメータの 1 つが NULL だった
}
```

A.4.2.40 function wolfSSL_X509_get_notAfter

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notAfter(
    WOLFSSL_X509 *
)
```

この関数は、x509 が NULL であるかどうかをチェックし、そうでない場合は x509 構造体の notAfter メンバを返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

See: [wolfSSL_X509_get_notBefore](#)

Return:

- pointer x509 構造体の notAfter メンバへの ASN1_TIME を持つ構造体へのポインタ。
- NULL x509 オブジェクトが NULL の場合に返されます。

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){
    // 失敗ケース、x509 オブジェクトが null
}
```

A.4.2.41 function wolfSSL_X509_version

```
int wolfSSL_X509_version(
    WOLFSSL_X509 * x509
)
```

この関数は、X509 証明書のバージョンを取得します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_X509_get_subject_name](#)
- [wolfSSL_X509_get_issuer_name](#)
- [wolfSSL_X509_get_isCA](#)
- [wolfSSL_get_peer_certificate](#)

Return:

- 0 x509 構造体が NULL の場合に返されます。
- version x509 構造体に格納されているバージョンが返されます。

Example

```
WOLFSSL_X509* x509;
int version;
...
version = wolfSSL_X509_version(x509);
```

```
if(!version){
    // 関数は 0 を返した、失敗ケース
}
```

A.4.2.42 function wolfSSL_X509_d2i_fp

```
WOLFSSL_X509 * wolfSSL_X509_d2i_fp(
    WOLFSSL_X509 ** x509,
    FILE * file
)
```

NO_STDIO_FILESYSTEM が定義されている場合、この関数はヒープメモリを割り当て、WOLFSSL_X509 構造体を初期化し、そのポインタを返します。

Parameters:

- **x509** WOLFSSL_X509 ポインタへのポインタ。
- **file** FILE へのポインタである定義された型。

See:

- wolfSSL_X509_d2i
- XFTELL
- XREWIND
- XFSEEK

Return:

- *WOLFSSL_X509 関数が正常に実行された場合、WOLFSSL_X509 構造体ポインタが返されます。
- NULL XFTELL マクロの呼び出しが負の値を返した場合。

Example

```
WOLFSSL_X509* x509a = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
WOLFSSL_X509** x509 = x509a;
XFILE file; // (struct fs_file* にマップされる)
...
WOLFSSL_X509* newX509 = wolfSSL_X509_d2i_fp(x509, file);
if(newX509 == NULL){
    // 関数は NULL を返した
}
```

A.4.2.43 function wolfSSL_X509_load_certificate_file

```
WOLFSSL_X509 * wolfSSL_X509_load_certificate_file(
    const char * fname,
    int format
)
```

この関数は、x509 証明書をメモリにロードします。

Parameters:

- **fname** ロードする証明書ファイル。
- **format** 証明書のフォーマット。

See:

- InitDecodedCert
- PemToDer

- wolfSSL_get_certificate
- AssertNotNull

Return:

- pointer 正常に実行されると、WOLFSSL_X509 構造体へのポインタが返されます。
- NULL 証明書を書き込むことができなかった場合に返されます。

Example

```
#define cliCert    "certs/client-cert.pem"
...
X509* x509;
...
x509 = wolfSSL_X509_load_certificate_file(cliCert, SSL_FILETYPE_PEM);
AssertNotNull(x509);
```

A.4.2.44 function wolfSSL_X509_get_device_type

```
unsigned char * wolfSSL_X509_get_device_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

この関数は、x509 構造体からデバイスタイプをバッファにコピーします。

Parameters:

- **x509** WOLFSSL_X509_new() で作成された WOLFSSL_X509 構造体へのポインタ。
- **in** デバイスタイプ (バッファ) を保持するバイト型へのポインタ。
- **inOutSz** パラメータ inOutSz または x509 構造体の deviceTypeSz メンバのいずれか小さい方。

See:

- wolfSSL_X509_get_hw_type
- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_d2i

Return:

- pointer x509 構造体からデバイスタイプを保持するバイトポインタが返されます。
- NULL バッファサイズが NULL の場合に返されます。

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMACLOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
byte* in;
int* inOutSz;
...
byte* deviceType = wolfSSL_X509_get_device_type(x509, in, inOutSz);

if(!deviceType){
    // 失敗ケース、NULL が返された
}
```

A.4.2.45 function wolfSSL_X509_get_hw_type

```
unsigned char * wolfSSL_X509_get_hw_type(
    WOLFSSL_X509 * x509,
```

```

    unsigned char * in,
    int * inOutSz
)

```

この関数は、WOLFSSL_X509 構造体の hwType メンバをバッファにコピーします。

Parameters:

- **x509** 証明書情報を含む WOLFSSL_X509 構造体へのポインタ。
- **in** バッファを表す byte 型へのポインタ。
- **inOutSz** バッファのサイズを表す int 型へのポインタ。

See:

- [wolfSSL_X509_get_hw_serial_number](#)
- [wolfSSL_X509_get_device_type](#)

Return:

- byte 関数は、以前に WOLFSSL_X509 構造体の hwType メンバに保持されていたデータのバイト型を返します。
- NULL inOutSz が NULL の場合に返されます。

Example

```

WOLFSSL_X509* x509; // X509 証明書
byte* in; // バッファを初期化
int* inOutSz; // バッファのサイズを保持
...
byte* hwType = wolfSSL_X509_get_hw_type(x509, in, inOutSz);

if(hwType == NULL){
    // 失敗ケース、関数は NULL を返した
}

```

A.4.2.46 function wolfSSL_X509_get_hw_serial_number

```

unsigned char * wolfSSL_X509_get_hw_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)

```

この関数は、x509 オブジェクトの hwSerialNum メンバを返します。

Parameters:

- **x509** 証明書情報を含む WOLFSSL_X509 構造体へのポインタ。
- **in** コピー先のバッファへのポインタ。
- **inOutSz** バッファのサイズへのポインタ。

See:

- [wolfSSL_X509_get_subject_name](#)
- [wolfSSL_X509_get_issuer_name](#)
- [wolfSSL_X509_get_isCA](#)
- [wolfSSL_get_peer_certificate](#)
- [wolfSSL_X509_version](#)

Return: pointer 関数は、x509 オブジェクトからロードされたシリアル番号を含む in バッファへのバイトポインタを返します。

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){
    // 失敗ケース
}

```

A.4.2.47 function wolfSSL_SetTmpDH

```

int wolfSSL_SetTmpDH(
    WOLFSSL * ssl,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)

```

サーバ Diffie-Hellman エフェメラルパラメータの設定。この関数は、サーバが DHE を使用する暗号スイートをネゴシエートする場合に使用されるグループパラメータを設定します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **p** Diffie-Hellman 素数パラメータ。
- **pSz** p のサイズ。
- **g** Diffie-Hellman 「生成元」パラメータ。
- **gSz** g のサイズ。

See: `SSL_accept`

Return:

- `SSL_SUCCESS` 成功時。
- `MEMORY_ERROR` メモリエラーが発生した場合に返されます。
- `SIDE_ERROR` この関数が SSL サーバではなく SSL クライアントで呼び出された場合に返されます。

Example

```

WOLFSSL* ssl;
static unsigned char p[] = {...};
static unsigned char g[] = {...};
...
wolfSSL_SetTmpDH(ssl, p, sizeof(p), g, sizeof(g));

```

A.4.2.48 function wolfSSL_SetTmpDH_buffer

```

int wolfSSL_SetTmpDH_buffer(
    WOLFSSL * ssl,
    const unsigned char * b,
    long sz,
    int format
)

```

この関数は、Diffie-Hellman パラメータのラッパーである `wolfSSL_SetTmpDH_buffer_wrapper` を呼び出します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** `wolfSSL_SetTmpDH_file_wrapper` から渡される割り当てられたバッファ。
- **sz** ファイル (`wolfSSL_SetTmpDH_file_wrapper` 内の `fname`) のサイズを保持する long int 型。
- **format** `wolfSSL_SetTmpDH_file_wrapper()` から渡される証明書形式の表現である整数型。

See:

- `wolfSSL_SetTmpDH_buffer_wrapper`
- `wc_DhParamsLoad`
- `wolfSSL_SetTmpDH`
- `PemToDer`
- `wolfSSL_CTX_SetTmpDH`
- `wolfSSL_CTX_SetTmpDH_file`

Return:

- `SSL_SUCCESS` 実行が成功した場合。
- `SSL_BAD_FILETYPE` ファイルタイプが PEM でなく、ASN.1 でもない場合。また、`wc_DhParamsLoad` が正常に返されなかった場合にも返されます。
- `SSL_NO_PEM_HEADER` PEM ヘッダが存在しない場合、`PemToDer` から返されます。
- `SSL_BAD_FILE` `PemToDer` でファイルエラーがあった場合に返されます。
- `SSL_FATAL_ERROR` コピーエラーがあった場合、`PemToDer` から返されます。
- `MEMORY_E` メモリ割り当てエラーがあった場合。
- `BAD_FUNC_ARG` WOLFSSL 構造体が NULL の場合、またはサブルーチンに NULL 引数が渡された場合に返されます。
- `DH_KEY_SIZE_E` `wolfSSL_SetTmpDH()` または `wolfSSL_CTX_SetTmpDH()` でキーサイズエラーがある場合に返されます。
- `SIDE_ERROR` `wolfSSL_SetTmpDH` でサーバ側でない場合に返されます。

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);
```

A.4.2.49 function wolfSSL_SetTmpDH_file

```
int wolfSSL_SetTmpDH_file(
    WOLFSSL * ssl,
    const char * f,
    int format
)
```

この関数は、`wolfSSL_SetTmpDH_file_wrapper` を呼び出してサーバ Diffie-Hellman パラメータを設定します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **fname** 証明書を保持する定数 char 型ポインタ。
- **format** 証明書の形式を保持する整数型。

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 この関数とそのサブルーチンが正常に完了した場合に返されます。
- MEMORY_E この関数またはサブルーチンでメモリ割り当てが失敗した場合に返されます。
- SIDE_ERROR WOLFSSL 構造体内の Options 構造体の side メンバがサーバ側でない場合。
- SSL_BAD_FILETYPE 証明書が一連のチェックに失敗した場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL 構造体の minDhKeySz メンバの値より小さい場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL 構造体の maxDhKeySz メンバの値より大きい場合に返されます。
- BAD_FUNC_ARG WOLFSSL 構造体など、許可されていない引数値が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* dhParam;
...
AssertIntNE(SSL_SUCCESS,
    wolfSSL_SetTmpDH_file(ssl, dhParam, SSL_FILETYPE_PEM));
```

A.4.2.50 function wolfSSL_CTX_SetTmpDH

```
int wolfSSL_CTX_SetTmpDH(
    WOLFSSL_CTX * ctx,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

サーバ CTX Diffie-Hellman のパラメータを設定します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **p** serverDH_P 構造体の buffer メンバに読み込まれる定数 unsigned char 型ポインタ。
- **pSz** p のサイズを表す int 型で、MAX_DH_SIZE に初期化されます。
- **g** serverDH_G 構造体の buffer メンバに読み込まれる定数 unsigned char 型ポインタ。
- **gSz** g のサイズを表す int 型で、MAX_DH_SIZE に初期化されます。

See:

- [wolfSSL_SetTmpDH](#)
- [wc_DhParamsLoad](#)

Return:

- SSL_SUCCESS 関数とすべてのサブルーチンがエラーなく返された場合に返されます。
- BAD_FUNC_ARG ctx、p または g パラメータが NULL の場合に返されます。

- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の minDhKeySz メンバの値より小さい場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の maxDhKeySz メンバの値より大きい場合に返されます。
- MEMORY_E この関数またはサブルーチンでメモリ割り当てが失敗した場合に返されます。

Example

```
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){
    // 失敗ケース
}
```

A.4.2.51 function wolfSSL_CTX_SetTmpDH_buffer

```
int wolfSSL_CTX_SetTmpDH_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * b,
    long sz,
    int format
)
```

wolfSSL_SetTmpDH_buffer_wrapper を呼び出すラッパー関数です。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** バッファとして割り当てられ、wolfSSL_SetTmpDH_buffer_wrapper に渡される定数 unsigned char 型へのポインタ。
- **sz** wolfSSL_SetTmpDH_file_wrapper() の fname パラメータから導出される long 整数型。
- **format** wolfSSL_SetTmpDH_file_wrapper() から渡される整数型。

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTmpDH_buffer
- wolfSSL_SetTmpDH_file_wrapper
- wolfSSL_CTX_SetTmpDH_file

Return:

- 0 実行が成功した場合に返されます。
- BAD_FUNC_ARG ctx または buf パラメータが NULL の場合に返されます。
- MEMORY_E メモリ割り当てエラーがある場合。
- SSL_BAD_FILETYPE format が正しくない場合に返されます。

Example

```
static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
    Const char* fname, int format);
#ifdef WOLFSSL_SMALL_STACK
byte staticBuffer[1]; // ヒープ使用を強制
#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);
}

```

A.4.2.52 function wolfSSL_CTX_SetTmpDH_file

```

int wolfSSL_CTX_SetTmpDH_file(
    WOLFSSL_CTX * ctx,
    const char * f,
    int format
)

```

この関数は、wolfSSL_SetTmpDH_file_wrapper を呼び出してサーバ Diffie-Hellman パラメータを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **fname** 証明書ファイルへの定数文字ポインタ。
- **format** 証明書形式の表現である、wolfSSL_SetTmpDH_file_wrapper() から渡される整数型。

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 wolfSSL_SetTmpDH_file_wrapper またはそのサブルーチンのいずれかが正常に返された場合に返されます。
- MEMORY_E サブルーチンで動的メモリの割り当てが失敗した場合に返されます。
- BAD_FUNC_ARG ctx または fname パラメータが NULL の場合、またはサブルーチンに NULL 引数が渡された場合に返されます。
- SSL_BAD_FILE 証明書ファイルを開けない場合、または wolfSSL_SetTmpDH_file_wrapper からのファイルに対する一連のチェックが失敗した場合に返されます。
- SSL_BAD_FILETYPE wolfSSL_SetTmpDH_buffer_wrapper() から、形式が PEM でも ASN.1 でもない場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の minDhKeySz メンバの値より小さい場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の maxDhKeySz メンバの値より大きい場合に返されます。
- SIDE_ERROR サーバ側でない場合、wolfSSL_SetTmpDH() で返されます。
- SSL_NO_PEM_HEADER PEM ヘッダがない場合、PemToDer から返されます。
- SSL_FATAL_ERROR メモリコピーの失敗がある場合、PemToDer から返されます。

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));

```

A.4.2.53 function wolfSSL_CTX_SetMinDhKey_Sz

```

int wolfSSL_CTX_SetMinDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)

```

この関数は、WOLFSSL_CTX 構造体の minDhKeySz メンバにアクセスして、Diffie-Hellman 鍵サイズの最小サイズ (ビット単位) を設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz_bits** 最小 DH 鍵サイズをビット単位で設定するために使用される word16 型。WOLFSSL_CTX 構造体はこの情報を minDhKeySz メンバに保持します。

See:

- [wolfSSL_SetMinDhKey_Sz](#)
- [wolfSSL_CTX_SetMaxDhKey_Sz](#)
- [wolfSSL_SetMaxDhKey_Sz](#)
- [wolfSSL_GetDhKey_Sz](#)
- [wolfSSL_CTX_SetTMpDH_file](#)

Return:

- SSL_SUCCESS 関数が正常に完了した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合、または keySz_bits が 16,000 より大きいのか 8 で割り切れない場合に返されます。

Example

```

public static int CTX_SetMinDhKey_Sz(IntPtr ctx, short minDhKey){
    ...
    return wolfSSL_CTX_SetMinDhKey_Sz(local_ctx, minDhKeyBits);
}

```

A.4.2.54 function wolfSSL_SetMinDhKey_Sz

```

int wolfSSL_SetMinDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)

```

WOLFSSL 構造体内の Diffie-Hellman 鍵の最小サイズ (ビット単位) を設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz_bits** 最小 DH 鍵サイズをビット単位で設定するために使用される word16 型。WOLFSSL_CTX 構造体はこの情報を minDhKeySz メンバに保持します。

See:

- `wolfSSL_CTX_SetMinDhKey_Sz`
- `wolfSSL_GetDhKey_Sz`

Return:

- `SSL_SUCCESS` 最小サイズが正常に設定されました。
- `BAD_FUNC_ARG` WOLFSSL 構造体が `NULL` の場合、または `keySz_bits` が 16,000 より大きいか 8 で割り切れない場合。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz_bits;
...
if(wolfSSL_SetMinDhKey_Sz(ssl, keySz_bits) != SSL_SUCCESS){
    // 設定に失敗しました
}
```

A.4.2.55 function wolfSSL_CTX_SetMaxDhKey_Sz

```
int wolfSSL_CTX_SetMaxDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)
```

この関数は、WOLFSSL_CTX 構造体の `maxDhKeySz` メンバにアクセスして、Diffie-Hellman 鍵サイズの最大サイズ (ビット単位) を設定します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz_bits** 最大 DH 鍵サイズをビット単位で設定するために使用される `word16` 型。WOLFSSL_CTX 構造体はこの情報を `maxDhKeySz` メンバに保持します。

See:

- `wolfSSL_SetMinDhKey_Sz`
- `wolfSSL_CTX_SetMinDhKey_Sz`
- `wolfSSL_SetMaxDhKey_Sz`
- `wolfSSL_GetDhKey_Sz`
- `wolfSSL_CTX_SetTMpDH_file`

Return:

- `SSL_SUCCESS` 関数が正常に完了した場合に返されます。
- `BAD_FUNC_ARG` WOLFSSL_CTX 構造体が `NULL` の場合、または `keySz_bits` が 16,000 より大きいか 8 で割り切れない場合に返されます。

Example

```
public static int CTX_SetMaxDhKey_Sz(IntPtr ctx, short maxDhKey){
    ...
    return wolfSSL_CTX_SetMaxDhKey_Sz(local_ctx, keySz_bits);
}
```

A.4.2.56 function wolfSSL_SetMaxDhKey_Sz

```
int wolfSSL_SetMaxDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

WOLFSSL 構造体内の Diffie-Hellman 鍵の最大サイズ (ビット単位) を設定します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz** 最大 DH 鍵のビットサイズを表す word16 型。

See:

- `wolfSSL_CTX_SetMaxDhKey_Sz`
- `wolfSSL_GetDhKey_Sz`

Return:

- `SSL_SUCCESS` 最大サイズが正常に設定されました。
- `BAD_FUNC_ARG` WOLFSSL 構造体が NULL の場合、または `keySz` パラメータが許容サイズより大きい 8 で割り切れない場合。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz;
...
if(wolfSSL_SetMaxDhKey(ssl, keySz) != SSL_SUCCESS){
    // 設定に失敗しました
}
```

A.4.2.57 function `wolfSSL_GetDhKey_Sz`

```
int wolfSSL_GetDhKey_Sz(
    WOLFSSL * ssl
)
```

`options` 構造体のメンバである `dhKeySz` の値 (ビット単位) を返します。この値は、Diffie-Hellman 鍵サイズをバイト単位で表します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_SetMinDhKey_sz`
- `wolfSSL_CTX_SetMinDhKey_Sz`
- `wolfSSL_CTX_SetTmpDH`
- `wolfSSL_SetTmpDH`
- `wolfSSL_CTX_SetTmpDH_file`

Return:

- `dhKeySz` `ssl->options.dhKeySz` に保持されている値を返します。これはビット単位のサイズを表す整数値です。
- `BAD_FUNC_ARG` WOLFSSL 構造体が NULL の場合に返されます。

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){
```



```

    // 失敗ケース
} else {
    // dhKeySz は鍵のサイズを保持しています
}

```

A.4.2.58 function wolfSSL_CTX_SetMinRsaKey_Sz

```

int wolfSSL_CTX_SetMinRsaKey_Sz(
    WOLFSSL_CTX * ctx,
    short keySz
)

```

WOLFSSL_CTX 構造体と WOLFSSL_CERT_MANAGER 構造体の両方で最小 RSA 鍵サイズを設定します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **keySz** ctx 構造体と cm 構造体の minRsaKeySz に格納され、バイトに変換される short 整数型。

See: `wolfSSL_SetMinRsaKey_Sz`

Return:

- SSL_SUCCESS 関数の実行が成功した場合に返されます。
- BAD_FUNC_ARG ctx 構造体が NULL の場合、または keySz がゼロ未満か 8 で割り切れない場合に返されます。

Example

```

WOLFSSL_CTX* ctx = SSL_CTX_new(method);
(void)minDhKeyBits;
ourCert = myoptarg;
...
minDhKeyBits = atoi(myoptarg);
...
if(wolfSSL_CTX_SetMinRsaKey_Sz(ctx, minRsaKeyBits) != SSL_SUCCESS){
    ...
}

```

A.4.2.59 function wolfSSL_SetMinRsaKey_Sz

```

int wolfSSL_SetMinRsaKey_Sz(
    WOLFSSL * ssl,
    short keySz
)

```

WOLFSSL 構造体にある RSA の最小許容鍵サイズをビット単位で設定します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz** 最小鍵をビット単位で表す short 整数値。

See: `wolfSSL_CTX_SetMinRsaKey_Sz`

Return:

- SSL_SUCCESS 最小値が正常に設定されました。
- BAD_FUNC_ARG ssl 構造体が NULL の場合、または keySz がゼロ未満か 8 で割り切れない場合に返されます。

Example

```

WOLFSSL* ssl = wolfSSL_new(ctx);
short keySz;
...

int isSet = wolfSSL_SetMinRsaKey_Sz(ssl, keySz);
if(isSet != SSL_SUCCESS){
    設定に失敗しました
}

```

A.4.2.60 function wolfSSL_CTX_SetMinEccKey_Sz

```

int wolfSSL_CTX_SetMinEccKey_Sz(
    WOLFSSL_CTX * ctx,
    short keySz
)

```

WOLF_CTX 構造体と WOLFSSL_CERT_MANAGER 構造体で ECC 鍵の最小サイズをビット単位で設定します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **keySz** 最小 ECC 鍵サイズをビット単位で表す short 整数型。

See: [wolfSSL_SetMinEccKey_Sz](#)

Return:

- SSL_SUCCESS 実行が成功し、minEccKeySz メンバが設定された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合、または keySz が負の値か 8 で割り切れない場合に返されます。

Example

```

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
short keySz; // 最小鍵サイズ
...
if(wolfSSL_CTX_SetMinEccKey(ctx, keySz) != SSL_SUCCESS){
    // 最小鍵サイズの設定に失敗しました
}

```

A.4.2.61 function wolfSSL_SetMinEccKey_Sz

```

int wolfSSL_SetMinEccKey_Sz(
    WOLFSSL * ssl,
    short keySz
)

```

options 構造体の minEccKeySz メンバの値を設定します。options 構造体は WOLFSSL 構造体のメンバであり、ssl パラメータを通じてアクセスされます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz** 最小 ECC 鍵サイズを設定するために使用される値。options 構造体に値を設定します。

See:

- [wolfSSL_CTX_SetMinEccKey_Sz](#)
- [wolfSSL_CTX_SetMinRsaKey_Sz](#)
- [wolfSSL_SetMinRsaKey_Sz](#)

Return:

- SSL_SUCCESS 関数が options 構造体の minEccKeySz メンバを正常に設定した場合。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合、または鍵サイズ (keySz) が 0 未満か 8 で割り切れない場合。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx); // 新しいセッション
short keySz = 999; // 許容される最小鍵サイズに設定すべきです
...
if(wolfSSL_SetMinEccKey_Sz(ssl, keySz) != SSL_SUCCESS){
    // 失敗ケース
}
```

A.4.2.62 function wolfSSL_make_eap_keys

```
int wolfSSL_make_eap_keys(
    WOLFSSL * ssl,
    void * key,
    unsigned int len,
    const char * label
)
```

この関数は、マスターシークレットから鍵材料を導出するために EAP-TLS と EAP-TTLS によって使用されます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **key** p_hash 関数の結果を保持する void ポインタ変数。
- **len** key 変数の長さを表す unsigned 整数。
- **label** wc_PRF() でコピーされる定数 char ポインタ。

See:

- wc_PRF
- wc_HmacFinal
- wc_HmacUpdate

Return:

- BUFFER_E バッファの実際のサイズが許容される最大サイズを超えた場合に返されます。
- MEMORY_E メモリ割り当てでエラーが発生した場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
void* key;
unsigned int len;
const char* label;
...
return wolfSSL_make_eap_keys(ssl, key, len, label);
```

A.4.2.63 function wolfSSL_CTX_load_verify_buffer

```
int wolfSSL_CTX_load_verify_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

)

この関数は CA 証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ `sz` の `in` 引数によって提供されます。`format` はバッファの形式タイプを指定します。`SSL_FILETYPE_ASN1` または `SSL_FILETYPE_PEM` です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。適切な使用方法については例を参照してください。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファ (`in`) のサイズ。
- **format** バッファ証明書の形式、`SSL_FILETYPE_ASN1` または `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` 成功時。
- `SSL_BAD_FILETYPE` ファイルの形式が間違っている場合に返されます。
- `SSL_BAD_FILE` ファイルが存在しない、読み取れない、または破損している場合に返されます。
- `MEMORY_E` メモリ不足の状態が発生した場合に返されます。
- `ASN_INPUT_E` ファイルの Base16 デコードが失敗した場合に返されます。
- `BUFFER_E` チェーンバッファが受信バッファより大きい場合に返されます。

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) {
    // バッファからの CA 証明書のロードエラー
}
...
```

A.4.2.64 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
)
```

この関数は CA 証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ `sz` の `in` 引数によって提供されます。`format` はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。`_ex` 版は PR 2413 で追加され、`userChain` と `flags` の追加引数をサポートします。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファ (`in`) のサイズ。
- **format** バッファ証明書の形式、SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。
- **userChain** 形式 WOLFSSL_FILETYPE_ASN1 を使用している場合、これを非ゼロに設定すると、DER のチェーンが提示されていることを示します。
- **flags** `ssl.h` の WOLFSSL_LOAD_VERIFY_DEFAULT_FLAGS 付近を参照してください。

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 成功時。
- SSL_BAD_FILETYPE ファイルの形式が間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファより大きい場合に返されます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

// 期限切れ証明書を強制的にロードする例
ret = wolfSSL_CTX_load_verify_buffer_ex(ctx, certBuff, sz, SSL_FILETYPE_PEM,
    0, (WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY));
if (ret != SSL_SUCCESS) {
    // バッファからの CA 証明書のロードエラー
}
...
```

A.4.2.65 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
```

)

この関数は CA 証明書チェーンバッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ `sz` の `in` 引数によって提供されます。`format` はバッファの形式タイプを指定します。`SSL_FILETYPE_ASN1` または `SSL_FILETYPE_PEM` です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。適切な使用方法については例を参照してください。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファ (`in`) のサイズ。
- **format** バッファ証明書の形式、`SSL_FILETYPE_ASN1` または `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` 成功時。
- `SSL_BAD_FILETYPE` ファイルの形式が間違っている場合に返されます。
- `SSL_BAD_FILE` ファイルが存在しない、読み取れない、または破損している場合に返されます。
- `MEMORY_E` メモリ不足の状態が発生した場合に返されます。
- `ASN_INPUT_E` ファイルの Base16 デコードが失敗した場合に返されます。
- `BUFFER_E` チェーンバッファが受信バッファより大きい場合に返されます。

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) {
    // バッファからの CA 証明書のロードエラー
}
...
```

A.4.2.66 function wolfSSL_CTX_use_certificate_buffer

```
int wolfSSL_CTX_use_certificate_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ `sz` の `in` 引数によって提供されます。`format` はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された SSL コンテキストへのポインタ。
- **in** ロードする証明書を含む入力バッファ。
- **sz** 入力バッファのサイズ。
- **format** 入力バッファ (`in`) に格納されている証明書の形式。指定可能な値は SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。

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 成功時。
- SSL_BAD_FILETYPE ファイルの形式が間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。

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) {
    // バッファからの証明書のロードエラー
}
...
```

A.4.2.67 function wolfSSL_CTX_use_PrivateKey_buffer

```
int wolfSSL_CTX_use_PrivateKey_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は秘密鍵バッファを SSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ `sz` の `in` 引数によって提供されます。`format` はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された SSL コンテキストへのポインタ。

- **in** ロードする秘密鍵を含む入力バッファ。
- **sz** 入力バッファのサイズ。
- **format** 入力バッファ (in) に格納されている秘密鍵の形式。指定可能な値は SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。

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 成功時。
- SSL_BAD_FILETYPE ファイルの形式が間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。
- NO_PASSWORD 鍵ファイルが暗号化されているがパスワードが提供されていない場合に返されます。

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) {
    // バッファからの秘密鍵のロードエラー
}
...
```

A.4.2.68 function wolfSSL_CTX_use_certificate_chain_buffer

```
int wolfSSL_CTX_use_certificate_chain_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz
)
```

この関数は証明書チェーンバッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。バッファは PEM 形式で、サブジェクトの証明書から始まり、ルート証明書で終わる必要があります。適切な使用方法については例を参照してください。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#) で作成された SSL コンテキストへのポインタ。
- **in** ロードする PEM 形式の証明書チェーンを含む入力バッファ。
- **sz** 入力バッファのサイズ。

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` 成功時。
- `SSL_BAD_FILETYPE` ファイルの形式が間違っている場合に返されます。
- `SSL_BAD_FILE` ファイルが存在しない、読み取れない、または破損している場合に返されます。
- `MEMORY_E` メモリ不足の状態が発生した場合に返されます。
- `ASN_INPUT_E` ファイルの Base16 デコードが失敗した場合に返されます。
- `BUFFER_E` チェーンバッファが受信バッファより大きい場合に返されます。

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) {
    // バッファから証明書チェーンの読み込みに失敗
}
...
```

A.4.2.69 function wolfSSL_use_certificate_buffer

```
int wolfSSL_use_certificate_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、証明書バッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができる点が異なります。バッファはサイズ `sz` の `in` 引数によって提供されます。`format` はバッファのフォーマットタイプを指定します。`SSL_FILETYPE_ASN1` または `SSL_FILETYPE_PEM` です。適切な使用方法については例を参照してください。

Parameters:

- **ssl** `wolfSSL_new()` で作成された SSL セッションへのポインタ。
- **in** ロードする証明書を含むバッファ。
- **sz** バッファ内の証明書のサイズ。
- **format** ロードする証明書のフォーマット。指定可能な値は `SSL_FILETYPE_ASN1` または `SSL_FILETYPE_PEM` です。

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` 成功時。

- SSL_BAD_FILETYPE ファイルのフォーマットが間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードに失敗した場合に返されます。

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) {
    // バッファから証明書の読み込みに失敗
}
```

A.4.2.70 function wolfSSL_use_PrivateKey_buffer

```
int wolfSSL_use_PrivateKey_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、秘密鍵バッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。

Parameters:

- **ssl** `wolfSSL_new()` で作成された SSL セッションへのポインタ。
- **in** ロードする秘密鍵を含むバッファ。
- **sz** バッファ内の秘密鍵のサイズ。
- **format** ロードする秘密鍵のフォーマット。指定可能な値は SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。

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 成功時。
- SSL_BAD_FILETYPE ファイルのフォーマットが間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードに失敗した場合に返されます。
- NO_PASSWORD 鍵ファイルが暗号化されているがパスワードが提供されていない場合に返されます。

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) {
    // バッファから秘密鍵の読み込みに失敗
}

```

A.4.2.71 function wolfSSL_use_certificate_chain_buffer

```

int wolfSSL_use_certificate_chain_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz
)

```

この関数は、証明書チェーンバッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができます。バッファはサイズ sz の in 引数によって提供されます。バッファは PEM フォーマットである必要があり、サブジェクトの証明書から始まり、ルート証明書で終わる必要があります。適切な使用方法については例を参照してください。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL セッションへのポインタ。
- **in** ロードする証明書を含むバッファ。
- **sz** バッファ内の証明書のサイズ。

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_SUCCESS` 成功時。
- `SSL_BAD_FILETYPE` ファイルのフォーマットが間違っている場合に返されます。
- `SSL_BAD_FILE` ファイルが存在しない、読み取れない、または破損している場合に返されます。
- `MEMORY_E` メモリ不足状態が発生した場合に返されます。
- `ASN_INPUT_E` ファイルの Base16 デコードに失敗した場合に返されます。
- `BUFFER_E` チェーンバッファが受信バッファより大きい場合に返されます。

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) {

```

```
    // バッファから証明書チェーンの読み込みに失敗
}
```

A.4.2.72 function wolfSSL_UnloadCertsKeys

```
int wolfSSL_UnloadCertsKeys(
    WOLFSSL * ssl
)
```

この関数は、SSL が所有する証明書または鍵をアンロードします。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See: [wolfSSL_CTX_UnloadCAs](#)

Return:

- SSL_SUCCESS 関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL オブジェクトが NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
int unloadKeys = wolfSSL_UnloadCertsKeys(ssl);
if(unloadKeys != SSL_SUCCESS){
    // 失敗ケース
}
```

A.4.2.73 function wolfSSL_GetIVSize

```
int wolfSSL_GetIVSize(
    WOLFSSL * ssl
)
```

WOLFSSL 構造体に保持されている specs 構造体の iv_size メンバを返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_GetKeySize](#)
- [wolfSSL_GetClientWriteIV](#)
- [wolfSSL_GetServerWriteIV](#)

Return:

- iv_size ssl->specs.iv_size に保持されている値を返します。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int ivSize;
...
ivSize = wolfSSL_GetIVSize(ssl);

if(ivSize > 0){
```

```

    // ivSize は specs.iv_size 値を保持しています。
}

```

A.4.2.74 function wolfSSL_KeepArrays

```

void wolfSSL_KeepArrays(
    WOLFSSL * ssl
)

```

通常、SSL ハンドシェイクの終了時に、wolfSSL は一時配列を解放します。ハンドシェイクが始まる前にこの関数を呼び出すと、wolfSSL は一時配列を解放しなくなります。一時配列は wolfSSL_get_keys() や PSK ヒントなどに必要になる場合があります。ユーザーが一時的配列を使い終わったら、wolfSSL_FreeArrays() を呼び出してリソースを即座に解放するか、あるいは関連する SSL オブジェクトが解放されるときにリソースが解放されます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_FreeArrays

Return: none 戻り値なし。

Example

```

WOLFSSL* ssl;
...
wolfSSL_KeepArrays(ssl);

```

A.4.2.75 function wolfSSL_FreeArrays

```

void wolfSSL_FreeArrays(
    WOLFSSL * ssl
)

```

通常、SSL ハンドシェイクの終了時に、wolfSSL は一時配列を解放します。ハンドシェイクの前に wolfSSL_KeepArrays() が呼び出されていた場合、wolfSSL は一時配列を解放しません。この関数は一時配列を明示的に解放し、ユーザーが一時的配列を使い終わり、SSL オブジェクトが解放されるのを待たずにこれらのリソースを解放したい場合に呼び出す必要があります。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_KeepArrays

Return: none 戻り値なし。

Example

```

WOLFSSL* ssl;
...
wolfSSL_FreeArrays(ssl);

```

A.4.2.76 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
)

```

TLS 鍵を導出するための外部向けラッパー。

Parameters:

- **key_data** DeriveTlsKeys で割り当てられ、最終ハッシュを保持するために wc_PRF に渡されるバイトポインタ。
- **keyLen** DeriveTlsKeys で WOLFSSL 構造体の specs メンバから導出される word32 型。
- **ms** WOLFSSL 構造体内の arrays 構造体に保持されているマスターシークレットを保持する定数ポインタ型。
- **msLen** 列挙定義 SECRET_LEN でマスターシークレットの長さを保持する word32 型。
- **sr** WOLFSSL 構造体内の arrays 構造体の serverRandom メンバへの定数バイトポインタ。
- **cr** WOLFSSL 構造体内の arrays 構造体の clientRandom メンバへの定数バイトポインタ。
- **tls1_2** IsAtLeastTLSv1_2() から返される整数型。
- **hash_type** WOLFSSL 構造体に保持されている整数型。

See:

- wc_PRF
- DeriveTlsKeys
- IsAtLeastTLSv1_2

Return:

- 0 成功時に返されます。
- BUFFER_E labLen と seedLen の合計 (合計サイズを計算) が最大サイズを超えた場合に返されます。
- MEMORY_E メモリの割り当てに失敗した場合に返されます。

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);
    ...
}

```

A.4.2.77 function wolfSSL_X509_get_ext_by_NID

```

int wolfSSL_X509_get_ext_by_NID(
    const WOLFSSL_X509 * x,
    int nid,
    int lastpos
)

```

この関数は、渡された NID 値に一致する拡張インデックスを検索して返します。

Parameters:

- **x509** 拡張を検索するために解析する証明書。
- **nid** 見つける拡張 OID。
- **lastpos** lastPos 以降の拡張から検索を開始します。最初は-1 に設定します。

Return:

- ≥ 0 成功時、拡張インデックスが返されます。
- -1 拡張が見つからない場合、またはエラーが発生した場合。

Example

```
const WOLFSSL_X509* x509;
int lastPos = -1;
int idx;

idx = wolfSSL_X509_get_ext_by_NID(x509, NID_basic_constraints, lastPos);
```

A.4.2.78 function wolfSSL_X509_get_ext_d2i

```
void * wolfSSL_X509_get_ext_d2i(
    const WOLFSSL_X509 * x509,
    int nid,
    int * c,
    int * idx
)
```

この関数は渡された NID 値に一致する拡張を検索して返します。

Parameters:

- **x509** 拡張を解析する証明書。
- **nid** 検索する拡張 OID。
- **c** NULL でない場合、複数の拡張が見つかった場合は-2、見つからなかった場合は-1、見つかってクリティカルでない場合は 0、見つかってクリティカルな場合は 1 に設定されます。
- **idx** NULL の場合は最初に一致した拡張を返します。それ以外の場合、x509 に格納されていなければ idx から開始します。

See: wolfSSL_sk_ASN1_OBJECT_free

Return:

- pointer 成功した場合、STACK_OF(WOLFSSL_ASN1_OBJECT) ポインタが返されます。
- NULL 拡張が見つからない、またはエラーが発生した場合。

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);
// sk が NULL でないか確認してから使用。使用後は sk を解放する必要があります
```

A.4.2.79 function wolfSSL_X509_digest

```
int wolfSSL_X509_digest(
    const WOLFSSL_X509 * x509,
    const WOLFSSL_EVP_MD * digest,
    unsigned char * buf,
    unsigned int * len
)
```

この関数は DER 証明書のハッシュを返します。

Parameters:

- **x509** ハッシュを取得する証明書。
- **digest** 使用するハッシュアルゴリズム。
- **buf** ハッシュを保持するバッファ。
- **len** バッファの長さ。

See: none

Return:

- SSL_SUCCESS ハッシュの作成に成功した場合。
- SSL_FAILURE 不正な入力またはハッシュ失敗時に返されます。

Example

```
WOLFSSL* x509;  
unsigned char buffer[64];  
unsigned int bufferSz;  
int ret;  
  
ret = wolfSSL_X509_digest(x509, wolfSSL_EVP_sha256(), buffer, &bufferSz);  
// ret 値を確認
```

A.4.2.80 function wolfSSL_use_PrivateKey

```
int wolfSSL_use_PrivateKey(  
    WOLFSSL * ssl,  
    WOLFSSL_EVP_PKEY * pkey  
)
```

WOLFSSL 構造体の秘密鍵を設定するために使用されます。

Parameters:

- **ssl** 引数を設定する WOLFSSL 構造体。
- **pkey** 使用する秘密鍵。

See:

- **wolfSSL_new**
- **wolfSSL_free**

Return:

- SSL_SUCCESS 引数の設定に成功した場合。
- SSL_FAILURE NULL の ssl が渡された場合。すべてのエラーケースは負の値になります。

Example

```
WOLFSSL* ssl;  
WOLFSSL_EVP_PKEY* pkey;  
int ret;  
// ssl オブジェクトを作成して秘密鍵をセットアップ  
ret = wolfSSL_use_PrivateKey(ssl, pkey);  
// ret 値を確認
```

A.4.2.81 function wolfSSL_use_PrivateKey_ASN1

```
int wolfSSL_use_PrivateKey_ASN1(  
    int pri,  
    WOLFSSL * ssl,  
    const unsigned char * der,  
    long derSz
```


)

WOLFSSL 構造体の秘密鍵を設定するために使用されます。DER 形式の鍵バッファが必要です。

Parameters:

- **pri** 秘密鍵のタイプ。
- **ssl** 引数を設定する WOLFSSL 構造体。
- **der** DER 鍵を保持するバッファ。
- **derSz** der バッファのサイズ。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)
- [wolfSSL_use_PrivateKey](#)

Return:

- SSL_SUCCESS 秘密鍵の解析と設定に成功した場合。
- SSL_FAILURE NULL の ssl が渡された場合。すべてのエラーケースは負の値になります。

Example

```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// ssl オブジェクトを作成して秘密鍵をセットアップ
ret = wolfSSL_use_PrivateKey_ASN1(1, ssl, pkey, pkeySz);
// ret 値を確認
```

A.4.2.82 function wolfSSL_use_RSAPrivateKey_ASN1

```
int wolfSSL_use_RSAPrivateKey_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

WOLFSSL 構造体の秘密鍵を設定するために使用されます。DER 形式の RSA 鍵バッファが必要です。

Parameters:

- **ssl** 引数を設定する WOLFSSL 構造体。
- **der** DER 鍵を保持するバッファ。
- **derSz** der バッファのサイズ。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)
- [wolfSSL_use_PrivateKey](#)

Return:

- SSL_SUCCESS 秘密鍵の解析と設定に成功した場合。
- SSL_FAILURE NULL の ssl が渡された場合。すべてのエラーケースは負の値になります。

Example

```

WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// ssl オブジェクトを作成して RSA 秘密鍵をセットアップ
ret = wolfSSL_use_RSAPrivateKey_ASN1(ssl, pkey, pkeySz);
// ret 値を確認

```

A.4.2.83 function wolfSSL_DSA_dup_DH

```

WOLFSSL_DH * wolfSSL_DSA_dup_DH(
    const WOLFSSL_DSA * r
)

```

この関数は dsa のパラメータを新しく作成された WOLFSSL_DH 構造体に複製します。

Parameters:

- **dsa** 複製する WOLFSSL_DSA 構造体。

See: none

Return:

- WOLFSSL_DH 複製に成功した場合、WOLFSSL_DH 構造体を返します。
- NULL 失敗時。

Example

```

WOLFSSL_DH* dh;
WOLFSSL_DSA* dsa;
// dsa をセットアップ
dh = wolfSSL_DSA_dup_DH(dsa);

// dh が null でないか確認

```

A.4.2.84 function wolfSSL_d2i_X509_bio

```

WOLFSSL_X509 * wolfSSL_d2i_X509_bio(
    WOLFSSL_BIO * bio,
    WOLFSSL_X509 ** x509
)

```

この関数は bio から DER バッファを取得し、それを WOLFSSL_X509 構造体に変換します。

Parameters:

- **bio** DER 証明書バッファを持つ WOLFSSL_BIO 構造体へのポインタ。
- **x509** 作成された新しい WOLFSSL_X509 構造体に設定されるポインタ。

See: none

Return:

- pointer 成功時に WOLFSSL_X509 構造体ポインタを返します。
- Null 失敗時に NULL を返します。

Example

```

WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// DER を bio にロード

```

```
x509 = wolfSSL_d2i_X509_bio(bio, NULL);
// または
wolfSSL_d2i_X509_bio(bio, &x509);
// 返された x509 を使用 (NULL をチェック)
```

A.4.2.85 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
)
```

この関数は wolfSSL_PEM_read_bio_X509 と同じように動作します。AUX は、信頼された/拒否されたユーースケースや人間が読みやすいフレンドリ名などの追加情報を含むことを意味します。

Parameters:

- **bp** PEM バッファを取得する WOLFSSL_BIO 構造体。
- **x** 関数の副作用によって WOLFSSL_X509 を設定する場合。
- **cb** パスワードコールバック。
- **u** NULL 終端のユーザーパスワード。

See: wolfSSL_PEM_read_bio_X509

Return:

- WOLFSSL_X509 PEM バッファの解析に成功した場合、WOLFSSL_X509 構造体が返されます。
- Null PEM バッファの解析に失敗した場合。

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// bio をセットアップ
X509 = wolfSSL_PEM_read_bio_X509_AUX(bio, NULL, NULL, NULL);
//x509 が null でないことを確認してから使用
```

A.4.2.86 function wolfSSL_CTX_set_tmp_dh

```
long wolfSSL_CTX_set_tmp_dh(
    WOLFSSL_CTX * ctx,
    WOLFSSL_DH * dh
)
```

WOLFSSL_CTX 構造体の dh メンバを Diffie-Hellman パラメータで初期化します。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **dh** WOLFSSL_DH 構造体へのポインタ。

See: wolfSSL_BN_bn2bin

Return:

- SSL_SUCCESS 関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG ctx または dh 構造体が NULL の場合に返されます。
- SSL_FATAL_ERROR 構造体の値の設定にエラーがあった場合に返されます。
- MEMORY_E メモリの割り当てに失敗した場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL_DH* dh;
...
return wolfSSL_CTX_set_tmp_dh(ctx, dh);
```

A.4.2.87 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
)
```

この関数は、bio 内の PEM バッファから DSA パラメータを取得します。

Parameters:

- **bio** PEM メモリポインタを取得するための WOLFSSL_BIO 構造体へのポインタ。
- **x** 新しい WOLFSSL_DSA 構造体に設定されるポインタ。
- **cb** パスワードコールバック関数。
- **u** null 終端のパスワード文字列。

See: none

Return:

- WOLFSSL_DSA PEM バッファの解析に成功した場合、WOLFSSL_DSA 構造体を作成されて返されます。
- Null PEM バッファの解析に失敗した場合。

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_DSA* dsa;
// bio をセットアップ
dsa = wolfSSL_PEM_read_bio_DSAParams(bio, NULL, NULL, NULL);

// dsa が NULL でないことを確認してから dsa を使用
```

A.4.2.88 function WOLF_STACK_OF

```
WOLF_STACK_OF(
    WOLFSSL_X509
) const
```

この関数は、ピアの証明書チェーンを取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer ピアの Certificate スタックへのポインタを返します。
- NULL ピア証明書がない場合に返されます。

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);
if(chain){
    // ピア証明書チェーンへのポインタがあります
}
```

A.4.2.89 function wolfSSL_X509_get_next_altname

```
char * wolfSSL_X509_get_next_altname(
    WOLFSSL_X509 *
```

この関数は、ピア証明書から次の代替名 (もしあれば) を返します。

Parameters:

- **cert** wolfSSL_X509 構造体へのポインタ。

See:

- [wolfSSL_X509_get_issuer_name](#)
- [wolfSSL_X509_get_subject_name](#)

Return:

- NULL 次の代替名がない場合。
- cert->altNamesNext->name WOLFSSL_X509 構造体から返されます。altName リストからの文字列値が存在する場合に返されます。

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){
    //別の alt name はありません。
}
```

A.4.2.90 function wolfSSL_X509_get_notBefore

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notBefore(
    WOLFSSL_X509 *
```

この関数は、x509 が NULL かどうかを確認し、NULL でない場合は x509 構造体の notBefore メンバを返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

See: [wolfSSL_X509_get_notAfter](#)**Return:**

- pointer x509 構造体の notBefore メンバへの ASN1_TIME を持つ構造体へのポインタ。
- NULL x509 構造体が NULL の場合、関数は NULL を返します。

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){
    //x509 オブジェクトが NULL でした。
}
```

A.5 wolfSSL 接続、セッション、I/O**A.5.1 Functions**

	Name
long	wolfSSL_get_verify_depth (WOLFSSL * ssl) この関数は、有効なセッション(つまり、非 NULL のセッションオブジェクト (ssl) が存在する場合)に対して許可される最大チェーン深度を返します。デフォルトは9です。
char *	wolfSSL_get_cipher_list (int priority) 渡された優先度レベルでの暗号の名前を取得します。
int	wolfSSL_get_ciphers (char * buf, int len) この関数は、wolfSSL で有効になっている暗号を取得します。
const char *	wolfSSL_get_cipher_name (WOLFSSL * ssl) この関数は、引数を wolfSSL_get_cipher_name_internal に渡すことによって、DHE-RSA 形式で暗号名を取得します。
int	wolfSSL_get_fd (const WOLFSSL * ssl) この関数は、SSL 接続の入力機能として使用される読み取りファイル記述子 (fd) を返します。通常、これはソケットファイル記述子になります。
int	wolfSSL_get_wfd (const WOLFSSL * ssl) この関数は、SSL 接続の出力機能として使用される書き込みファイルディスクリプタ (fd) を返します。通常はソケットファイルディスクリプタになります。
int	wolfSSL_get_using_nonblock (WOLFSSL *) この関数により、アプリケーションは wolfSSL がノンブロッキング I/O を使用しているかどうかを判定できます。wolfSSL がノンブロッキング I/O を使用している場合、この関数は 1 を返し、それ以外の場合は 0 を返します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキングソケットと共に使用する場合は、wolfSSL_set_using_nonblock() を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、recvfrom 呼び出しがブロックすることを意味すると認識できます。

	Name
int	wolfSSL_write(WOLFSSL * ssl, const void * data, int sz) この関数は、バッファ data から sz バイトを SSL 接続 ssl に書き込みます。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_write() は SSL/TLS セッションをネゴシエートします。(D)TLSv1.3 を使用していて early data 機能がコンパイルされている場合、この関数はデータ送信が可能になるまでハンドシェイクを進めます。次の wolfSSL_Connect()、wolfSSL_Accept()、wolfSSL_read() の呼び出しでハンドシェイクが完了します。wolfSSL_write() は、ブロッキングとノンブロッキングの両方の I/O で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_write() は、下層の I/O が wolfSSL_write() を継続するために必要な要求を満たせなくなった時点で返されます。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、下層の I/O の準備ができたなら、wolfSSL_write() の呼び出しを繰り返す必要があります。下層の I/O がブロッキングの場合、wolfSSL_write() は、サイズ sz のバッファデータが完全に書き込まれるか、エラーが発生するまで返されません。
int	wolfSSL_read(WOLFSSL * ssl, void * data, int sz) この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファ data に読み込みます。読み取られたバイトは内部受信バッファから削除されます。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_read() は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズが 16kB の SSL レコードを使用します (最大レコードサイズは、/wolfssl/internal.h 内の MAX_RECORD_SIZE 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号できるようにする前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_read() の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。内部 wolfSSL 受信バッファには、まだ復号されていない追加データが待機している可能性があり、これは次の wolfSSL_read() 呼び出しで取得および復号されます。sz が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_read() は内部読み取りバッファ内で利用可能なバイトを返します。内部読み取りバッファにまだバイトがバッファリングされていない場合、wolfSSL_read() の呼び出しは次のレコードの処理をトリガします。

	Name
int	wolfSSL_peek(WOLFSSL * ssl, void * data, int sz) この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファ data にコピーします。この関数は、内部 SSL セッション受信バッファ内のデータが削除または変更されない点を除いて、wolfSSL_read() と同じです。wolfSSL_read() と同様に、必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_peek() は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズが 16kB の SSL レコードを使用します (最大レコードサイズは、wolfssl/internal.h 内の MAX_RECORD_SIZE 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号できるようになる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_peek() の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。内部 wolfSSL 受信バッファには、まだ復号されていない追加データが待機している可能性があります。これは次の wolfSSL_peek() または wolfSSL_read() 呼び出しで取得および復号されます。sz が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_peek() は内部読み取りバッファ内で利用可能なバイトを返します。内部読み取りバッファにまだバイトがバッファリングされていない場合、wolfSSL_peek() の呼び出しは次のレコードの処理をトリガします。
int	wolfSSL_accept(WOLFSSL * ssl) この関数はサーバ側で呼び出され、SSL クライアントが SSL/TLS ハンドシェイクを開始するのを待機します。この関数が呼び出されると、下層の通信チャネルはすでに設定されています。wolfSSL_accept() は、ブロッキングとノンブロッキングの両方の I/O で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_accept() は、下層の I/O が wolfSSL_accept がハンドシェイクを継続するために必要な要求を満たせなくなった時点で返されます。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、データの読み取りが可能になったら wolfSSL_accept() の呼び出しを繰り返す必要があります。wolfSSL は中断した箇所から再開します。ノンブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。下層の I/O がブロッキングの場合、wolfSSL_accept() は、ハンドシェイクが完了するか、エラーが発生するまで返されません。

	Name
int	wolfDTLS_accept_stateless (WOLFSSL * ssl) この関数はサーバ側で呼び出され、SSL クライアントが DTLS ハンドシェイクを開始するのをステートレスに待機します。
int	wolfSSL_send (WOLFSSL * ssl, const void * data, int sz, int flags) この関数は、指定されたフラグを使用して、バッファ data から sz バイトを SSL 接続 ssl に書き込みます。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_send() は SSL/TLS セッションをネゴシエートします。wolfSSL_send() は、ブロッキング I/O とノンブロッキング I/O の両方で動作します。基盤となる I/O がノンブロッキングの場合、基盤となる I/O が wolfSSL_send の継続に必要な要求を満たせなかった場合、wolfSSL_send() は戻ります。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、基盤となる I/O の準備ができたときに、wolfSSL_send() の呼び出しを繰り返す必要があります。基盤となる I/O がブロッキングの場合、wolfSSL_send() は、サイズ sz のバッファ data が完全書き込まれるか、エラーが発生するまで戻りません。

	Name
int	wolfSSL_recv (WOLFSSL * ssl, void * data, int sz, int flags) この関数は、指定されたフラグを使用して、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファ data に読み込みます。読み取られたバイトは、内部受信バッファから削除されます。この関数は、基盤となる読み取り操作の recv フラグをアプリケーションが設定できる点を除いて、wolfSSL_read() と同じです。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_recv() は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズ 16kB の SSL レコードを使用します (最大レコードサイズは、/wolfssl/internal.h の MAX_RECORD_SIZE 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号する前に、内部で SSL レコード全体を読み取る必要があります。このため、wolfSSL_recv() の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。まだ復号されていない追加データが内部 wolfSSL 受信バッファで待機している可能性があり、次の wolfSSL_recv() 呼び出しで取得および復号されます。sz が内部読み取りバッファのバイト数より大きい場合、SSL_recv() は内部読み取りバッファで利用可能なバイトを返します。内部読み取りバッファにまだバッファリングされているバイトがない場合、wolfSSL_recv() の呼び出しは次のレコードの処理をトリガーします。
int	wolfSSL_get_alert_history (WOLFSSL * ssl, WOLFSSL_ALERT_HISTORY * h) この関数はアラート履歴を取得します。

	Name
WOLFSSL_SESSION *	wolfSSL_get_session (WOLFSSL * ssl) NO_SESSION_CACHE_REF が定義されている場合、この関数は ssl で使用されている現在のセッション (WOLFSSL_SESSION) へのポインタを返します。この関数は、WOLFSSL_SESSION オブジェクトへの非永続的なポインタを返します。返されたポインタは、wolfSSL_free が呼び出されたときに解放されます。この呼び出しは、現在のセッションを検査または変更するためにのみ使用する必要があります。セッション再開には、wolfSSL_get1_session() を使用することをお勧めします。後方互換性のため、NO_SESSION_CACHE_REF が定義されていない場合、この関数はローカルキャッシュに格納されている永続的なセッションオブジェクトポインタを返します。キャッシュサイズは有限であり、アプリケーションが wolfSSL_set_session() を呼び出すまでに、別の ssl 接続によってセッションオブジェクトが上書きされるリスクがあります。アプリケーションで NO_SESSION_CACHE_REF を定義し、セッション再開に wolfSSL_get1_session() を使用することをお勧めします。
void	wolfSSL_flush_sessions (WOLFSSL_CTX * ctx, long tm) この関数は、期限切れのセッションをセッションキャッシュからフラッシュします。時刻 tm は、時刻比較に使用されます。wolfSSL は現在セッションに静的テーブルを使用しているため、フラッシュは不要です。そのため、この関数は現在スタブにすぎません。この関数は、wolfSSL が OpenSSL 互換レイヤーでコンパイルされている場合、OpenSSL 互換性 (SSL_flush_sessions) を提供します。
int	wolfSSL_GetSessionIndex (WOLFSSL * ssl) この関数は、WOLFSSL 構造体のセッションインデックスを取得します。
int	wolfSSL_GetSessionAtIndex (int index, WOLFSSL_SESSION * session) この関数はセッションキャッシュの指定されたインデックスにあるセッションを取得し、メモリにコピーします。WOLFSSL_SESSION 構造体はセッション情報を保持します。
WOLFSSL_X509_CHAIN *	wolfSSL_SESSION_get_peer_chain (WOLFSSL_SESSION * session) WOLFSSL_SESSION 構造体からピア証明書チェーンを返します。
int	wolfSSL_pending (WOLFSSL * ssl) この関数は SSL オブジェクト内でバッファリングされ、wolfSSL_read() によって読み取り可能な利用可能なバイト数を返します。
int	wolfSSL_save_session_cache (const char * fname) この関数は、セッションキャッシュをファイルに永続化します。追加のメモリ使用量のため、memsave は使用しません。

	Name
int	wolfSSL_restore_session_cache (const char * fname) この関数は、永続的なセッションキャッシュをファイルから復元します。追加のメモリ使用量のため、memstore は使用しません。
int	wolfSSL_memsave_session_cache (void * mem, int sz) この関数は、セッションキャッシュをメモリに永続化します。
int	wolfSSL_memrestore_session_cache (const void * mem, int sz) この関数は、永続的なセッションキャッシュをメモリから復元します。
int	wolfSSL_get_session_cache_memsize (void) この関数は、セッションキャッシュの保存バッファがどのくらい大きくあるべきかを返します。
int	wolfSSL_session_reused (WOLFSSL * ssl) この関数は options 構造体の resuming メンバを返します。このフラグはセッションを再利用するかどうかを示します。再利用しない場合、新しいセッションを確立する必要があります。
const char *	wolfSSL_get_version (WOLFSSL * ssl) 使用されている SSL バージョンを文字列として返します。
int	wolfSSL_get_current_cipher_suite (WOLFSSL * ssl) ssl セッションが使用している現在の暗号スイートを返します。
WOLFSSL_CIPHER *	wolfSSL_get_current_cipher (WOLFSSL * ssl) この関数は ssl セッション内の現在の暗号へのポインタを返します。
const char *	wolfSSL_CIPHER_get_name (const WOLFSSL_CIPHER * cipher) この関数は SSL オブジェクト内の暗号スイートを利用可能なスイートと照合し、文字列表現を返します。
const char *	wolfSSL_get_cipher (WOLFSSL *) この関数は SSL オブジェクト内の暗号スイートを利用可能なスイートと照合します。
int	wolfSSL_BIO_get_mem_data (WOLFSSL_BIO * bio, void * p) これは、内部メモリバッファの先頭にバイトポインタを設定するために使用されます。
long	wolfSSL_BIO_set_fd (WOLFSSL_BIO * b, int fd, int flag) bio が使用するファイルディスクリプタを設定します。
int	wolfSSL_BIO_set_close (WOLFSSL_BIO * b, long flag) クローズフラグを設定します。これは、BIO が解放される際に I/O ストリームをクローズすべきかを示すために使用されます。
WOLFSSL_BIO_METHOD *	wolfSSL_BIO_s_socket (void) これは、BIO_SOCKET タイプの WOLFSSL_BIO_METHOD を取得するために使用されます。

	Name
int	wolfSSL_BIO_set_write_buf_size (WOLFSSL_BIO * b, long size) これは、WOLFSSL_BIO の書き込みバッファのサイズを設定するために使用されます。書き込みバッファが以前に設定されていた場合、この関数はサイズをリセットする際にそれを解放します。これは、読み取りと書き込みのインデックスを 0 にリセットする点で wolfSSL_BIO_reset に似ています。
int	wolfSSL_BIO_make_bio_pair (WOLFSSL_BIO * b1, WOLFSSL_BIO * b2) これは、2 つの bio をペアにするために使用されます。ペアになった bio は双方向パイプのように動作し、一方への書き込みはもう一方から読み取ることができ、その逆も同様です。両方の bio が同じスレッドにあることが期待されます。この関数はスレッドセーフではありません。2 つの bio のうちの 1 つを解放すると、両方のペアが解除されます。いずれかの bio に書き込みバッファサイズが以前に設定されていなかった場合、ペアになる前にデフォルトサイズの 17000(WOLFSSL_BIO_SIZE) に設定されます。
int	wolfSSL_BIO_ctrl_reset_read_request (WOLFSSL_BIO * b) これは、読み取り要求フラグを 0 に戻すために使用されます。
int	wolfSSL_BIO_nread0 (WOLFSSL_BIO * bio, char ** buf) これは、読み取り用のバッファポインタを取得するために使用されます。 wolfSSL_BIO_nread とは異なり、内部読み取りインデックスは関数呼び出しから返される数だけ進められません。返される値を超えて読み取ると、配列の境界外を読み取る結果になる可能性があります。
int	wolfSSL_BIO_nread (WOLFSSL_BIO * bio, char ** buf, int num) これは、読み取り用のバッファポインタを取得するために使用されます。内部読み取りインデックスは、関数呼び出しから返される数だけ進められ、buf は読み取るバッファの先頭を指します。読み取りバッファ内のバイト数が num で要求された値より少ない場合、より小さい値が返されます。返される値を超えて読み取ると、配列の境界外を読み取る結果になる可能性があります。
int	wolfSSL_BIO_nwrite (WOLFSSL_BIO * bio, char ** buf, int num) 関数が返す数だけのバイトを書き込むためのバッファへのポインタを取得します。返される値よりも多くのバイトを返されたポインタに書き込むと、境界外への書き込みになる可能性があります。
int	wolfSSL_BIO_reset (WOLFSSL_BIO * bio) bio を初期状態にリセットします。例えば、BIO_BIO タイプの場合、これは読み取りと書き込みのインデックスをリセットします。

	Name
int	wolfSSL_BIO_seek (WOLFSSL_BIO * bio, int ofs) この関数は、ファイルポインタを指定されたオフセットに調整します。これはファイルの先頭からのオフセットです。
int	wolfSSL_BIO_write_filename (WOLFSSL_BIO * bio, char * name) ファイルを設定し、書き込むために使用されます。ファイル内の既存データを上書きし、bio が解放される際にファイルを閉じるよう設定されます。
long	wolfSSL_BIO_set_mem_eof_return (WOLFSSL_BIO * bio, int v) ファイル終端値を設定するために使用されます。一般的な値は-1 で、期待される正の値と混同しないようにします。
long	wolfSSL_BIO_get_mem_ptr (WOLFSSL_BIO * bio, WOLFSSL_BUF_MEM ** m) WOLFSSL_BIO メモリポインタの getter 関数です。
const char *	wolfSSL_lib_version (void) この関数は現在のライブラリバージョンを返します。
word32	wolfSSL_lib_version_hex (void) この関数は現在のライブラリバージョンを 16 進表記で返します。
int	wolfSSL_negotiate (WOLFSSL * ssl) SSL メソッドの側面に基づいて実際の接続または受け入れを実行します。クライアント側から呼び出された場合は wolfSSL_connect() が実行され、サーバー側から呼び出された場合は wolfSSL_accept() が実行されます。
int	wolfSSL_connect_cert (WOLFSSL * ssl) この関数はクライアント側で呼び出され、ピアの証明書チェーンを取得するのに十分な長さだけサーバーとの SSL/TLS ハンドシェイクを開始します。この関数が呼び出されるとき、基礎となる通信チャネルはすでに設定されています。 wolfSSL_connect_cert() は、ブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、wolfSSL_connect_cert() は、基礎となる I/O が wolfSSL_connect_cert() がハンドシェイクを続行するために必要な処理を満たせないときに戻ります。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。呼び出しプロセスは、基礎となる I/O が準備できたときに wolfSSL_connect_cert() の呼び出しを繰り返す必要があり、wolfSSL は中断した場所から再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件をチェックできます。基礎となる I/O がブロッキング I/O の場合、wolfSSL_connect_cert() はピアの証明書チェーンが受信されたときにのみ戻ります。

	Name
int	wolfSSL_writev (WOLFSSL * ssl, const struct iovec * iov, int iovcnt) writev のセマンティクスをシミュレートしますが、SSL_write() の動作とフロント追加が小さい場合があるため、実際には一度にブロック単位では行いません。writev を使用するソフトウェアへの移植を容易にします。
unsigned char	wolfSSL_SNI_Status (WOLFSSL * ssl, unsigned char type) この関数は SNI オブジェクトのステータスを取得します。
int	wolfSSL_UseSecureRenegotiation (WOLFSSL * ssl) この関数は、提供された WOLFSSL 構造体に対して安全な再ネゴシエーションを強制します。これは推奨されません。
int	wolfSSL_Rehandshake (WOLFSSL * ssl) この関数は安全な再ネゴシエーションハンドシェイクを実行します。wolfSSL はこの機能を推奨していないため、これはユーザが強制するものです。
int	wolfSSL_UseSessionTicket (WOLFSSL * ssl) 提供された WOLFSSL 構造体にセッションチケットを使用するよう強制します。定数 HAVE_SESSION_TICKET が定義されており、定数 NO_WOLFSSL_CLIENT が定義されていない必要があります。
int	wolfSSL_get_SessionTicket (WOLFSSL * ssl, unsigned char * buf, word32 * bufSz) この関数は、Session 構造体の ticket メンバをバッファにコピーします。buf が NULL で bufSz が非 NULL の場合、bufSz はチケット長に設定されます。
int	wolfSSL_set_SessionTicket (WOLFSSL * ssl, const unsigned char * buf, word32 bufSz) この関数は、WOLFSSL 構造体内の WOLFSSL_SESSION 構造体の ticket メンバを設定します。関数に渡されたバッファはメモリにコピーされます。
int	wolfSSL_PrintSessionStats (void) この関数は、セッションの統計情報を出力します。
int	wolfSSL_get_session_stats (unsigned int * active, unsigned int * total, unsigned int * peak, unsigned int * maxSessions) この関数は、セッションの統計情報を取得します。
long	wolfSSL_BIO_set_fp (WOLFSSL_BIO * bio, XFILE fp, int c) これは、BIO の内部ファイルポインタを設定するために使用されます。
long	wolfSSL_BIO_get_fp (WOLFSSL_BIO * bio, XFILE * fp) これは、BIO の内部ファイルポインタを取得するために使用されます。

	Name
size_t	wolfSSL_BIO_ctrl_pending (WOLFSSL_BIO * b) 読み取り保留中のバイト数を取得します。BIO タイプが BIO_BIO の場合、ペアから読み取るバイト数です。BIO が SSL オブジェクトを含む場合、SSL オブジェクトからの保留中のデータです (wolfSSL_pending(ssl))。BIO_MEMORY タイプの場合、メモリバッファのサイズを返します。
int	wolfSSL_set_jobject (WOLFSSL * ssl, void * objPtr) この関数は、WOLFSSL 構造体の jobjectRef メンバを設定します。
void *	wolfSSL_get_jobject (WOLFSSL * ssl) この関数は、WOLFSSL 構造体の jobjectRef メンバを返します。
int	wolfSSL_connect (WOLFSSL * ssl) この関数はクライアント側で呼び出され、サーバとの SSL/TLS ハンドシェイクを開始します。この関数が呼び出されると、基礎となる通信チャネルはすでに設定されています。wolfSSL_connect() は、ブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、wolfSSL_connect() は、基礎となる I/O が wolfSSL_connect() がハンドシェイクを続行するために必要とするものを満たすことができないときに返されます。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE を返します。呼び出しプロセスは、基礎となる I/O の準備ができたときに wolfSSL_connect() の呼び出しを繰り返す必要があります。wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキングの場合、wolfSSL_connect() はハンドシェイクが完了するかエラーが発生するまで返されません。wolfSSL は、OpenSSL とは異なるアプローチで証明書検証を行います。クライアントのデフォルトポリシーはサーバを検証することです。つまり、サーバを検証するための CA を読み込まない場合、接続エラー、検証不可 (_155) が発生します。サーバの検証が失敗しても SSL_connect が成功し、セキュリティを低下させる OpenSSL の動作を模倣したい場合は、SSL_new() を呼び出す前に SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); を呼び出すことでこれを実現できます。ただし、推奨されません。

	Name
int	wolfSSL_update_keys (WOLFSSL * ssl) この関数は、鍵のロールオーバーを強制するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。KeyUpdate メッセージがピアに送信され、暗号化用の新しい鍵が計算されます。ピアは KeyUpdate メッセージを送り返し、その後新しい復号鍵が計算されます。この関数は、ハンドシェイクが完了した後にのみ呼び出すことができます。
int	wolfSSL_key_update_response (WOLFSSL * ssl, int * required) この関数は、鍵のロールオーバーが進行中かどうかを判断するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。wolfSSL_update_keys() が呼び出されると、KeyUpdate メッセージが送信され、暗号化鍵が更新されます。復号鍵は、応答を受信したときに更新されます。
int	wolfSSL_request_certificate (WOLFSSL * ssl) この関数は、TLS v1.3 クライアントからクライアント証明書を要求します。これは、Web サーバがクライアント認証を必要とするページと必要としないページの両方を提供している場合に便利です。接続上で最大 256 回の要求を送信できます。

	Name
int	wolfSSL_connect_TLSv13(WOLFSSL * ssl) この関数はクライアント側で呼び出され、サーバとの TLS v1.3 ハンドシェイクを開始します。この関数が呼び出されるとき、基礎となる通信チャネルはすでに設定されています。wolfSSL_connect() はブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、基礎となる I/O が wolfSSL_connect() がハンドシェイクを続行するために必要なものを満たせない場合、wolfSSL_connect() は返されます。この場合、wolfSSL_get_error() を呼び出すと SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、基礎となる I/O の準備ができたときに wolfSSL_connect() の呼び出しを繰り返す必要があり、wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキング I/O の場合、wolfSSL_connect() はハンドシェイクが完了するかエラーが発生するまで返されません。wolfSSL は証明書検証に OpenSSL とは異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバを検証することです。つまり、サーバを検証するための CA をロードしない場合、接続エラー「検証できません (155)」が発生します。サーバの検証が失敗しても SSL_connect が成功するという OpenSSL の動作を模倣し、セキュリティを低下させたい場合は、SSL_new() を呼び出す前に SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0) を呼び出すことでこれを行うことができます。ただし、これは推奨されません。

	Name
	wolfSSL_accept_TLSv13(WOLFSSL * ssl) この関数はサーバ側で呼び出され、SSL/TLS クライアントが SSL/TLS ハンドシェイクを開始するのを待ちます。この関数が呼び出されるとき、基礎となる通信チャンネルはすでに設定されています。 wolfSSL_accept() はブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、基礎となる I/O が wolfSSL_accept() がハンドシェイクを続行するために必要なものを満たせない場合、wolfSSL_accept() は返されます。この場合、wolfSSL_get_error() を呼び出すと SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、データが読み取り可能になったときに wolfSSL_accept の呼び出しを繰り返す必要があります。wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキング I/O の場合、wolfSSL_accept() はハンドシェイクが完了するかエラーが発生するまで返されません。TLS v1.3 接続を期待する場合にこの関数を呼び出してください。ただし、古いバージョンの ClientHello メッセージもサポートされています。
int	wolfSSL_write_early_data(WOLFSSL * ssl, const void * data, int sz, int * outSz) この関数は再開時にサーバーにアーリーデータを書き込みます。wolfSSL_connect() または wolfSSL_connect_TLSv13() の前にこの関数を呼び出してください。この関数はクライアントでのみ使用されます。
int	wolfSSL_read_early_data(WOLFSSL * ssl, void * data, int sz, int * outSz) この関数は再開時にクライアントからのアーリーデータを読み取ります。クライアントを受け入れ、ハンドシェイクでアーリーデータを読み取るには、wolfSSL_accept() または wolfSSL_accept_TLSv13() の代わりにこの関数を呼び出します。wolfSSL_is_init_finished() が true を返すまで関数を呼び出す必要があります。アーリーデータは複数のメッセージでクライアントから送信される場合があります。アーリーデータがない場合、ハンドシェイクは通常通り処理されます。この関数はサーバーでのみ使用されます。

	Name
int	wolfSSL_inject (WOLFSSL * ssl, const void * data, int sz) この関数は WOLFSSL オブジェクトにデータを注入するために呼び出されます。これは、データを単一の場所から読み取り、複数の接続に分割する必要がある場合に便利です。呼び出し元は wolfSSL_read() を呼び出して WOLFSSL オブジェクトから平文データを抽出する必要があります。
void *	wolfSSL_GetIOReadCtx (WOLFSSL * ssl) この関数は WOLFSSL 構造体の IOCB_ReadCtx メンバーを返します。
void *	wolfSSL_GetIOWriteCtx (WOLFSSL * ssl) この関数は WOLFSSL 構造体の IOCB_WriteCtx メンバーを返します。
void	wolfSSL_SetIO_NetX (WOLFSSL * ssl, NX_TCP_SOCKET * nxsocket, ULONG waitoption) この関数は、WOLFSSL 構造体内の nxCtx 構造体の nxSocket および nxWait メンバーを設定します。

A.5.2 Functions Documentation

A.5.2.1 function wolfSSL_get_verify_depth

```
long wolfSSL_get_verify_depth(
    WOLFSSL * ssl
)
```

この関数は、有効なセッション (つまり、非 NULL のセッションオブジェクト (ssl) が存在する場合) に対して許可される最大チェーン深度を返します。デフォルトは 9 です。

Parameters:

- **ssl** **wolfSSL_new()**を使用して作成された WOLFSSL 構造体へのポインタ。

See: **wolfSSL_CTX_get_verify_depth**

Return:

- WOLFSSL 構造体が NULL でない場合は MAX_CHAIN_DEPTH が返されます。デフォルトでは値は 9 です。
- WOLFSSL 構造体が NULL の場合は BAD_FUNC_ARG が返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
long sslDep = wolfSSL_get_verify_depth(ssl);

if(sslDep > EXPECTED){
    // 検証された深度は期待値よりも大きい
} else {
    // 検証された深度は期待値以下
}
```

A.5.2.2 function wolfSSL_get_cipher_list

```
char * wolfSSL_get_cipher_list(  
    int priority  
)
```

渡された優先度レベルでの暗号の名前を取得します。

Parameters:

- **priority** 暗号の優先度レベルを表す整数。

See:

- [wolfSSL_CIPHER_get_name](#)
- [wolfSSL_get_current_cipher](#)

Return:

- string 成功
- 0 優先度が範囲外または無効です。

Example

```
printf("The cipher at 1 is %s", wolfSSL_get_cipher_list(1));
```

A.5.2.3 function wolfSSL_get_ciphers

```
int wolfSSL_get_ciphers(  
    char * buf,  
    int len  
)
```

この関数は、wolfSSL で有効になっている暗号を取得します。

Parameters:

- **buf** バッファを表す char ポインタ。
- **len** バッファの長さ。

See:

- GetCipherNames
- [wolfSSL_get_cipher_list](#)
- ShowCiphers

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- BAD_FUNC_ARG buf パラメータが NULL または len 引数がゼロ以下の場合に返されます。
- BUFFER_E バッファが十分に大きくなくオーバーフローする場合に返されます。

Example

```
static void ShowCiphers(void){  
    char* ciphers;  
    int ret = wolfSSL_get_ciphers(ciphers, (int)sizeof(ciphers));  
  
    if(ret == SSL_SUCCESS){  
        printf("%s\n", ciphers);  
    }  
}
```

A.5.2.4 function wolfSSL_get_cipher_name

```
const char * wolfSSL_get_cipher_name(  
    WOLFSSL * ssl  
)
```

この関数は、引数を wolfSSL_get_cipher_name_internal に渡すことによって、DHE-RSA 形式で暗号名を取得します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal

Return:

- string この関数は、マッチした暗号スイートの文字列表現を返します。
- NULL エラーまたは暗号が見つかりません。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );  
WOLFSSL* ssl = wolfSSL_new(ctx);  
...  
char* cipher = wolfSSL_get_cipher_name(ssl);  
  
if(cipher == NULL){  
    // 暗号スイートがマッチしませんでした  
} else {  
    // 暗号スイートがマッチしました  
    printf("%s\n", cipherS);  
}
```

A.5.2.5 function wolfSSL_get_fd

```
int wolfSSL_get_fd(  
    const WOLFSSL * ssl  
)
```

この関数は、SSL 接続の入力機能として使用される読み取りファイル記述子 (fd) を返します。通常、これはソケットファイル記述子になります。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。

See:

- wolfSSL_set_fd
- wolfSSL_set_read_fd
- wolfSSL_set_write_fd

Return: fd 成功した場合、この関数は SSL セッションファイルディスクリプタを返します。

Example

```
int sockfd;  
WOLFSSL* ssl = 0;
```

```
...
sockfd = wolfSSL_get_fd(ssl);
...
```

A.5.2.6 function wolfSSL_get_wfd

```
int wolfSSL_get_wfd(
    const WOLFSSL * ssl
)
```

この関数は、SSL 接続の出力機能として使用される書き込みファイルディスクリプタ (fd) を返します。通常はソケットファイルディスクリプタになります。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) で作成された SSL セッションへのポインタ。

See:

- [wolfSSL_set_fd](#)
- [wolfSSL_set_read_fd](#)
- [wolfSSL_set_write_fd](#)

Return: fd 成功した場合、この関数は SSL セッションファイルディスクリプタを返します。

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_wfd(ssl);
...
```

A.5.2.7 function wolfSSL_get_using_nonblock

```
int wolfSSL_get_using_nonblock(
    WOLFSSL *
)
```

この関数により、アプリケーションは wolfSSL がノンブロッキング I/O を使用しているかどうかを判定できます。wolfSSL がノンブロッキング I/O を使用している場合、この関数は 1 を返し、それ以外の場合は 0 を返します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキングソケットと共に使用する場合は、[wolfSSL_set_using_nonblock\(\)](#) を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、recvfrom 呼び出しがブロックすることを意味すると認識できます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) で作成された SSL セッションへのポインタ。

See: [wolfSSL_set_session](#)

Return:

- 0 下層の I/O がブロッキングです。
- 1 下層の I/O がノンブロッキングです。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
```

```
ret = wolfSSL_get_using_nonblock(ssl);
if (ret == 1) {
    // 下層の I/O はノンブロッキング
}
...
```

A.5.2.8 function wolfSSL_write

```
int wolfSSL_write(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

この関数は、バッファ data から sz バイトを SSL 接続 ssl に書き込みます。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_write() は SSL/TLS セッションをネゴシエートします。(D)TLSv1.3 を使用していて early data 機能がコンパイルされている場合、この関数はデータ送信が可能になるまでハンドシェイクを進めます。次の wolfSSL_Connect()、wolfSSL_Accept()、wolfSSL_read() の呼び出しでハンドシェイクが完了します。wolfSSL_write() は、ブロッキングとノンブロッキングの両方の I/O で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_write() は、下層の I/O が wolfSSL_write() を継続するために必要な要求を満たせなくなった時点で返されます。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、下層の I/O の準備ができたなら、wolfSSL_write() の呼び出しを繰り返す必要があります。下層の I/O がブロッキングの場合、wolfSSL_write() は、サイズ sz のバッファデータが完全に書き込まれるか、エラーが発生するまで返されません。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **data** ピアに送信されるデータバッファ。
- **sz** ピアに送信するデータ (data) のサイズ (バイト単位)。

See:

- wolfSSL_send
- wolfSSL_read
- wolfSSL_recv

Return:

- > 0 成功時に書き込まれたバイト数。
- 0 失敗時に返されます。具体的なエラーコードについては、wolfSSL_get_error() を呼び出してください。
- SSL_FATAL_ERROR エラーが発生した場合、またはノンブロッキングソケットを使用している場合に SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーを受け取り、アプリケーションが wolfSSL_write() を再度呼び出す必要がある場合に返されます。具体的なエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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() が失敗、wolfSSL_get_error() を呼び出す
}
```

A.5.2.9 function wolfSSL_read

```
int wolfSSL_read(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファ data に読み込みます。読み取られたバイトは内部受信バッファから削除されます。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_read() は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズが 16kB の SSL レコードを使用します (最大レコードサイズは、wolfssl/internal.h 内の MAX_RECORD_SIZE 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号できるようになる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_read() の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。内部 wolfSSL 受信バッファには、まだ復号されていない追加データが待機している可能性があり、これは次の wolfSSL_read() 呼び出しで取得および復号されます。sz が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_read() は内部読み取りバッファ内で利用可能なバイトを返します。内部読み取りバッファにまだバイトがバッファリングされていない場合、wolfSSL_read() の呼び出しは次のレコードの処理をトリガします。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **data** wolfSSL_read() が読み取ったデータを格納するバッファ。
- **sz** data に読み込むバイト数。

See:

- wolfSSL_recv
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- > 0 成功時に読み取られたバイト数。
- 0 失敗時に返されます。これは、クリーンシャットダウン (close notify アラート) またはピアが接続を閉じたことが原因である可能性があります。具体的なエラーコードについては、wolfSSL_get_error() を呼び出してください。
- SSL_FATAL_ERROR エラーが発生した場合、またはノンブロッキングソケットを使用している場合に SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーを受け取り、アプリケーションが wolfSSL_read() を再度呼び出す必要がある場合に返されます。具体的なエラーコードを取得するには、wolfSSL_get_error() を使用してください。

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_read(ssl, reply, sizeof(reply));
if (input > 0) {
    // バッファ"reply" に"input" バイトが返された
}
```

wolfSSL_read() のより完全な例については、wolfSSL の例 (client、server、echoclient、
↪ echoserver) を参照してください。

A.5.2.10 function wolfSSL_peek

```
int wolfSSL_peek(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファ data にコピーします。この関数は、内部 SSL セッション受信バッファ内のデータが削除または変更されない点を除いて、wolfSSL_read() と同じです。wolfSSL_read() と同様に、必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_peek() は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズが 16kB の SSL レコードを使用します (最大レコードサイズは、wolfssl/internal.h 内の MAX_RECORD_SIZE 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号できるようになる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_peek() の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。内部 wolfSSL 受信バッファには、まだ復号されていない追加データが待機している可能性があり、これは次の wolfSSL_peek() または wolfSSL_read() 呼び出しで取得および復号されます。sz が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_peek() は内部読み取りバッファ内で利用可能なバイトを返します。内部読み取りバッファにまだバイトがバッファリングされていない場合、wolfSSL_peek() の呼び出しは次のレコードの処理をトリガします。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **data** wolfSSL_peek() が読み取ったデータを格納するバッファ。
- **sz** data に読み込むバイト数。

See: [wolfSSL_read](#)

Return:

- > 0 成功時に読み取られたバイト数。
- 0 失敗時に返されます。これは、クリーンシャットダウン (close notify アラート) またはピアが接続を閉じたことが原因である可能性があります。具体的なエラーコードについては、wolfSSL_get_error() を呼び出してください。
- SSL_FATAL_ERROR エラーが発生した場合、またはノンブロッキングソケットを使用している場合に SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーを受け取り、アプリケーションが wolfSSL_peek() を再度呼び出す必要がある場合に返されます。具体的なエラーコードを取得するには、wolfSSL_get_error() を使用してください。

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_peek(ssl, reply, sizeof(reply));
if (input > 0) {
    // バッファ"reply" に"input" バイトが返された
}
```

A.5.2.11 function wolfSSL_accept

```
int wolfSSL_accept(
```

```
WOLFSSL * ssl
)
```

この関数はサーバ側で呼び出され、SSL クライアントが SSL/TLS ハンドシェイクを開始するのを待機します。この関数が呼び出されるとき、下層の通信チャネルはすでに設定されています。wolfSSL_accept() は、ブロッキングとノンブロッキングの両方の I/O で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_accept() は、下層の I/O が wolfSSL_accept がハンドシェイクを継続するために必要な要求を満たせなくなった時点で返されます。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、データの読み取りが可能になったら wolfSSL_accept() の呼び出しを繰り返す必要があります。wolfSSL は中断した箇所から再開します。ノンブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。下層の I/O がブロッキングの場合、wolfSSL_accept() は、ハンドシェイクが完了するか、エラーが発生するまで返されません。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_connect

Return:

- SSL_SUCCESS 成功時。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出してください。

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));
}
```

A.5.2.12 function wolfDTLS_accept_stateless

```
int wolfDTLS_accept_stateless(
    WOLFSSL * ssl
)
```

この関数はサーバ側で呼び出され、SSL クライアントが DTLS ハンドシェイクを開始するのをステートレスに待機します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_accept
- wolfSSL_get_error
- wolfSSL_connect

Return:

- WOLFSSL_SUCCESS 有効なクッキーを含む ClientHello が受信されました。wolfSSL_accept() で接続を続行できます。
- WOLFSSL_FAILURE I/O 層が WANT_READ を返しました。これは、読み取るデータがなくノンブロッキングソケットを使用しているか、クッキーリクエストを送信して応答を待っているためです。ユーザは、I/O 層でデータが利用可能になった後、wolfDTLS_accept_stateless を再度呼び出す必要があります。
- WOLFSSL_FATAL_ERROR 致命的なエラーが発生しました。ssl オブジェクトを解放して再割り当てしてから続行する必要があります。

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
...
do {
    ret = wolfDTLS_accept_stateless(ssl);
    if (ret == WOLFSSL_FATAL_ERROR)
        // wolfSSL_free() と wolfSSL_new() で ssl オブジェクトを再割り当て
} 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));
}
```

A.5.2.13 function wolfSSL_send

```
int wolfSSL_send(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int flags
)
```

この関数は、指定されたフラグを使用して、バッファ data から sz バイトを SSL 接続 ssl に書き込みます。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_send() は SSL/TLS セッションをネゴシエートします。wolfSSL_send() は、ブロッキング I/O とノンブロッキング I/O の両方で動作します。基盤となる I/O がノンブロッキングの場合、基盤となる I/O が wolfSSL_send の継続に必要な要求を満たせなかった場合、wolfSSL_send() は戻ります。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、基盤となる I/O の準備ができたときに、wolfSSL_send() の呼び出しを繰り返す必要があります。基盤となる I/O がブロッキングの場合、wolfSSL_send() は、サイズ sz のバッファ data が完全に書き込まれるか、エラーが発生するまで戻りません。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **data** ピアに送信するデータバッファ。
- **sz** ピアに送信するデータのサイズ (バイト単位)。
- **flags** 基盤となる送信操作に使用する送信フラグ。

See:

- wolfSSL_write
- wolfSSL_read

- `wolfSSL_recv`

Return:

- `> 0` は、成功時に書き込まれたバイト数です。
- `0` は、失敗時に返されます。具体的なエラーコードについては、`wolfSSL_get_error()` を呼び出してください。
- `SSL_FATAL_ERROR` は、エラーが発生した場合、またはノンブロッキングソケットを使用している場合に `SSL_ERROR_WANT_READ` または `SSL_ERROR_WANT_WRITE` エラーが発生し、アプリケーションが `wolfSSL_send()` を再度呼び出す必要がある場合に返されます。具体的なエラーコードを取得するには、`wolfSSL_get_error()` を使用してください。

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() が失敗しました
}
```

A.5.2.14 function wolfSSL_recv

```
int wolfSSL_recv(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int flags
)
```

この関数は、指定されたフラグを使用して、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファ data に読み込みます。読み取られたバイトは、内部受信バッファから削除されます。この関数は、基盤となる読み取り操作の `recv` フラグをアプリケーションが設定できる点を除いて、`wolfSSL_read()` と同じです。必要に応じて、`wolfSSL_connect()` または `wolfSSL_accept()` によってハンドシェイクがまだ実行されていない場合、`wolfSSL_recv()` は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズ 16kB の SSL レコードを使用します (最大レコードサイズは、`/wolfssl/internal.h` の `MAX_RECORD_SIZE` 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号する前に、内部で SSL レコード全体を読み取る必要があります。このため、`wolfSSL_recv()` の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。まだ復号されていない追加データが内部 wolfSSL 受信バッファで待機している可能性があり、次の `wolfSSL_recv()` 呼び出しで取得および復号されます。sz が内部読み取りバッファのバイト数より大きい場合、`SSL_recv()` は内部読み取りバッファで利用可能なバイトを返します。内部読み取りバッファにまだバッファリングされているバイトがない場合、`wolfSSL_recv()` の呼び出しは次のレコードの処理をトリガーします。

Parameters:

- **ssl** `wolfSSL_new()` で作成された SSL セッションへのポインタ。
- **data** `wolfSSL_recv()` が読み取ったデータを配置するバッファ。
- **sz** data に読み込むバイト数。
- **flags** 基盤となる受信操作に使用する受信フラグ。

See:

- `wolfSSL_read`
- `wolfSSL_write`

- `wolfSSL_peek`
- `wolfSSL_pending`

Return:

- `> 0` は、成功時に読み取られたバイト数です。
- `0` は、失敗時に返されます。これは、クリーンな (close notify アラート) シャットダウンによって引き起こされる場合もあれば、単にピアが接続を閉じた場合もあります。具体的なエラーコードについては、`wolfSSL_get_error()` を呼び出してください。
- `SSL_FATAL_ERROR` は、エラーが発生した場合、またはノンブロッキングソケットを使用している場合に `SSL_ERROR_WANT_READ` または `SSL_ERROR_WANT_WRITE` エラーが発生し、アプリケーションが `wolfSSL_recv()` を再度呼び出す必要がある場合に返されます。具体的なエラーコードを取得するには、`wolfSSL_get_error()` を使用してください。

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
int flags = ... ;
...

input = wolfSSL_recv(ssl, reply, sizeof(reply), flags);
if (input > 0) {
    // バッファ"reply" に"input" バイトが返されました
}
```

A.5.2.15 function wolfSSL_get_alert_history

```
int wolfSSL_get_alert_history(
    WOLFSSL * ssl,
    WOLFSSL_ALERT_HISTORY * h
)
```

この関数はアラート履歴を取得します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **h** WOLFSSL 構造体の `alert_history` メンバーの値を保持する WOLFSSL_ALERT_HISTORY 構造体へのポインタ。

See: `wolfSSL_get_error`

Return: `SSL_SUCCESS` は、関数が正常に完了したときに返されます。アラート履歴があってもなくても、いずれの場合も戻り値は `SSL_SUCCESS` です。

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 に ssl->alert_history の内容のコピーが格納されました
```

A.5.2.16 function wolfSSL_get_session

```
WOLFSSL_SESSION * wolfSSL_get_session(
    WOLFSSL * ssl
)
```


NO_SESSION_CACHE_REF が定義されている場合、この関数は ssl で使用されている現在のセッション (WOLFSSL_SESSION) へのポインタを返します。この関数は、WOLFSSL_SESSION オブジェクトへの非永続的なポインタを返します。返されたポインタは、wolfSSL_free が呼び出されたときに解放されます。この呼び出しは、現在のセッションを検査または変更するためにのみ使用する必要があります。セッション再開には、wolfSSL_get1_session() を使用することをお勧めします。後方互換性のため、NO_SESSION_CACHE_REF が定義されていない場合、この関数はローカルキャッシュに格納されている永続的なセッションオブジェクトポインタを返します。キャッシュサイズは有限であり、アプリケーションが wolfSSL_set_session() を呼び出すまでに、別の ssl 接続によってセッションオブジェクトが上書きされるリスクがあります。アプリケーションで NO_SESSION_CACHE_REF を定義し、セッション再開に wolfSSL_get1_session() を使用することをお勧めします。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- pointer 呼び出しが成功した場合、現在の SSL セッションオブジェクトへのポインタを返します。
- NULL は、ssl が NULL の場合、SSL セッションキャッシュが無効になっている場合、wolfSSL がセッション ID を利用できない場合、またはミューテックス関数が失敗した場合に返されます。

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get_session(ssl);
if (session == NULL) {
    // セッションポインタの取得に失敗しました
}
...
```

A.5.2.17 function wolfSSL_flush_sessions

```
void wolfSSL_flush_sessions(
    WOLFSSL_CTX * ctx,
    long tm
)
```

この関数は、期限切れのセッションをセッションキャッシュからフラッシュします。時刻 tm は、時刻比較に使用されます。wolfSSL は現在セッションに静的テーブルを使用しているため、フラッシュは不要です。そのため、この関数は現在スタブにすぎません。この関数は、wolfSSL が OpenSSL 互換レイヤーでコンパイルされている場合、OpenSSL 互換性 (SSL_flush_sessions) を提供します。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **tm** セッション有効期限比較に使用される時刻。

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return: none 戻り値はありません。

Example

```
WOLFSSL_CTX* ssl;
...
wolfSSL_flush_sessions(ctx, time(0));
```

A.5.2.18 function wolfSSL_GetSessionIndex

```
int wolfSSL_GetSessionIndex(
    WOLFSSL * ssl
)
```

この関数は、WOLFSSL 構造体のセッションインデックスを取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_GetSessionAtIndex

Return: int この関数は、WOLFSSL 構造体内の sessionIndex を表す int 型を返します。

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)){
    // インデックス番号が範囲外であり、何かが正しくありません。
}
```

A.5.2.19 function wolfSSL_GetSessionAtIndex

```
int wolfSSL_GetSessionAtIndex(
    int index,
    WOLFSSL_SESSION * session
)
```

この関数はセッションキャッシュの指定されたインデックスにあるセッションを取得し、メモリにコピーします。WOLFSSL_SESSION 構造体はセッション情報を保持します。

Parameters:

- **index** セッションインデックスを表す int 型。
- **session** WOLFSSL_SESSION 構造体へのポインタ。

See:

- UnLockMutex
- LockMutex
- wolfSSL_GetSessionIndex

Return:

- SSL_SUCCESS 関数が正常に実行され、エラーが発生しなかった場合に返されます。
- BAD_MUTEX_E mutex のアンロックまたはロックエラーがあった場合に返されます。
- SSL_FAILURE 関数が正常に実行されなかった場合に返されます。

Example

```
int idx; // セッションを特定するインデックス。
WOLFSSL_SESSION* session; // コピー先のバッファ。
```



```
...
if(wolfSSL_GetSessionAtIndex(idx, session) != SSL_SUCCESS){
    // 失敗ケース。
}

```

A.5.2.20 function wolfSSL_SESSION_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_SESSION_get_peer_chain(
    WOLFSSL_SESSION * session
)
```

WOLFSSL_SESSION 構造体からピア証明書チェーンを返します。

Parameters:

- **session** WOLFSSL_SESSION 構造体へのポインタ。

See:

- [wolfSSL_GetSessionAtIndex](#)
- [wolfSSL_GetSessionIndex](#)
- [AddSession](#)

Return: pointer ピア証明書チェーンを含む WOLFSSL_X509_CHAIN 構造体へのポインタ。

Example

```
WOLFSSL_SESSION* session;
WOLFSSL_X509_CHAIN* chain;
...
chain = wolfSSL_SESSION_get_peer_chain(session);
if(!chain){
    // チェーンがありませんでした。失敗ケース。
}

```

A.5.2.21 function wolfSSL_pending

```
int wolfSSL_pending(
    WOLFSSL * ssl
)
```

この関数は SSL オブジェクト内でバッファリングされ、wolfSSL_read() によって読み取り可能な利用可能なバイト数を返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)で作成された SSL セッションへのポインタ。

See:

- [wolfSSL_recv](#)
- [wolfSSL_read](#)
- [wolfSSL_peek](#)

Return: int この関数は保留中のバイト数を返します。

Example

```
int pending = 0;
WOLFSSL* ssl = 0;
...

```

```
pending = wolfSSL_pending(ssl);  
printf(" バッファリングされ読み取り可能な%dバイトがあります", pending);
```

A.5.2.22 function wolfSSL_save_session_cache

```
int wolfSSL_save_session_cache(  
    const char * fname  
)
```

この関数は、セッションキャッシュをファイルに永続化します。追加のメモリ使用量のため、memsave は使用しません。

Parameters:

- **fname** 書き込み用ファイルを指す定数 char 型ポインタ。

See:

- XFWRITE
- [wolfSSL_restore_session_cache](#)
- [wolfSSL_memrestore_session_cache](#)

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。セッションキャッシュがファイルに書き込まれました。
- SSL_BAD_FILE fname を開くことができないか、またはそれ以外の理由で破損している場合に返されます。
- FWRITE_ERROR XFWRITE がファイルへの書き込みに失敗した場合に返されます。
- BAD_MUTEX_E mutex ロックの失敗があった場合に返されます。

Example

```
const char* fname;  
...  
if(wolfSSL_save_session_cache(fname) != SSL_SUCCESS){  
    // ファイルへの書き込みに失敗しました。  
}
```

A.5.2.23 function wolfSSL_restore_session_cache

```
int wolfSSL_restore_session_cache(  
    const char * fname  
)
```

この関数は、永続的なセッションキャッシュをファイルから復元します。追加のメモリ使用量のため、memstore は使用しません。

Parameters:

- **fname** 読み取られる定数 char 型ポインタファイル入力。

See:

- XFREAD
- XFOPEN

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- SSL_BAD_FILE 関数に渡されたファイルが破損しており、XFOPEN で開くことができなかった場合に返されます。

- FREAD_ERROR ファイルが XFREAD からの読み取りエラーを持っていた場合に返されます。
- CACHE_MATCH_ERROR セッションキャッシュヘッダのマッチングに失敗した場合に返されます。
- BAD_MUTEX_E mutex ロックの失敗があった場合に返されます。

Example

```
const char *fname;
...
if(wolfSSL_restore_session_cache(fname) != SSL_SUCCESS){
    // 失敗ケースです。関数は SSL_SUCCESS を返しませんでした。
}
```

A.5.2.24 function wolfSSL_memsave_session_cache

```
int wolfSSL_memsave_session_cache(
    void * mem,
    int sz
)
```

この関数は、セッションキャッシュをメモリに永続化します。

Parameters:

- **mem** メモリコピー XMEMCPY() の宛先を表す void 型ポインタ。
- **sz** mem のサイズを表す int 型。

See:

- XMEMCPY
- [wolfSSL_get_session_cache_memsize](#)

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。セッションキャッシュがメモリに正常に永続化されました。
- BAD_MUTEX_E mutex ロックエラーがあった場合に返されます。
- BUFFER_E バッファサイズが小さすぎた場合に返されます。

Example

```
void* mem;
int sz; // メモリバッファの最大サイズ。
...
if(wolfSSL_memsave_session_cache(mem, sz) != SSL_SUCCESS){
    // 失敗ケースです。セッションキャッシュをメモリに永続化できませんでした。
}
```

A.5.2.25 function wolfSSL_memrestore_session_cache

```
int wolfSSL_memrestore_session_cache(
    const void * mem,
    int sz
)
```

この関数は、永続的なセッションキャッシュをメモリから復元します。

Parameters:

- **mem** 復元のソースを含む定数 void 型ポインタ。
- **sz** メモリバッファのサイズを表す整数。

See: [wolfSSL_save_session_cache](#)

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- BUFFER_E メモリバッファが小さすぎる場合に返されます。
- BAD_MUTEX_E セッションキャッシュの mutex ロックに失敗した場合に返されます。
- CACHE_MATCH_ERROR セッションキャッシュヘッダのマッチングに失敗した場合に返されます。

Example

```
const void* memoryFile;
int szMf;
...
if(wolfSSL_memrestore_session_cache(memoryFile, szMf) != SSL_SUCCESS){
    // 失敗ケースです。SSL_SUCCESS が返されませんでした。
}
```

A.5.2.26 function wolfSSL_get_session_cache_memsize

```
int wolfSSL_get_session_cache_memsize(
    void
)
```

この関数は、セッションキャッシュの保存バッファがどのくらい大きくあるべきかを返します。

Parameters:

- **none** パラメータなし。

See: [wolfSSL_memrestore_session_cache](#)

Return: int この関数は、セッションキャッシュの保存バッファのサイズを表す整数を返します。

Example

```
int sz = // エラーチェックのための最小サイズ;
...
if(sz < wolfSSL_get_session_cache_memsize()){
    // メモリバッファが小さすぎます。
}
```

A.5.2.27 function wolfSSL_session_reused

```
int wolfSSL_session_reused(
    WOLFSSL * ssl
)
```

この関数は options 構造体の resuming メンバを返します。このフラグはセッションを再利用するかどうかを示します。再利用しない場合、新しいセッションを確立する必要があります。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_SESSION_free](#)
- [wolfSSL_GetSessionIndex](#)
- [wolfSSL_memsave_session_cache](#)

Return: この関数はセッション再利用のフラグを表す Options 構造体に保持された int 型を返します。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(!wolfSSL_session_reused(sslResume)){
    // セッション再利用は許可されていません。
}
```

A.5.2.28 function wolfSSL_get_version

```
const char * wolfSSL_get_version(
    WOLFSSL * ssl
)
```

使用されている SSL バージョンを文字列として返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) を使用して作成された WOLFSSL 構造体へのポインタ。

See: [wolfSSL_lib_version](#)

Return:

- "SSLv3" SSLv3 を使用しています
- "TLSv1" TLSv1 を使用しています
- "TLSv1.1" TLSv1.1 を使用しています
- "TLSv1.2" TLSv1.2 を使用しています
- "TLSv1.3" TLSv1.3 を使用しています
- "DTLS" DTLS を使用しています
- "DTLSv1.2" DTLSv1.2 を使用しています
- "unknown" 使用されている TLS のバージョンを判定する際に問題が発生しました。

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // 何らかの wolfSSL メソッド
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);
printf(wolfSSL_get_version("Using version: %s", ssl));
```

A.5.2.29 function wolfSSL_get_current_cipher_suite

```
int wolfSSL_get_current_cipher_suite(
    WOLFSSL * ssl
)
```

ssl セッションが使用している現在の暗号スイートを返します。

Parameters:

- **ssl** チェックする SSL セッション。

See:

- [wolfSSL_CIPHER_get_name](#)
- [wolfSSL_get_current_cipher](#)
- [wolfSSL_get_cipher_list](#)

Return:

- `ssl->options.cipherSuite` 現在の暗号スイートを表す整数。

- 0 提供された ssl セッションが null です。

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // 何らかの wolfSSL メソッド
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);
```

```
if(wolfSSL_get_current_cipher_suite(ssl) == 0)
{
    // 暗号スイートの取得エラー
}
```

A.5.2.30 function wolfSSL_get_current_cipher

```
WOLFSSL_CIPHER * wolfSSL_get_current_cipher(
    WOLFSSL * ssl
)
```

この関数は ssl セッション内の現在の暗号へのポインタを返します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_get_cipher`
- `wolfSSL_get_cipher_name_internal`
- `wolfSSL_get_cipher_name`

Return:

- この関数は WOLFSSL 構造体の cipher メンバのアドレスを返します。これは WOLFSSL_CIPHER 構造体へのポインタです。
- NULL WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
WOLFSSL_CIPHER* cipherCurr = wolfSSL_get_current_cipher;

if(!cipherCurr){
    // 失敗ケース。
} else {
    // 暗号が cipherCurr に返されました
}
```

A.5.2.31 function wolfSSL_CIPHER_get_name

```
const char * wolfSSL_CIPHER_get_name(
    const WOLFSSL_CIPHER * cipher
)
```

この関数は SSL オブジェクト内の暗号スイートを利用可能なスイートと照合し、文字列表現を返します。

Parameters:

- **cipher** WOLFSSL_CIPHER 構造体への定数ポインタ。

See:

- `wolfSSL_get_cipher`
- `wolfSSL_get_current_cipher`
- `wolfSSL_get_cipher_name_internal`
- `wolfSSL_get_cipher_name`

Return:

- `string` この関数は一致した暗号スイートの文字列表現を返します。
- `none` 一致するスイートがない場合は "None" を返します。

Example

```
// DHE_RSA ... の形式で暗号名を取得します
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){
    // 返された暗号の健全性チェック
}
```

A.5.2.32 function wolfSSL_get_cipher

```
const char * wolfSSL_get_cipher(
    WOLFSSL *
)
```

この関数は SSL オブジェクト内の暗号スイートを利用可能なスイートと照合します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_CIPHER_get_name`
- `wolfSSL_get_current_cipher`

Return: この関数は一致したスイートの文字列値を返します。一致するスイートがない場合は "None" を返します。

Example

```
#ifdef WOLFSSL_DTLS
...
// 有効なスイートが使用されていることを確認
if(wolfSSL_get_cipher(ssl) == NULL){
    WOLFSSL_MSG(" インポートされた暗号スイートと一致しません");
    return MATCH_SUITE_ERROR;
}
...
#endif // WOLFSSL_DTLS
```

A.5.2.33 function wolfSSL_BIO_get_mem_data

```
int wolfSSL_BIO_get_mem_data(
    WOLFSSL_BIO * bio,
    void * p
)
```

これは、内部メモリバッファの先頭にバイトポインタを設定するために使用されます。

Parameters:

- **bio** メモリバッファを取得する WOLFSSL_BIO 構造体。
- **p** メモリバッファに設定するバイトポインタ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- [wolfSSL_BIO_set_fp](#)
- wolfSSL_BIO_free

Return:

- size 成功時、バッファのサイズが返されます。
- SSL_FATAL_ERROR エラーケースが発生した場合。

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);
// ret 値を確認
```

A.5.2.34 function wolfSSL_BIO_set_fd

```
long wolfSSL_BIO_set_fd(
    WOLFSSL_BIO * b,
    int fd,
    int flag
)
```

bio が使用するファイルディスクリプタを設定します。

Parameters:

- **bio** fd を設定する WOLFSSL_BIO 構造体。
- **fd** 使用するファイルディスクリプタ。
- **closeF** fd をクローズする際の動作フラグ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) 成功時。

Example

```
WOLFSSL_BIO* bio;
int fd;
// bio をセットアップ
wolfSSL_BIO_set_fd(bio, fd, BIO_NOCLOSE);
```


A.5.2.35 function wolfSSL_BIO_set_close

```
int wolfSSL_BIO_set_close(  
    WOLFSSL_BIO * b,  
    long flag  
)
```

クローズフラグを設定します。これは、BIO が解放される際に I/O ストリームをクローズすべきかを示すために使用されます。

Parameters:

- **bio** WOLFSSL_BIO 構造体。
- **flag** I/O ストリームをクローズする際の動作フラグ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) 成功時。

Example

```
WOLFSSL_BIO* bio;  
// bio をセットアップ  
wolfSSL_BIO_set_close(bio, BIO_NOCLOSE);
```

A.5.2.36 function wolfSSL_BIO_s_socket

```
WOLFSSL_BIO_METHOD * wolfSSL_BIO_s_socket(  
    void  
)
```

これは、BIO_SOCKET タイプの WOLFSSL_BIO_METHOD を取得するために使用されます。

Parameters:

- **none** パラメータなし。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return: WOLFSSL_BIO_METHOD ソケットタイプである WOLFSSL_BIO_METHOD 構造体へのポインタ。

Example

```
WOLFSSL_BIO* bio;  
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_socket);
```

A.5.2.37 function wolfSSL_BIO_set_write_buf_size

```
int wolfSSL_BIO_set_write_buf_size(  
    WOLFSSL_BIO * b,  
    long size  
)
```

これは、WOLFSSL_BIO の書き込みバッファのサイズを設定するために使用されます。書き込みバッファが以前に設定されていた場合、この関数はサイズをリセットする際にそれを解放します。これは、読み取りと書き込みのインデックスを 0 にリセットする点で wolfSSL_BIO_reset に似ています。

Parameters:

- **bio** fd を設定する WOLFSSL_BIO 構造体。
- **size** 割り当てるバッファのサイズ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS 書き込みバッファの設定に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。

Example

```
WOLFSSL_BIO* bio;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret = wolfSSL_BIO_set_write_buf_size(bio, 15000);
// 戻り値を確認
```

A.5.2.38 function wolfSSL_BIO_make_bio_pair

```
int wolfSSL_BIO_make_bio_pair(
    WOLFSSL_BIO * b1,
    WOLFSSL_BIO * b2
)
```

これは、2 つの bio をペアにするために使用されます。ペアになった bio は双方向パイプのように動作し、一方への書き込みはもう一方から読み取ることができ、その逆も同様です。両方の bio が同じスレッドにあることが期待されます。この関数はスレッドセーフではありません。2 つの bio のうちの 1 つを解放すると、両方のペアが解除されます。いずれかの bio に書き込みバッファサイズが以前に設定されていなかった場合、ペアになる前にデフォルトサイズの 17000(WOLFSSL_BIO_SIZE) に設定されます。

Parameters:

- **b1** ペアを設定する WOLFSSL_BIO 構造体。
- **b2** ペアを完成させる 2 番目の WOLFSSL_BIO 構造体。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS 2 つの bio のペア化に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。

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);
// ret 値を確認
```

A.5.2.39 function wolfSSL_BIO_ctrl_reset_read_request

```
int wolfSSL_BIO_ctrl_reset_read_request(  
    WOLFSSL_BIO * b  
)
```

これは、読み取り要求フラグを 0 に戻すために使用されます。

Parameters:

- **b** 読み取り要求フラグを設定する WOLFSSL_BIO 構造体。

See:

- wolfSSL_BIO_new, wolfSSL_BIO_s_mem
- wolfSSL_BIO_new, wolfSSL_BIO_free

Return:

- SSL_SUCCESS 値の設定に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。

Example

```
WOLFSSL_BIO* bio;  
int ret;  
...  
ret = wolfSSL_BIO_ctrl_reset_read_request(bio);  
// ret 値を確認
```

A.5.2.40 function wolfSSL_BIO_nread0

```
int wolfSSL_BIO_nread0(  
    WOLFSSL_BIO * bio,  
    char ** buf  
)
```

これは、読み取り用のバッファポインタを取得するために使用されます。wolfSSL_BIO_nread とは異なり、内部読み取りインデックスは関数呼び出しから返される数だけ進められません。返される値を超えて読み取ると、配列の境界外を読み取る結果になる可能性があります。

Parameters:

- **bio** 読み取る WOLFSSL_BIO 構造体。
- **buf** 読み取り配列の先頭に設定するポインタ。

See: wolfSSL_BIO_new

Return: > =0 成功時、読み取るバイト数を返します。

Example

```
WOLFSSL_BIO* bio;  
char* bufPt;  
int ret;  
// bio をセットアップ  
ret = wolfSSL_BIO_nread0(bio, &bufPt); // 可能な限り多くのバイトを読み取り  
// 負の ret 値を確認  
// bufPt から ret バイトを読み取り
```

A.5.2.41 function wolfSSL_BIO_nread

```
int wolfSSL_BIO_nread(  
    WOLFSSL_BIO * bio,  
    char ** buf,  
    int num  
)
```

これは、読み取り用のバッファポインタを取得するために使用されます。内部読み取りインデックスは、関数呼び出しから返される数だけ進められ、buf は読み取るバッファの先頭を指します。読み取りバッファ内のバイト数が num で要求された値より少ない場合、より小さい値が返されます。返される値を超えて読み取ると、配列の境界外を読み取る結果になる可能性があります。

Parameters:

- **bio** 読み取る WOLFSSL_BIO 構造体。
- **buf** 読み取り配列の先頭に設定するポインタ。
- **num** 読み取りを試みるバイト数。

See:

- wolfSSL_BIO_new
- [wolfSSL_BIO_nwrite](#)

Return:

- ≥ 0 成功時、読み取るバイト数を返します。
- WOLFSSL_BIO_ERROR(-1) 読み取るものがない場合のエラーケースで-1 を返します。

Example

```
WOLFSSL_BIO* bio;  
char* bufPt;  
int ret;  
  
// bio をセットアップ  
ret = wolfSSL_BIO_nread(bio, &bufPt, 10); // 10 バイトの読み取りを試行  
// 負の ret 値を確認  
// bufPt から ret バイトを読み取り
```

A.5.2.42 function wolfSSL_BIO_nwrite

```
int wolfSSL_BIO_nwrite(  
    WOLFSSL_BIO * bio,  
    char ** buf,  
    int num  
)
```

関数が返す数だけのバイトを書き込むためのバッファへのポインタを取得します。返される値よりも多くのバイトを返されたポインタに書き込むと、境界外への書き込みになる可能性があります。

Parameters:

- **bio** 書き込む WOLFSSL_BIO 構造体。
- **buf** 書き込むバッファへのポインタ。
- **num** 書き込みたいバイト数。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free
- [wolfSSL_BIO_nread](#)

Return:

- int バッファポインタに書き込むことができるバイト数を返します。
- WOLFSSL_BIO_UNSET(-2) bio ペアの一部ではない場合。
- WOLFSSL_BIO_ERROR(-1) 書き込むスペースがこれ以上ない場合。

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;
// bio をセットアップ
ret = wolfSSL_BIO_nwrite(bio, &bufPt, 10); // 10 バイトの書き込みを試行
// 負の ret 値を確認
// bufPt に ret バイトを書き込み
```

A.5.2.43 function wolfSSL_BIO_reset

```
int wolfSSL_BIO_reset(
    WOLFSSL_BIO * bio
)
```

bio を初期状態にリセットします。例えば、BIO_BIO タイプの場合、これは読み取りと書き込みのインデックスをリセットします。

Parameters:

- **bio** リセットする WOLFSSL_BIO 構造体。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return:

- 0 bio のリセットに成功した場合。
- WOLFSSL_BIO_ERROR(-1) 不正な入力またはリセットに失敗した場合に返されます。

Example

```
WOLFSSL_BIO* bio;
// bio をセットアップ
wolfSSL_BIO_reset(bio);
//pt を使用
```

A.5.2.44 function wolfSSL_BIO_seek

```
int wolfSSL_BIO_seek(
    WOLFSSL_BIO * bio,
    int ofs
)
```

この関数は、ファイルポインタを指定されたオフセットに調整します。これはファイルの先頭からのオフセットです。

Parameters:

- **bio** 設定する WOLFSSL_BIO 構造体。
- **ofs** ファイルへのオフセット。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- **wolfSSL_BIO_set_fp**
- wolfSSL_BIO_free

Return:

- 0 シークに成功した場合。
- -1 エラーケースが発生した場合。

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret = wolfSSL_BIO_set_fp(bio, &fp);
// ret 値を確認
ret = wolfSSL_BIO_seek(bio, 3);
// ret 値を確認
```

A.5.2.45 function wolfSSL_BIO_write_filename

```
int wolfSSL_BIO_write_filename(
    WOLFSSL_BIO * bio,
    char * name
)
```

ファイルを設定し、書き込むために使用されます。ファイル内の既存データを上書きし、bio が解放される際にファイルを閉じるよう設定されます。

Parameters:

- **bio** ファイルを設定する WOLFSSL_BIO 構造体。
- **name** 書き込み先のファイル名。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_file
- **wolfSSL_BIO_set_fp**
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS ファイルのオープンと設定が成功した場合。
- SSL_FAILURE エラーが発生した場合。

Example

```
WOLFSSL_BIO* bio;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret = wolfSSL_BIO_write_filename(bio, "test.txt");
// ret 値を確認
```

A.5.2.46 function wolfSSL_BIO_set_mem_eof_return

```
long wolfSSL_BIO_set_mem_eof_return(
    WOLFSSL_BIO * bio,
    int v
)
```

ファイル終端値を設定するために使用されます。一般的な値は-1 で、期待される正の値と混同しないようにします。

Parameters:

- **bio** ファイル終端値を設定する WOLFSSL_BIO 構造体。
- **v** bio に設定する値。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- **wolfSSL_BIO_set_fp**
- wolfSSL_BIO_free

Return: 0 完了時に返されます。

Example

```
WOLFSSL_BIO* bio;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret = wolfSSL_BIO_set_mem_eof_return(bio, -1);
// ret 値を確認
```

A.5.2.47 function wolfSSL_BIO_get_mem_ptr

```
long wolfSSL_BIO_get_mem_ptr(
    WOLFSSL_BIO * bio,
    WOLFSSL_BUF_MEM ** m
)
```

WOLFSSL_BIO メモリポインタの getter 関数です。

Parameters:

- **bio** メモリポインタを取得するための WOLFSSL_BIO 構造体へのポインタ。
- **ptr** 現在 char* である構造体。bio のメモリを指すように設定されます。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return:

- SSL_SUCCESS ポインタの取得に成功した場合、SSL_SUCCESS が返されます (現在の値は 1)。
- SSL_FAILURE NULL 引数が渡された場合に返されます (現在の値は 0)。

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_BUF_MEM* pt;
// bio をセットアップ
wolfSSL_BIO_get_mem_ptr(bio, &pt);
// pt を使用
```

A.5.2.48 function wolfSSL_lib_version

```
const char * wolfSSL_lib_version(
    void
)
```

この関数は現在のライブラリバージョンを返します。

Parameters:

- **none** パラメータなし。

See: word32_wolfSSL_lib_version_hex

Return: LIBWOLFSSL_VERSION_STRING バージョンを定義する const char ポインタ。

Example

```
char version[MAXSIZE];
version = wolfSSL_KeepArrays();
...
if(version != ExpectedVersion){
    // 不一致ケースを処理
}
```

A.5.2.49 function wolfSSL_lib_version_hex

```
word32 wolfSSL_lib_version_hex(
    void
)
```

この関数は現在のライブラリバージョンを 16 進表記で返します。

Parameters:

- **none** パラメータなし。

See: wolfSSL_lib_version

Return: LIBWOLFSSL_VERSION_HEX wolfssl/version.h で定義された 16 進バージョンを返します。

Example

```
word32 libV;
libV = wolfSSL_lib_version_hex();

if(libV != EXPECTED_HEX){
    // 予期しない値の処理方法
} else {
    // libV の期待される結果
}
```

A.5.2.50 function wolfSSL_negotiate

```
int wolfSSL_negotiate(
    WOLFSSL * ssl
)
```

SSL メソッドの側面に基づいて実際の接続または受け入れを実行します。クライアント側から呼び出された場合は wolfSSL_connect() が実行され、サーバー側から呼び出された場合は wolfSSL_accept() が実行されます。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。

See:

- SSL_connect

- `SSL_accept`

Return:

- `SSL_SUCCESS` 成功した場合に返されます (注: 古いバージョンでは 0 が返されます)。
- `SSL_FATAL_ERROR` 基礎となる呼び出しがエラーになった場合に返されます。特定のエラーコードを取得するには `wolfSSL_get_error()` を使用してください。

Example

```
int ret = SSL_FATAL_ERROR;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_negotiate(ssl);
if (ret == SSL_FATAL_ERROR) {
    // SSL 確立に失敗しました
    int error_code = wolfSSL_get_error(ssl);
    ...
}
...
```

A.5.2.51 function wolfSSL_connect_cert

```
int wolfSSL_connect_cert(
    WOLFSSL * ssl
)
```

この関数はクライアント側で呼び出され、ピアの証明書チェーンを取得するのに十分な長さだけサーバーとの SSL/TLS ハンドシェイクを開始します。この関数が呼び出されるとき、基礎となる通信チャネルはすでに設定されています。`wolfSSL_connect_cert()` は、ブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、`wolfSSL_connect_cert()` は、基礎となる I/O が `wolfSSL_connect_cert()` がハンドシェイクを続行するために必要な処理を満たせないときに戻ります。この場合、`wolfSSL_get_error()` の呼び出しは `SSL_ERROR_WANT_READ` または `SSL_ERROR_WANT_WRITE` のいずれかを生成します。呼び出しプロセスは、基礎となる I/O が準備できたときに `wolfSSL_connect_cert()` の呼び出しを繰り返す必要があり、`wolfSSL` は中断した場所から再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、`select()` を使用して必要な条件をチェックできます。基礎となる I/O がブロッキング I/O の場合、`wolfSSL_connect_cert()` はピアの証明書チェーンが受信されたときにのみ戻ります。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された `WOLFSSL` 構造体へのポインタ。

See:

- `wolfSSL_get_error`
- `wolfSSL_connect`
- `wolfSSL_accept`

Return:

- `SSL_SUCCESS` 成功時。
- `SSL_FAILURE` SSL セッションパラメータが `NULL` の場合に返されます。
- `SSL_FATAL_ERROR` エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、`wolfSSL_get_error()` を呼び出してください。

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));
}

```

A.5.2.52 function wolfSSL_writev

```

int wolfSSL_writev(
    WOLFSSL * ssl,
    const struct iovec * iov,
    int iovcnt
)

```

writev のセマンティクスをシミュレートしますが、SSL_write() の動作とフロント追加が小さい場合があるため、実際には一度にブロック単位では行いません。writev を使用するソフトウェアへの移植を容易にします。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **iov** 書き込む I/O ベクトルの配列。
- **iovcnt** iov 配列内のベクトル数。

See: wolfSSL_write

Return:

- > 0 成功時、書き込まれたバイト数。
- 0 失敗時に返されます。wolfSSL_get_error() を呼び出して特定のエラーコードを取得してください。
- MEMORY_ERROR メモリエラーが発生した場合に返されます。
- SSL_FATAL_ERROR エラーが発生した場合、またはノンブロッキングソケットを使用しているときに SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーを受信し、アプリケーションが wolfSSL_write() を再度呼び出す必要がある場合に返されます。wolfSSL_get_error() を使用して特定のエラーコードを取得してください。

Example

```

WOLFSSL* ssl = 0;
char *bufA = "hello\n";
char *bufB = "hello world\n";
int iovcnt;
struct iovec iov[2];

iov[0].iov_base = bufA;
iov[0].iov_len = strlen(bufA);
iov[1].iov_base = bufB;
iov[1].iov_len = strlen(bufB);
iovcnt = 2;
...
ret = wolfSSL_writev(ssl, iov, iovcnt);
// "ret" バイトを書き込みました。または <=0 の場合はエラー

```

A.5.2.53 function wolfSSL_SNI_Status

```

unsigned char wolfSSL_SNI_Status(
    WOLFSSL * ssl,

```

```
    unsigned char type
)
```

この関数は SNI オブジェクトのステータスを取得します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **type** SNI タイプ。

See:

- TLSX_SNI_Status
- TLSX_SNI_find
- TLSX_Find

Return:

- value SNI が NULL でない場合、この関数は SNI 構造体の status メンバーのバイト値を返します。
- 0 SNI オブジェクトが NULL の場合。

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));
...
```

A.5.2.54 function wolfSSL_UseSecureRenegotiation

```
int wolfSSL_UseSecureRenegotiation(
    WOLFSSL * ssl
)
```

この関数は、提供された WOLFSSL 構造体に対して安全な再ネゴシエーションを強制します。これは推奨されません。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- TLSX_Find
- TLSX_UseSecureRenegotiation

Return:

- SSL_SUCCESS 安全な再ネゴシエーションの設定に成功しました。
- BAD_FUNC_ARG ssl が NULL の場合にエラーを返します。
- MEMORY_E 安全な再ネゴシエーション用のメモリを割り当てできない場合にエラーを返します。

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // 何らかの wolfSSL メソッド
```

```
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSecureRenegotiation(ssl) != SSL_SUCCESS)
{
    // 安全な再ネゴシエーションの設定エラー
}
```

A.5.2.55 function wolfSSL_Rehandshake

```
int wolfSSL_Rehandshake(
    WOLFSSL * ssl
)
```

この関数は安全な再ネゴシエーションハンドシェイクを実行します。wolfSSL はこの機能を推奨していないため、これはユーザが強制するものです。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_negotiate`
- `wc_InitSha512`
- `wc_InitSha384`
- `wc_InitSha256`
- `wc_InitSha`
- `wc_InitMd5`

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合、またはサブルーチンで受け入れられない引数が渡された場合に返されます。
- SECURE_RENEGOTIATION_E ハンドシェイクの再ネゴシエーションでエラーがあった場合に返されます。
- SSL_FATAL_ERROR サーバまたはクライアント設定にエラーがあり、再ネゴシエーションを完了できなかった場合に返されます。wolfSSL_negotiate() を参照してください。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_Rehandshake(ssl) != SSL_SUCCESS){
    // エラーが発生し、再ハンドシェイクは成功しませんでした。
}
```

A.5.2.56 function wolfSSL_UseSessionTicket

```
int wolfSSL_UseSessionTicket(
    WOLFSSL * ssl
)
```

提供された WOLFSSL 構造体にセッションチケットを使用するよう強制します。定数 HAVE_SESSION_TICKET が定義されており、定数 NO_WOLFSSL_CLIENT が定義されていない必要があります。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS セッションチケットの使用設定に成功しました。
- BAD_FUNC_ARG ssl が NULL の場合に返されます。
- MEMORY_E セッションチケット設定のためのメモリ割り当てエラー。

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // 何らかの wolfSSL メソッド
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSessionTicket(ssl) != SSL_SUCCESS)
{
    // セッションチケットの設定エラー
}
```

A.5.2.57 function wolfSSL_get_SessionTicket

```
int wolfSSL_get_SessionTicket(
    WOLFSSL * ssl,
    unsigned char * buf,
    word32 * bufSz
)
```

この関数は、Session 構造体の ticket メンバをバッファにコピーします。buf が NULL で bufSz が非 NULL の場合、bufSz はチケット長に設定されます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** メモリバッファを表す byte ポインタ。
- **bufSz** バッファサイズを表す word32 ポインタ。

See:

- [wolfSSL_UseSessionTicket](#)
- [wolfSSL_set_SessionTicket](#)

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- BAD_FUNC_ARG ssl または bufSz が NULL の場合、または bufSz が非 NULL で buf が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buf;
word32 bufSz; // buf サイズで初期化
...
if(wolfSSL_get_SessionTicket(ssl, buf, bufSz) <= 0){
    // バッファに何も書き込まれませんでした。
} else {
```

```
// バッファは ssl->session->ticket の内容を保持しています。
}
```

A.5.2.58 function wolfSSL_set_SessionTicket

```
int wolfSSL_set_SessionTicket(
    WOLFSSL * ssl,
    const unsigned char * buf,
    word32 bufSz
)
```

この関数は、WOLFSSL 構造体内の WOLFSSL_SESSION 構造体の ticket メンバを設定します。関数に渡されたバッファはメモリにコピーされます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** セッション構造体の ticket メンバに読み込まれる byte ポインタ。
- **bufSz** バッファのサイズを表す word32 型。

See: [wolfSSL_set_SessionTicket_cb](#)

Return:

- SSL_SUCCESS 関数の実行が成功した場合に返されます。関数はエラーなく返されました。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合に返されます。bufSz 引数がゼロでないのに buf 引数が NULL の場合にも返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buffer; // 読み込むファイル
word32 bufSz;
...
if(wolfSSL_KeepArrays(ssl, buffer, bufSz) != SSL_SUCCESS){
    // バッファをメモリに読み込む際にエラーが発生しました。
}
```

A.5.2.59 function wolfSSL_PrintSessionStats

```
int wolfSSL_PrintSessionStats(
    void
)
```

この関数は、セッションの統計情報を出力します。

Parameters:

- **none** パラメータなし。

See: [wolfSSL_get_session_stats](#)

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで返された場合に返されます。セッション統計が正常に取得され、出力されました。
- BAD_FUNC_ARG サブルーチン [wolfSSL_get_session_stats\(\)](#) に許容できない引数が渡された場合に返されます。
- BAD_MUTEX_E サブルーチンで mutex エラーが発生した場合に返されます。

Example

```
// 統計を取得するためのセッションオブジェクトが必要です
if(wolfSSL_PrintSessionStats(void) != SSL_SUCCESS){
    // セッション統計を出力しませんでした
}
```

A.5.2.60 function wolfSSL_get_session_stats

```
int wolfSSL_get_session_stats(
    unsigned int * active,
    unsigned int * total,
    unsigned int * peak,
    unsigned int * maxSessions
)
```

この関数は、セッションの統計情報を取得します。

Parameters:

- **active** 総現在セッション数を表す word32 ポインタ。
- **total** 総セッション数を表す word32 ポインタ。
- **peak** ピークセッション数を表す word32 ポインタ。
- **maxSessions** 最大セッション数を表す word32 ポインタ。

See: [wolfSSL_PrintSessionStats](#)

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで返された場合に返されます。セッション統計が正常に取得され、出力されました。
- BAD_FUNC_ARG サブルーチン wolfSSL_get_session_stats() に許容できない引数が渡された場合に返されます。
- BAD_MUTEX_E サブルーチンで mutex エラーが発生した場合に返されます。

Example

```
int wolfSSL_PrintSessionStats(void){
    ...
    ret = wolfSSL_get_session_stats(&totalSessionsNow,
        &totalSessionsSeen, &peak, &maxSessions);
    ...
    return ret;
}
```

A.5.2.61 function wolfSSL_BIO_set_fp

```
long wolfSSL_BIO_set_fp(
    WOLFSSL_BIO * bio,
    XFILE fp,
    int c
)
```

これは、BIO の内部ファイルポインタを設定するために使用されます。

Parameters:

- **bio** ペアを設定する WOLFSSL_BIO 構造体。
- **fp** bio に設定するファイルポインタ。
- **c** ファイルクローズ動作フラグ。

See:

- wolfSSL_BIO_new

- wolfSSL_BIO_s_mem
- **wolfSSL_BIO_get_fp**
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS ファイルポインタの設定に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。

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);
// ret 値を確認
```

A.5.2.62 function wolfSSL_BIO_get_fp

```
long wolfSSL_BIO_get_fp(
    WOLFSSL_BIO * bio,
    XFILE * fp
)
```

これは、BIO の内部ファイルポインタを取得するために使用されます。

Parameters:

- **bio** ペアを設定する WOLFSSL_BIO 構造体。
- **fp** bio に設定するファイルポインタ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- **wolfSSL_BIO_set_fp**
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS ファイルポインタの取得に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret = wolfSSL_BIO_get_fp(bio, &fp);
// ret 値を確認
```

A.5.2.63 function wolfSSL_BIO_ctrl_pending

```
size_t wolfSSL_BIO_ctrl_pending(
    WOLFSSL_BIO * b
)
```

読み取り保留中のバイト数を取得します。BIO タイプが BIO_BIO の場合、ペアから読み取るバイト数です。BIO が SSL オブジェクトを含む場合、SSL オブジェクトからの保留中のデータです (wolfSSL_pending(ssl))。BIO_MEMORY タイプの場合、メモリバッファのサイズを返します。

Parameters:

- **bio** すでに作成されている WOLFSSL_BIO 構造体へのポインタ。

See:

- [wolfSSL_BIO_make_bio_pair](#)
- [wolfSSL_BIO_new](#)

Return: > =0 保留中のバイト数。

Example

```
WOLFSSL_BIO* bio;    int pending;
bio = wolfSSL_BIO_new();
...
pending = wolfSSL_BIO_ctrl_pending(bio);
```

A.5.2.64 function wolfSSL_set_jobject

```
int wolfSSL_set_jobject(
    WOLFSSL * ssl,
    void * objPtr
)
```

この関数は、WOLFSSL 構造体の jobjectRef メンバを設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **objPtr** jobjectRef に設定される void ポインタ。

See: [wolfSSL_get_jobject](#)

Return:

- SSL_SUCCESS jobjectRef が objPtr に適切に設定された場合に返されます。
- SSL_FAILURE 関数が適切に実行されず、jobjectRef が設定されていない場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new();
void* objPtr = &obj;
...
if(wolfSSL_set_jobject(ssl, objPtr)){
    // 成功ケース
}
```

A.5.2.65 function wolfSSL_get_jobject

```
void * wolfSSL_get_jobject(
    WOLFSSL * ssl
)
```

この関数は、WOLFSSL 構造体の jobjectRef メンバを返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See: [wolfSSL_set_jobject](#)

Return:

- value WOLFSSL 構造体が NULL でない場合、関数は jobjectRef 値を返します。
- NULL WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL(ctx);
...
void* jobject = wolfSSL_get_jobject(ssl);

if(jobject != NULL){
    // 成功ケース
}
```

A.5.2.66 function wolfSSL_connect

```
int wolfSSL_connect(
    WOLFSSL * ssl
)
```

この関数はクライアント側で呼び出され、サーバとの SSL/TLS ハンドシェイクを開始します。この関数が呼び出されるとき、基礎となる通信チャンネルはすでに設定されています。wolfSSL_connect() は、ブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、wolfSSL_connect() は、基礎となる I/O が wolfSSL_connect() がハンドシェイクを続行するために必要とするものを満たすことができないときに返されます。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE を返します。呼び出しプロセスは、基礎となる I/O の準備ができたときに wolfSSL_connect() の呼び出しを繰り返す必要があり、wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキングの場合、wolfSSL_connect() はハンドシェイクが完了するかエラーが発生するまで返されません。wolfSSL は、OpenSSL とは異なるアプローチで証明書検証を行います。クライアントのデフォルトポリシーはサーバを検証することです。つまり、サーバを検証するための CA を読み込まない場合、接続エラー、検証不可 (-155) が発生します。サーバの検証が失敗しても SSL_connect が成功し、セキュリティを低下させる OpenSSL の動作を模倣したい場合は、SSL_new() を呼び出す前に SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); を呼び出すことでこれを実現できます。ただし、推奨されません。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功した場合。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出してください。

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));
}

```

A.5.2.67 function wolfSSL_update_keys

```

int wolfSSL_update_keys(
    WOLFSSL * ssl
)

```

この関数は、鍵のロールオーバーを強制するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。KeyUpdate メッセージがピアに送信され、暗号化用の新しい鍵が計算されます。ピアは KeyUpdate メッセージを送り返し、その後新しい復号鍵が計算されます。この関数は、ハンドシェイクが完了した後にのみ呼び出すことができます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) を使用して作成された WOLFSSL 構造体へのポインタ。

See: [wolfSSL_write](#)

Return:

- BAD_FUNC_ARG ssl が NULL または TLS v1.3 を使用していない場合。
- WANT_WRITE 書き込みの準備ができていない場合。
- WOLFSSL_SUCCESS 成功した場合。

Example

```

int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_update_keys(ssl);
if (ret == WANT_WRITE) {
    // I/O の準備ができたなら再度呼び出す必要があります。
}
else if (ret != WOLFSSL_SUCCESS) {
    // 鍵更新の送信に失敗しました。
}

```

A.5.2.68 function wolfSSL_key_update_response

```

int wolfSSL_key_update_response(
    WOLFSSL * ssl,
    int * required
)

```

この関数は、鍵のロールオーバーが進行中かどうかを判断するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。wolfSSL_update_keys() が呼び出されると、KeyUpdate メッセージが送信され、暗号化鍵が更新されます。復号鍵は、応答を受信したときに更新されます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) を使用して作成された WOLFSSL 構造体へのポインタ。
- **required** 鍵更新応答が不要な場合は 0。鍵更新応答が必要な場合は 1。

See: [wolfSSL_update_keys](#)

Return:

- 0 成功した場合。

- BAD_FUNC_ARG ssl が NULL または TLS v1.3 を使用していない場合。

Example

```
int ret;
WOLFSSL* ssl;
int required;
...
ret = wolfSSL_key_update_response(ssl, &required);
if (ret != 0) {
    // 不正なパラメータ
}
if (required) {
    // 暗号化鍵が更新され、復号鍵を変更するための応答を待っています。
}
```

A.5.2.69 function wolfSSL_request_certificate

```
int wolfSSL_request_certificate(
    WOLFSSL * ssl
)
```

この関数は、TLS v1.3 クライアントからクライアント証明書を要求します。これは、Web サーバがクライアント認証を必要とするページと必要としないページの両方を提供している場合に便利です。接続上で最大 256 回の要求を送信できます。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_allow_post_handshake_auth`
- `wolfSSL_write`

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- WANT_WRITE 書き込みの準備ができていない場合。
- SIDE_ERROR クライアントで呼び出された場合。
- NOT_READY_ERROR ハンドシェイクが完了していないときに呼び出された場合。
- POST_HAND_AUTH_ERROR ポストハンドシェイク認証が許可されていない場合。
- MEMORY_E 動的メモリ割り当てが失敗した場合。
- WOLFSSL_SUCCESS 成功時。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_request_certificate(ssl);
if (ret == WANT_WRITE) {
    // I/O の準備ができれば再度呼び出す必要があります
}
else if (ret != WOLFSSL_SUCCESS) {
    // クライアント証明書の要求に失敗しました
}
```

A.5.2.70 function wolfSSL_connect_TLSv13

```
int wolfSSL_connect_TLSv13(
    WOLFSSL * ssl
)
```

この関数はクライアント側で呼び出され、サーバとの TLS v1.3 ハンドシェイクを開始します。この関数が呼び出されるとき、基礎となる通信チャネルはすでに設定されています。wolfSSL_connect() はブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、基礎となる I/O が wolfSSL_connect() がハンドシェイクを続行するために必要なものを満たせない場合、wolfSSL_connect() は返されます。この場合、wolfSSL_get_error() を呼び出すと SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、基礎となる I/O の準備ができたときに wolfSSL_connect() の呼び出しを繰り返す必要があり、wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキング I/O の場合、wolfSSL_connect() はハンドシェイクが完了するかエラーが発生するまで返されません。wolfSSL は証明書検証に OpenSSL とは異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバを検証することです。つまり、サーバを検証するための CA をロードしない場合、接続エラー「検証できません (-155)」が発生します。サーバの検証が失敗しても SSL_connect が成功するという OpenSSL の動作を模倣し、セキュリティを低下させたい場合は、SSL_new() を呼び出す前に SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0) を呼び出すことでこれを行うことができます。ただし、これは推奨されません。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功時。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出してください。

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));
}
```

A.5.2.71 function wolfSSL_accept_TLSv13

```
wolfSSL_accept_TLSv13(
    WOLFSSL * ssl
)
```

この関数はサーバ側で呼び出され、SSL/TLS クライアントが SSL/TLS ハンドシェイクを開始するのを待ちます。この関数が呼び出されるとき、基礎となる通信チャネルはすでに設定されています。wolfSSL_accept() はブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、基礎となる I/O が wolfSSL_accept() がハンドシェイクを続行するために必要なものを満たせない場合、wolfSSL_accept() は返されます。この場合、wolfSSL_get_error() を呼び出すと SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、データが読み取り可能になったときに wolfSSL_accept の呼び出しを繰り返す必要があります。wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキング I/O の場合、wolfSSL_accept() はハンドシェイクが完了するかエラーが発生するまで返されません。TLS v1.3 接続を期待する場合にこの関数を呼び出してください。ただし、古いバージョンの ClientHello メッセージもサポートされています。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_connect_TLSv13
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功時。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出してください。

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));
}
```

A.5.2.72 function wolfSSL_write_early_data

```
int wolfSSL_write_early_data(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int * outSz
)
```

この関数は再開時にサーバーにアーリーデータを書き込みます。wolfSSL_connect() または wolfSSL_connect_TLSv13() の前にこの関数を呼び出してください。この関数はクライアントでのみ使用されます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

- **data** サーバーに書き込むアーリーデータを保持するバッファ。
- **sz** 書き込むアーリーデータの量 (バイト単位)。
- **outSz** 書き込まれたアーリーデータの量 (バイト単位)。

See:

- [wolfSSL_read_early_data](#)
- [wolfSSL_connect](#)
- [wolfSSL_connect_TLSv13](#)

Return:

- BAD_FUNC_ARG ポインタパラメータが NULL、sz が 0 未満、または TLSv1.3 を使用していない場合。
- SIDE_ERROR サーバーで呼び出された場合。
- WOLFSSL_FATAL_ERROR 接続が確立されなかった場合。
- 成功した場合、書き込まれたアーリーデータのバイト数。

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)) {
    // すべてのアーリーデータが送信されませんでした
}
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));
}
```

A.5.2.73 function wolfSSL_read_early_data

```
int wolfSSL_read_early_data(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int * outSz
)
```

この関数は再開時にクライアントからのアーリーデータを読み取ります。クライアントを受け入れ、ハンドシェイクでアーリーデータを読み取るには、wolfSSL_accept() または wolfSSL_accept_TLSv13() の代わりにこの関数を呼び出します。wolfSSL_is_init_finished() が true を返すまで関数を呼び出す必要があります。アーリーデータは複数のメッセージでクライアントから送信される場合があります。アーリーデータがない場合、ハンドシェイクは通常通り処理されます。この関数はサーバーでのみ使用されます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **data** クライアントから読み取ったアーリーデータを保持するバッファ。
- **sz** バッファのサイズ (バイト単位)。
- **outSz** 読み取ったアーリーデータのバイト数。

See:

- `wolfSSL_write_early_data`
- `wolfSSL_accept`
- `wolfSSL_accept_TLSv13`

Return:

- `BAD_FUNC_ARG` ポインタパラメータが NULL、`sz` が 0 未満、または TLSv1.3 を使用していない場合。
- `SIDE_ERROR` クライアントで呼び出された場合。
- `WOLFSSL_FATAL_ERROR` 接続の受け入れが失敗した場合。
- 読み取ったアーリーデータのバイト数 (ゼロの場合もあります)。

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) {
        // アーリーデータが利用可能
    }
} while (!wolfSSL_is_init_finished(ssl));
```

A.5.2.74 function `wolfSSL_inject`

```
int wolfSSL_inject(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

この関数は WOLFSSL オブジェクトにデータを注入するために呼び出されます。これは、データを単一の場所から読み取り、複数の接続に分割する必要がある場合に便利です。呼び出し元は `wolfSSL_read()` を呼び出して WOLFSSL オブジェクトから平文データを抽出する必要があります。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **data** ssl オブジェクトに注入するデータ。
- **sz** 注入するデータのバイト数。

See: `wolfSSL_read`

Return:

- BAD_FUNC_ARG いずれかのポインタパラメータが NULL または `sz <= 0` の場合。
- APP_DATA_READY 読み取るべきアプリケーションデータが残っている場合。
- MEMORY_E 割り当てが失敗した場合。
- WOLFSSL_SUCCESS 成功時。

Example

```
byte buf[2000]
sz = recv(fd, buf, sizeof(buf), 0);
if (sz <= 0)
    // エラー
if (wolfSSL_inject(ssl, buf, sz) != WOLFSSL_SUCCESS)
    // エラー
sz = wolfSSL_read(ssl, buf, sizeof(buf));
```

A.5.2.75 function wolfSSL_GetIOReadCtx

```
void * wolfSSL_GetIOReadCtx(
    WOLFSSL * ssl
)
```

この関数は WOLFSSL 構造体の IOCB_ReadCtx メンバーを返します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_GetIOWriteCtx`
- `wolfSSL_SetIOReadFlags`
- `wolfSSL_SetIOWriteCtx`
- `wolfSSL_SetIOReadCtx`
- `wolfSSL_CTX_SetIOSend`

Return:

- pointer この関数は WOLFSSL 構造体の IOCB_ReadCtx メンバーへの void 型ポインタを返します。
- NULL WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
void* ioRead;
...
ioRead = wolfSSL_GetIOReadCtx(ssl);
if(ioRead == NULL){
    // 失敗ケース。ssl オブジェクトが NULL でした。
}
```

A.5.2.76 function wolfSSL_GetIOWriteCtx

```
void * wolfSSL_GetIOWriteCtx(
    WOLFSSL * ssl
)
```

この関数は WOLFSSL 構造体の IOCB_WriteCtx メンバーを返します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_GetIOReadCtx](#)
- [wolfSSL_SetIOWriteCtx](#)
- [wolfSSL_SetIOReadCtx](#)
- [wolfSSL_CTX_SetIOSend](#)

Return:

- pointer この関数は WOLFSSL 構造体の IOCB_WriteCtx メンバーへの void 型ポインタを返します。
- NULL WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL* ssl;
void* ioWrite;
...
ioWrite = wolfSSL_GetIOWriteCtx(ssl);
if(ioWrite == NULL){
    // 関数が NULL を返しました。
}
```

A.5.2.77 function wolfSSL_SetIO_NetX

```
void wolfSSL_SetIO_NetX(
    WOLFSSL * ssl,
    NX_TCP_SOCKET * nxsocket,
    ULONG waitoption
)
```

この関数は、WOLFSSL 構造体内の nxCtx 構造体の nxSocket および nxWait メンバーを設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **nxSocket** nxCTX 構造体の nxSocket メンバーに設定される NX_TCP_SOCKET 型へのポインタ。
- **waitOption** nxCtx 構造体の nxWait メンバーに設定される ULONG 型。

See:

- [set_fd](#)
- [NetX_Send](#)
- [NetX_Receive](#)

Return: none 戻り値なし。

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 {
    // 適切なパラメータを渡す必要があります。
}
```

A.6 wolfSSL コンテキストとセッションのセットアップ

A.6.1 Functions

	Name
WOLFSSL_METHOD *	wolfSSLv23_method (void) この関数は wolfSSLv23_client_method に似た WOLFSSL_METHOD を返しますが、どちら側 (サーバ/クライアント) であるかがまだ決定されていない点が異なります。
WOLFSSL_METHOD *	wolfSSLv3_server_method (void)wolfSSLv3_server_method() 関数は、アプリケーションがサーバであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	wolfSSLv3_client_method (void)wolfSSLv3_client_method() 関数は、アプリケーションがクライアントであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	wolfTLSv1_server_method (void)wolfTLSv1_server_method() 関数は、アプリケーションがサーバであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	wolfTLSv1_client_method (void)wolfTLSv1_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	wolfTLSv1_1_server_method (void)wolfTLSv1_1_server_method() 関数は、アプリケーションがサーバであり、TLS 1.1 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

	Name
WOLFSSL_METHOD *	wolfTLSv1_1_client_method (void) wolfTLSv1_1_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	wolfTLSv1_2_server_method (void) wolfTLSv1_2_server_method() 関数は、アプリケーションがサーバであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	wolfTLSv1_2_client_method (void) wolfTLSv1_2_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	wolfDTLSv1_client_method (void) wolfDTLSv1_client_method() 関数は、アプリケーションがクライアントであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポート付きでコンパイルされている場合 (-enable_dtls、または wolfSSL_DTLS を定義することによって) にのみ利用可能です。
WOLFSSL_METHOD *	wolfDTLSv1_server_method (void) wolfDTLSv1_server_method() 関数は、アプリケーションがサーバであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable_dtls、または wolfSSL_DTLS を定義することによって)。

	Name
WOLFSSL_METHOD *	wolfDTLSv1_3_server_method (void) wolfDTLSv1_3_server_method() 関数は、アプリケーションがサーバーであり、DTLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLSv1.3 サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls13、または wolfSSL_DTLS13 を定義することによって)。
WOLFSSL_METHOD *	wolfDTLSv1_3_client_method (void) wolfDTLSv1_3_client_method() 関数は、アプリケーションがクライアントであり、DTLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLSv1.3 サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls13、または wolfSSL_DTLS13 を定義することによって)。
WOLFSSL_METHOD *	wolfDTLS_server_method (void) wolfDTLS_server_method() 関数は、アプリケーションがサーバーであり、利用可能な最新バージョンの DTLS および許可される最小バージョンまでのすべてのバージョンをサポートすることを示すために使用されます。デフォルトの許可される最小バージョンは、WOLFSSL_MIN_DTLS_DOWNGRADE の定義に基づいており、wolfSSL_SetMinVersion() を使用して実行時に変更できます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls、または wolfSSL_DTLS を定義することによって)。

	Name
WOLFSSL_METHOD *	wolfDTLS_client_method (void) wolfDTLS_client_method() 関数は、アプリケーションがクライアントであり、利用可能な最新バージョンの DTLS および許可される最小バージョンまでのすべてのバージョンをサポートすることを示すために使用されます。デフォルトの許可される最小バージョンは、WOLFSSL_MIN_DTLS_DOWNGRADE の定義に基づいており、wolfSSL_SetMinVersion() を使用して実行時に変更できます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls、または wolfSSL_DTLS を定義することによって)。
int	wolfSSL_use_old_poly (WOLFSSL * ssl, int value) chacha-poly AEAD 構築の最初のリリースと新しいバージョンの間にはいくつかの違いがあるため、古いバージョンを使用するサーバー/クライアントと通信するためのオプションを追加しました。デフォルトでは、wolfSSL は新しいバージョンを使用します。
int	wolfSSL_CTX_trust_peer_cert (WOLFSSL_CTX * ctx, const char * file, int type) この関数は、TLS/SSL ハンドシェイクを実行する際にピアを検証するために使用する証明書を読み込みます。ハンドシェイク中に送信されるピア証明書は、利用可能な場合は SKID を使用し、署名を使用して比較されます。これら 2 つが一致しない場合は、読み込まれた CA が使用されます。この機能はマクロ WOLFSSL_TRUST_PEER_CERT を定義することで有効になります。適切な使用方法については、examples を参照してください。
long	wolfSSL_CTX_get_verify_depth (WOLFSSL_CTX * ctx) この関数は、CTX 構造体を使用して証明書チェーンの深度を取得します。
WOLFSSL_CTX *	wolfSSL_CTX_new (WOLFSSL_METHOD *) この関数は、入力として希望する SSL/TLS プロトコルメソッドを取り、新しい SSL コンテキストを作成します。
WOLFSSL *	wolfSSL_new (WOLFSSL_CTX *) この関数は、既に作成された SSL コンテキストを入力として取り、新しい SSL セッションを作成します。
int	wolfSSL_set_fd (WOLFSSL * ssl, int fd) この関数は、ファイルディスクリプタ (fd) を SSL 接続の入出力機能として割り当てます。通常、これはソケットファイルディスクリプタになります。

	Name
int	wolfSSL_set_dtls_fd_connected (WOLFSSL * ssl, int fd) この関数は、ファイルディスクリプタ (fd) を SSL 接続の入出力機能として割り当てます。通常、これはソケットファイルディスクリプタになります。これは、ソケットが接続されていることをマークするため、DTLS 固有の API です。この fd に対する recvfrom および sendto 呼び出しは、addr および addr_len パラメータが NULL に設定されます。
int	wolfDTLS_SetChGoodCb (WOLFSSL * ssl, ClientHelloGoodCb cb, void * user_ctx) 正しく処理および検証された DTLS クライアント hello に対するコールバックを設定できます。クッキー交換メカニズム (DTLS 1.2 の HelloVerifyRequest またはクッキー拡張を伴う DTLS 1.3 の HelloRetryRequest) を使用する場合は、このコールバックはクッキー交換が成功した後に呼び出されます。これは、1 つの WOLFSSL オブジェクトを新しい接続のリスナーとして使用し、ClientHello が検証された後 (クッキー交換を通じて、または ClientHello が正しい形式であるかをチェックするだけで) に WOLFSSL オブジェクトを分離できるようにするのに役立ちます。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
void	wolfSSL_set_using_nonblock (WOLFSSL * ssl, int nonblock) この関数は、WOLFSSL オブジェクトに対して、下層の I/O がノンブロッキングであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキングソケットと共に使用する場合は、wolfSSL_set_using_nonblock() を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、recvfrom 呼び出しがブロックすることを意味すると認識できます。
void	wolfSSL_CTX_free (WOLFSSL_CTX * ctx) この関数は、割り当てられた WOLFSSL_CTX オブジェクトを解放します。この関数は CTX 参照カウントをデクリメントし、参照カウントが 0 になった場合にのみコンテキストを解放します。
void	wolfSSL_free (WOLFSSL * ssl) この関数は、割り当てられた wolfSSL オブジェクトを解放します。

	Name
int	wolfSSL_set_session(WOLFSSL * ssl, WOLFSSL_SESSION * session) この関数は、SSL オブジェクト ssl が SSL/TLS 接続を確立するために使用されるときに使用されるセッションを設定します。セッション再開の場合、セッションオブジェクトで wolfSSL_shutdown() を呼び出す前に、アプリケーションは wolfSSL_get1_session() を呼び出してオブジェクトからセッション ID を保存する必要があります。これはセッションへのポインタを返します。後で、アプリケーションは新しい WOLFSSL オブジェクトを作成し、wolfSSL_set_session() を使用して保存されたセッションを割り当てる必要があります。この時点で、アプリケーションは wolfSSL_connect() を呼び出すことができ、wolfSSL はセッションの再開を試みます。wolfSSL サーバーコードは、デフォルトでセッション再開を許可します。 wolfSSL_get1_session() によって返されたオブジェクトは、アプリケーションが使用を終えた後、wolfSSL_SESSION_free() を呼び出して解放する必要があります。

	Name
void	wolfSSL_CTX_set_verify (WOLFSSL_CTX * ctx, int mode, VerifyCallback verify_callback) この関数はリモートピアの検証方法を設定し、また検証コールバックを SSL コンテキストに登録できるようにします。検証コールバックは検証失敗が発生した場合にのみ呼び出されます。検証コールバックが不要な場合は、verify_callback に NULL ポインタを使用できます。ピア証明書の検証モードは論理 OR 演算されたフラグのリストです。可能なフラグ値は以下の通りです。SSL_VERIFY_NONE クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常通り続行されます。サーバーモード：サーバーはクライアントに証明書要求を送信しません。したがって、クライアント検証は有効になりません。SSL_VERIFY_PEER クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトで有効になっているため、このオプションを使用しても効果はありません。サーバーモード：サーバーはクライアントに証明書要求を送信し、受信したクライアント証明書を検証します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：クライアントが要求されたときに証明書の送信に失敗した場合（SSL サーバーで SSL_VERIFY_PEER を使用している場合）、サーバー側で検証が失敗します。SSL_VERIFY_FAIL_EXCEPT_PSK クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じですが、PSK 接続の場合を除きます。PSK 接続が行われている場合、接続はピア証明書なしで続行されます。

	Name
void	wolfSSL_set_verify(WOLFSSL * ssl, int mode, VerifyCallback verify_callback) この関数はリモートピアの検証方法を設定し、また検証コールバックを SSL セッションに登録できるようにします。検証コールバックは検証失敗が発生した場合にのみ呼び出されます。検証コールバックが不要な場合は、verify_callback に NULL ポインタを使用できます。ピア証明書の検証モードは論理 OR 演算されたフラグのリストです。可能なフラグ値は以下の通りです。SSL_VERIFY_NONE クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常通り続行されます。サーバーモード：サーバーはクライアントに証明書要求を送信しません。したがって、クライアント検証は有効になりません。SSL_VERIFY_PEER クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトで有効になっているため、このオプションを使用しても効果はありません。サーバーモード：サーバーはクライアントに証明書要求を送信し、受信したクライアント証明書を検証します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：クライアントが要求されたときに証明書の送信に失敗した場合（SSL サーバーで SSL_VERIFY_PEER を使用している場合）、サーバー側で検証が失敗します。SSL_VERIFY_FAIL_EXCEPT_PSK クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じですが、PSK 接続の場合を除きます。PSK 接続が行われている場合、接続はピア証明書なしで続行されます。
long	wolfSSL_CTX_set_session_cache_mode(WOLFSSL_CTX * ctx, long mode) この関数は SSL セッションキャッシングを有効または無効にします。動作は mode に使用される値に依存します。mode に使用できる値は以下の通りです。SSL_SESS_CACHE_OFF - セッションキャッシングを無効にします。セッションキャッシングはデフォルトで有効になっています。SSL_SESS_CACHE_NO_AUTO_CLEAR - セッションキャッシュの自動フラッシュを無効にします。自動フラッシュはデフォルトで有効になっています。
int	wolfSSL_CTX_memrestore_cert_cache(WOLFSSL_CTX * ctx, const void * mem, int sz) この関数は、証明書キャッシュをメモリから復元します。

	Name
int	wolfSSL_CTX_set_cipher_list (WOLFSSL_CTX * ctx, const char * list) この関数は、指定された WOLFSSL_CTX に対する暗号スイートリストを設定します。設定したリストは、本コンテキストから新規に作成されるすべての WOLFSSL セッションのデフォルトリストとなり、文字列内の順序がそのまま優先順位（高い順から低い順）として用いられます。本関数を呼び出すたびに前回のリストは置き換えられます。リストは null 終端のテキスト文字列で、暗号スイート名および／または OpenSSL 互換のキーワードをコロンで区切って指定します（例: "TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-SHA256"）。
int	**wolfSSL_set_cipher_list の説明を参照してください。本関数を呼び出すたびに、当該セッションのリストは置き換えられます。
void	wolfSSL_dtls13_set_send_more_acks (WOLFSSL * ssl, int value) この関数は、中断を検出したときにライブラリが ACK を他のピアに即座に送信すべきかどうかを設定します。ACK を即座に送信すると最小のレイテンシが保証されますが、必要以上に帯域幅を消費する可能性があります。アプリケーションが自分でタイマーを管理し、このオプションが 0 に設定されている場合、アプリケーションコードは wolfSSL_dtls13_use_quick_timeout() を使用して、これらの遅延 ACK を送信するためにより短いタイムアウトを設定すべきかどうかを判定できます。
int	wolfSSL_dtls_set_timeout_init (WOLFSSL * ssl, int timeout) この関数は dtls タイムアウトを設定します。

	Name
WOLFSSL_SESSION *	wolfSSL_get1_session(WOLFSSL * ssl) この関数は WOLFSSL 構造体から WOLFSSL_SESSION を参照型として返します。これには wolfSSL_SESSION_free を呼び出してセッション参照を解放する必要があります。指される WOLFSSL_SESSION には、セッション再開を実行し、新しいハンドシェイクなしで接続を再確立するために必要なすべての情報が含まれています。セッション再開のために、セッションオブジェクトで wolfSSL_shutdown() を呼び出す前に、アプリケーションは wolfSSL_get1_session() の呼び出しでオブジェクトからセッション ID を保存する必要があります、これはセッションへのポインタを返します。後で、アプリケーションは新しい WOLFSSL オブジェクトを作成し、wolfSSL_set_session() で保存されたセッションを割り当てる必要があります。この時点で、アプリケーションは wolfSSL_connect() を呼び出すことができ、wolfSSL はセッションの再開を試みます。wolfSSL サーバーコードはデフォルトでセッション再開を許可します。wolfSSL_get1_session() によって返されたオブジェクトは、アプリケーションが使用を終えた後、wolfSSL_SESSION_free() を呼び出すことによって解放する必要があります。 wolfSSLv23_client_method(void) wolfSSLv23_client_method() 関数は、アプリケーションがクライアントであり、SSL 3.0 から TLS 1.3 までの間でサーバがサポートする最も高いプロトコルバージョンをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい WOLFSSL_METHOD 構造体のためのメモリを割り当て、初期化します。wolfSSL のクライアントとサーバの両方は、堅牢なバージョンダウングレード機能を持っています。どちらか一方で特定のプロトコルバージョンメソッドが使用された場合、そのバージョンのみがネゴシエートされるか、エラーが返されます。例えば、TLSv1 を使用するクライアントが SSLv3 のみのサーバに接続しようとするとう失敗し、同様に TLSv1.1 への接続も失敗します。この問題を解決するために、wolfSSLv23_client_method() 関数を使用するクライアントは、サーバがサポートする最も高いプロトコルバージョンを使用し、必要に応じて SSLv3 までダウングレードします。この場合、クライアントは SSLv3 から TLSv1.3 までを実行しているサーバに接続できます。
WOLFSSL_METHOD *	

	Name
WOLFSSL_BIGNUM *	wolfSSL_ASN1_INTEGER_to_BN (const WOLFSSL_ASN1_INTEGER * ai, WOLFSSL_BIGNUM * bn) この関数は、WOLFSSL_ASN1_INTEGER 値を WOLFSSL_BIGNUM 構造体にコピーするために使用されます。
long	wolfSSL_CTX_add_extra_chain_cert (WOLFSSL_CTX * ctx, WOLFSSL_X509 * x509) この関数は、WOLFSSL_CTX 構造体内で構築中の内部チェーンに証明書を追加します。
int	wolfSSL_CTX_get_read_ahead (WOLFSSL_CTX * ctx) この関数は、WOLFSSL_CTX 構造体からリードアヘッドフラグを取得して返します。
int	wolfSSL_CTX_set_read_ahead (WOLFSSL_CTX * ctx, int v) この関数は、WOLFSSL_CTX 構造体内のリードアヘッドフラグを設定します。
long	wolfSSL_CTX_set_tlsext_status_arg (WOLFSSL_CTX * ctx, void * arg) この関数は、OCSP で使用するオプション引数を設定します。
long	wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg (WOLFSSL_CTX * ctx, void * arg) この関数は、PRF コールバックに渡されるオプション引数を設定します。
long	wolfSSL_CTX_set_options (WOLFSSL_CTX * ctx, long opt) この関数は、指定された WOLFSSL_CTX のオプションマスクに opt のビットを OR で追加します。設定したオプションマスクは、本コンテキストから後に作成されるすべての WOLFSSL セッションに継承されます。ビットは累積されるため、特定のオプションを解除するには wolfSSL_CTX_clear_options() を使用してください。OpenSSL 形式の "SSL_OP_" マクロは対応する "WOLFSSL_OP_" 値の別名であり、どちらの表記でも使用できます。
long	**wolfSSL_set_options の説明を参照してください。親 WOLFSSL_CTX から継承したフラグはそのまま保持されます。本関数で SSL_OP_NO_TLSv1_3 を設定した場合、当該セッションのネゴシエーション可能なバージョンは TLS 1.2 まで引き下げられます。
long	wolfSSL_get_options (const WOLFSSL * s) この関数は、現在のオプションマスクを返します。
long	wolfSSL_set_tlsext_debug_arg (WOLFSSL * s, void * arg) これは、渡されるデバッグ引数を設定するために使用されます。
long	wolfSSL_get_verify_result (const WOLFSSL * ssl) これは、ピアの証明書の検証を試みた後の結果を取得するために使用されます。
int	wolfSSL_CTX_allow_anon_cipher (WOLFSSL_CTX * ctx) この関数は、コンパイル時に HAVE_ANON が定義されている場合、CTX 構造体の havAnon メンバを有効にします。

	Name
WOLFSSL_METHOD *	wolfSSLv23_server_method (void) wolfSSLv23_server_method() 関数は、アプリケーションがサーバであり、SSL 3.0 から TLS 1.3 までのプロトコルバージョンで接続するクライアントをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい WOLFSSL_METHOD 構造体のためのメモリを割り当て、初期化します。
int	wolfSSL_state (WOLFSSL * ssl) これは、WOLFSSL 構造体の内部エラー状態を取得するために使用されます。
int	wolfSSL_check_domain_name (WOLFSSL * ssl, const char * dn) wolfSSL はデフォルトでピア証明書の有効な日付範囲と検証済み署名をチェックします。wolfSSL_connect() または wolfSSL_accept() の前にこの関数を呼び出すと、実行するチェックのリストにドメイン名チェックが追加されます。dn は、受信したピア証明書に対してチェックするドメイン名を保持します。
int	**wolfSSL_check_ip_address の前にこの関数を呼ぶと、ピア証明書の SAN ipAddress エントリに対する IP アドレス検証を追加します。
int	wolfSSL_set_compression (WOLFSSL * ssl) SSL 接続で圧縮を使用する機能をオンにします。両側で圧縮をオンにする必要があります。そうでない場合、圧縮は使用されません。zlib ライブラリが実際のデータ圧縮を実行します。ライブラリにコンパイルするには、configure システムで -with-libz を使用し、HAVE_LIBZ を定義してください。送信前にデータを圧縮すると送受信されるメッセージの実際のサイズは減少しますが、圧縮によって節約されるデータの量は、最も遅いネットワークを除いて、生データで送信するよりも解析に時間がかかることに注意してください。
int	wolfSSL_set_timeout (WOLFSSL * ssl, unsigned int to) この関数は SSL セッションのタイムアウト値を秒単位で設定します。
int	wolfSSL_CTX_set_timeout (WOLFSSL_CTX * ctx, unsigned int to) この関数は、指定された SSL コンテキストの SSL セッションのタイムアウト値を秒単位で設定します。
int	wolfSSL_CTX_UnloadCAs (WOLFSSL_CTX * ctx) この関数は、CA 署名者リストをアンロードし、署名者テーブル全体を解放します。
int	wolfSSL_CTX_UnloadIntermediateCerts (WOLFSSL_CTX * ctx) この関数は、CA 署名者リストに追加された中間証明書をアンロードし、解放します。

	Name
int	wolfSSL_CTX_Unload_trust_peers (WOLFSSL_CTX * ctx) この関数は以前にロードされたすべての信頼されたピア証明書をアンロードするために使用されます。この機能は、マクロ WOLFSSL_TRUST_PEER_CERT を定義することで有効になります。
int	wolfSSL_CTX_trust_peer_buffer (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は、TLS/SSL ハンドシェイクを実行する際にピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されるピア証明書は、利用可能な場合は SKID と署名を使用して比較されます。これら 2 つが一致しない場合は、ロードされた CA が使用されます。ファイルではなくバッファからの入力である点を除いて、wolfSSL_CTX_trust_peer_cert と同じ機能です。この機能は、マクロ WOLFSSL_TRUST_PEER_CERT を定義することで有効になります。適切な使用方法については例を参照してください。
int	wolfSSL_CTX_set_group_messages (WOLFSSL_CTX * ctx) この関数は、可能な場合にハンドシェイクメッセージのグループ化をオンにします。
int	wolfSSL_set_group_messages (WOLFSSL * ssl) この関数は、可能な場合にハンドシェイクメッセージのグループ化をオンにします。
int	wolfSSL_CTX_SetMinVersion (WOLFSSL_CTX * ctx, int version) この関数は、許可される最小ダウングレードバージョンを設定します。ダウングレードを許可する接続 (wolfSSLv23_client_method または wolfSSLv23_server_method) を使用する場合にのみ適用されます。
int	wolfSSL_SetVersion (WOLFSSL * ssl, int version) この関数は、version で指定されたバージョンを使用して、指定された SSL セッション (WOLFSSL オブジェクト) の SSL/TLS プロトコルバージョンを設定します。これにより、SSL セッション (ssl) のプロトコル設定が上書きされます。これは元々 SSL コンテキスト (wolfSSL_CTX_new()) のメソッドタイプによって定義および設定されたものです。
int	wolfSSL_UseALPN (WOLFSSL * ssl, char * protocol_name_list, unsigned int protocol_name_listSz, unsigned char options) wolfSSL セッションで ALPN の使用をセットアップします。
int	wolfSSL_CTX_UseSessionTicket (WOLFSSL_CTX * ctx) この関数は、wolfSSL コンテキストにセッションチケットを使用するように設定します。
int	wolfSSL_check_private_key (const WOLFSSL * ssl) この関数は、秘密鍵が使用されている証明書と一致していることを確認します。

	Name
int	wolfSSL_use_certificate (WOLFSSL * ssl, WOLFSSL_X509 * x509) ハンドシェイク中に使用する WOLFSSL 構造体の証明書を設定するために使用されます。
int	wolfSSL_use_certificate_ASN1 (WOLFSSL * ssl, const unsigned char * der, int derSz) ハンドシェイク中に使用する WOLFSSL 構造体の証明書を設定するために使用されます。DER 形式のバッファが必要です。
int	wolfSSL_SESSION_get_master_key (const WOLFSSL_SESSION * ses, unsigned char * out, int outSz) ハンドシェイク完了後にマスターキーを取得するために使用されます。
int	wolfSSL_SESSION_get_master_key_length (const WOLFSSL_SESSION * ses) マスターシークレットキーの長さを取得するために使用されます。
void	wolfSSL_CTX_set_cert_store (WOLFSSL_CTX * ctx, WOLFSSL_X509_STORE * str) ctx の WOLFSSL_X509_STORE 構造体の setter 関数です。
WOLFSSL_X509_STORE *	wolfSSL_CTX_get_cert_store (WOLFSSL_CTX * ctx) ctx の WOLFSSL_X509_STORE 構造体の getter 関数です。
size_t	wolfSSL_get_server_random (const WOLFSSL * ssl, unsigned char * out, size_t outlen) ハンドシェイク中にサーバーから送信されたランダムデータを取得するために使用されます。
size_t	wolfSSL_get_client_random (const WOLFSSL * ssl, unsigned char * out, size_t outSz) ハンドシェイク中にクライアントから送信されたランダムデータを取得するために使用されます。
wc_pem_password_cb *	wolfSSL_CTX_get_default_passwd_cb (WOLFSSL_CTX * ctx) ctx 内に設定されたパスワードコールバックの getter 関数です。
void *	wolfSSL_CTX_get_default_passwd_cb_userdata (WOLFSSL_CTX * ctx) ctx 内に設定されたパスワードコールバックユーザーデータの getter 関数です。
long	wolfSSL_CTX_clear_options (WOLFSSL_CTX * ctx, long opt) この関数は、WOLFSSL_CTX オブジェクトのオプションビットをリセットします。
int	wolfSSL_set_msg_callback (WOLFSSL * ssl, SSL_Msg_Cb cb) この関数は、ssl 内にコールバックを設定します。コールバックはハンドシェイクメッセージを監視するためのものです。cb の NULL 値はコールバックをリセットします。
int	wolfSSL_set_msg_callback_arg (WOLFSSL * ssl, void * arg) この関数は、ssl に関連するコールバックコンテキスト値を設定します。値はコールバック引数に渡されます。

	Name
int	wolfSSL_send_hrr_cookie (WOLFSSL * ssl, const unsigned char * secret, unsigned int secretSz) この関数は、HelloRetryRequest メッセージに Cookie を含める必要があることを示すために、サーバ側で呼び出されます。また、プロトコル DTLS v1.3 を使用する場合、ハンドシェイクには常に Cookie 交換が含まれることを示します。プロトコル DTLS v1.3 を使用する場合、Cookie 交換はデフォルトで有効になっていることに注意してください。Cookie は現在のトランスクリプトのハッシュを保持しており、別のサーバプロセスが応答の ClientHello を処理できるようにします。secret は、Cookie データの完全性チェックを生成する際に使用されます。
int	wolfSSL_disable_hrr_cookie (WOLFSSL * ssl) この関数は、HelloRetryRequest メッセージに Cookie を含めてはならないこと、およびプロトコル DTLS v1.3 を使用している場合、ハンドシェイクに Cookie 交換を含めないことを示すために、サーバ側で呼び出されます。プロトコル DTLS v1.3 を使用する際に Cookie 交換を行わないと、サーバが DoS/増幅攻撃に対して脆弱になる可能性があることに注意してください。
int	wolfSSL_CTX_no_ticket_TLSv13 (WOLFSSL_CTX * ctx) この関数は、ハンドシェイクが完了した後、再開セッションチケットの送信を停止するために、サーバで呼び出されます。
int	wolfSSL_no_ticket_TLSv13 (WOLFSSL * ssl) この関数は、ハンドシェイクが完了した後、再開セッションチケットの送信を停止するために、サーバで呼び出されます。
int	wolfSSL_CTX_no_dhe_psk (WOLFSSL_CTX * ctx) この関数は、事前共有鍵を認証に使用するハンドシェイクの際に、Diffie-Hellman(DH) スタイルの鍵交換を禁止するために、TLS v1.3 wolfSSL コンテキストで呼び出されます。
int	wolfSSL_no_dhe_psk (WOLFSSL * ssl) この関数は、事前共有鍵を認証に使用するハンドシェイクの際に、Diffie-Hellman(DH) スタイルの鍵交換を禁止するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。
int	wolfSSL_CTX_allow_post_handshake_auth (WOLFSSL_CTX * ctx) この関数は、サーバからの要求に応じてクライアント証明書をハンドシェイク後に送信できるようにするために、TLS v1.3 クライアント wolfSSL コンテキストで呼び出されます。これは、クライアント認証が必要なページとそうでないページを持つ Web サーバに接続する際に便利です。

	Name
int	wolfSSL_allow_post_handshake_auth (WOLFSSL * ssl) この関数は、サーバからの要求に応じてクライアント証明書をハンドシェイク後に送信できるようにするために、TLS v1.3 クライアント wolfSSL で呼び出されます。Post-Handshake Client Authentication 拡張が ClientHello で送信されます。これは、クライアント認証が必要なページとそうでないページを持つ Web サーバに接続する際に便利です。
int	wolfSSL_CTX_set1_sigalgs_list (WOLFSSL_CTX * ctx, const char * list) この関数は、wolfSSL コンテキストで優先順位に従って許可する署名アルゴリズムのリストを設定します。リストは null 終端のテキスト文字列で、コロン区切りのリストです。各要素は "+" の形式 (例: "RSA-PSS+SHA256:ECDSA+SHA384") で指定します。Edwards 曲線アルゴリズムである "ED25519" および "ED448" はハッシュがアルゴリズムに内包されるため、サフィックスを付けずに記述します。コンテキストに既に保持されているリストは置き換えられます。
int	** wolfSSL_set1_sigalgs_list の説明を参照してください。セッションに既に保持されているリストは置き換えられます。
int	wolfSSL_CTX_set1_groups_list (WOLFSSL_CTX * ctx, const char * list) この関数は、wolfSSL コンテキストで優先順位に従って許可する鍵交換グループ (名前付き楕円曲線および KEM) のリストを設定します。リストは null 終端のテキスト文字列で、グループ名をコロンで区切って指定します (例: "P-384:P-256:X25519")。TLS v1.3 接続で使用する鍵交換パラメータを設定するために本関数を呼び出します (本関数は HAVE_ECC、WOLFSSL_TLS13、HAVE_SUPPORTED_CURVES がすべて定義された場合にのみ提供されます)。
int	** wolfSSL_set1_groups_list の説明を参照してください。
int	wolfSSL_CTX_set_groups (WOLFSSL_CTX * ctx, int * groups, int count) この関数は、wolfSSL コンテキストで優先順位に従って許可する鍵交換グループ (名前付き楕円曲線および KEM) のリストを設定します。リストは名前付きグループ識別子 (下表参照) の配列で、count に配列内の識別子数を指定します。TLS v1.3 接続で使用する鍵交換パラメータを設定するために本関数を呼び出します。配列の順序がそのまま KeyShare の優先順位として用いられます。
int	** wolfSSL_set_groups の説明を参照してください。

	Name
int	wolfSSL_CTX_set_max_early_data (WOLFSSL_CTX * ctx, unsigned int sz) この関数は、TLS v1.3 クライアントまたはサーバが wolfSSL コンテキストを使用して交換する意思のある早期データの最大量を設定します。リプレイ攻撃を軽減するために処理する早期データの量を制限するには、この関数を呼び出してください。早期データは、セッションチケットが送信された接続の鍵から派生した鍵によって保護されるため、セッションチケットが再開に使用されるたびに同じになります。この値は再開用のセッションチケットに含まれます。サーバの値がゼロの場合、セッションチケットを使用してクライアントが早期データを送信しないことを示します。クライアントの値がゼロの場合、クライアントが早期データを送信しないことを示します。早期データのバイト数は、アプリケーションで実用的に可能な限り低く保つことをお勧めします。
int	wolfSSL_set_max_early_data (WOLFSSL * ssl, unsigned int sz) この関数は、TLS v1.3 クライアントまたはサーバが交換する意思のある早期データの最大量を設定します。リプレイ攻撃を軽減するために処理する早期データの量を制限するには、この関数を呼び出してください。早期データは、セッションチケットが送信された接続の鍵から派生した鍵によって保護されるため、セッションチケットが再開に使用されるたびに同じになります。この値は再開用のセッションチケットに含まれます。サーバの値がゼロの場合、セッションチケットを使用してクライアントが早期データを送信しないことを示します。クライアントの値がゼロの場合、クライアントが早期データを送信しないことを示します。早期データのバイト数は、アプリケーションで実用的に可能な限り低く保つことをお勧めします。
void	wolfSSL_CTX_set_psk_client_tls13_callback (WOLFSSL_CTX * ctx, wc_psk_client_tls13_callback cb) この関数は TLS v1.3 接続のための事前共有鍵 (PSK) クライアント側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL_CTX 構造体の client_psk_tls13_cb メンバーを設定します。
void	wolfSSL_set_psk_client_tls13_callback (WOLFSSL * ssl, wc_psk_client_tls13_callback cb) この関数は TLS v1.3 接続のための事前共有鍵 (PSK) クライアント側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL 構造体の options フィールドの client_psk_tls13_cb メンバーを設定します。

	Name
void	wolfSSL_CTX_set_psk_server_tls13_callback (WOLFSSL_CTX * ctx, wc_psk_server_tls13_callback cb) この関数は TLS v1.3 接続のための事前共有鍵 (PSK) サーバー側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL_CTX 構造体の server_psk_tls13_cb メンバーを設定します。
void	wolfSSL_set_psk_server_tls13_callback (WOLFSSL * ssl, wc_psk_server_tls13_callback cb) この関数は TLS v1.3 接続のための事前共有鍵 (PSK) サーバー側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL 構造体の options フィールドの server_psk_tls13_cb メンバーを設定します。
int	wolfSSL_UseKeyShare (WOLFSSL * ssl, word16 group) この関数は鍵ペアの生成を含む、グループから鍵共有エントリを作成します。KeyShare 拡張には鍵交換用に生成されたすべての公開鍵が含まれます。この関数が呼び出されると、指定されたグループのみが含まれます。サーバーに対して優先グループが以前に確立されている場合に、この関数を呼び出します。
int	wolfSSL_NoKeyShares (WOLFSSL * ssl) この関数は ClientHello で鍵共有が送信されないようにするために呼び出されます。これにより、ハンドシェイクで鍵交換が必要な場合、サーバーは HelloRetryRequest で応答することになります。期待される鍵交換グループが不明で、不要な鍵の生成を避けたい場合にこの関数を呼び出します。鍵交換が必要な場合、ハンドシェイクを完了するために追加のラウンドトリップが必要になることに注意してください。
WOLFSSL_METHOD *	wolfTLSv1_3_server_method_ex (void * heap) この関数はアプリケーションがサーバーであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。
WOLFSSL_METHOD *	wolfTLSv1_3_client_method_ex (void * heap) この関数はアプリケーションがクライアントであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。

	Name
WOLFSSL_METHOD *	wolfTLSv1_3_server_method (void) この関数は、アプリケーションがサーバーであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体用のメモリを割り当て、初期化します。
WOLFSSL_METHOD *	wolfTLSv1_3_client_method (void) この関数は、アプリケーションがクライアントであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体用のメモリを割り当て、初期化します。
WOLFSSL_METHOD *	wolfTLSv1_3_method_ex (void * heap) この関数は、どちら側 (サーバー/クライアント) であるかがまだ決定されていないことを除いて、wolfTLSv1_3_client_method と同様の WOLFSSL_METHOD を返します。
WOLFSSL_METHOD *	wolfTLSv1_3_method (void) この関数は、どちら側 (サーバー/クライアント) であるかがまだ決定されていないことを除いて、wolfTLSv1_3_client_method と同様の WOLFSSL_METHOD を返します。
int	wolfSSL_CTX_set_client_cert_type (WOLFSSL_CTX * ctx, const char * buf, int len) この関数がクライアント側で呼び出された場合、ピアに送信できる証明書タイプを設定します。サーバー側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が "Raw public key" 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。
int	wolfSSL_CTX_set_server_cert_type (WOLFSSL_CTX * ctx, const char * buf, int len) この関数がサーバー側で呼び出された場合、ピアに送信できる証明書タイプを設定します。クライアント側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が "Raw public key" 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。

	Name
int	wolfSSL_set_client_cert_type (WOLFSSL * ssl, const char * buf, int len) この関数がクライアント側で呼び出された場合、ピアに送信できる証明書タイプを設定します。サーバー側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が "Raw public key" 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。
int	wolfSSL_set_server_cert_type (WOLFSSL * ssl, const char * buf, int len) この関数がサーバー側で呼び出された場合、ピアに送信できる証明書タイプを設定します。クライアント側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が "Raw public key" 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。
int	wolfSSL_CTX_clear_group_messages (WOLFSSL_CTX * ctx) 指定された WOLFSSL_CTX コンテキストに対してハンドシェイクメッセージグループ化を無効にします。
int	wolfSSL_clear_group_messages (WOLFSSL * ssl) 指定された WOLFSSL オブジェクトに対してハンドシェイクメッセージグループ化を無効にします。
void *	wolfSSL_GetCookieCtx (WOLFSSL * ssl) この関数は WOLFSSL 構造体の IOCB_CookieCtx メンバーを返します。
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) この関数は、wolfSSL が WOLFSSL_ISOTP でコンパイルされている場合に使用するために、wolfSSL の ISO-TP コンテキストを設定します。
void	wolfSSL_SSLEnableRead (WOLFSSL * ssl) この関数は IO レイヤーからの読み取りを無効にします。

	Name
void	wolfSSL_SSLEnableRead (WOLFSSL * ssl) この関数は IO レイヤーからの読み取りを有効にします。読み取りはデフォルトで有効になっており、wolfSSL_SSLEnableRead() を元に戻すために使用する必要があります。

A.6.2 Functions Documentation

A.6.2.1 function wolfSSLv23_method

```
WOLFSSL_METHOD * wolfSSLv23_method(  
    void  
)
```

この関数は wolfSSLv23_client_method に似た WOLFSSL_METHOD を返しますが、どちら側 (サーバ/クライアント) であるかがまだ決定されていない点が異なります。

Parameters:

- **none** パラメータはありません。

See:

- **wolfSSL_new**
- **wolfSSL_free**

Return:

- WOLFSSL_METHOD* 正常に作成された場合、WOLFSSL_METHOD ポインタを返します
- NULL メモリ割り当てエラーまたはメソッド作成失敗の場合は Null を返します

Example

```
WOLFSSL* ctx;  
ctx = wolfSSL_CTX_new(wolfSSLv23_method());  
// ret 値をチェック
```

A.6.2.2 function wolfSSLv3_server_method

```
WOLFSSL_METHOD * wolfSSLv3_server_method(  
    void  
)
```

wolfSSLv3_server_method() 関数は、アプリケーションがサーバであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_server_method();
if (method == NULL) {
    メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.3 function wolfSSLv3_client_method

```
WOLFSSL_METHOD * wolfSSLv3_client_method(
    void
)
```

wolfSSLv3_client_method() 関数は、アプリケーションがクライアントであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_client_method();
```



```

if (method == NULL) {
    メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

A.6.2.4 function wolfTLSv1_server_method

```

WOLFSSL_METHOD * wolfTLSv1_server_method(
    void
)

```

wolfTLSv1_server_method() 関数は、アプリケーションがサーバであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```

#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_server_method();
if (method == NULL) {
    メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

A.6.2.5 function wolfTLSv1_client_method

```

WOLFSSL_METHOD * wolfTLSv1_client_method(
    void
)

```

wolfTLSv1_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_client_method();
if (method == NULL) {
    メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.6 function wolfTLSv1_1_server_method

```
WOLFSSL_METHOD * wolfTLSv1_1_server_method(
    void
)
```

wolfTLSv1_1_server_method() 関数は、アプリケーションがサーバであり、TLS 1.1 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_server_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.7 function wolfTLSv1_1_client_method

```
WOLFSSL_METHOD * wolfTLSv1_1_client_method(
    void
)
```

wolfTLSv1_1_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_client_method();
```

```

if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

A.6.2.8 function wolfTLSv1_2_server_method

```

WOLFSSL_METHOD * wolfTLSv1_2_server_method(
    void
)

```

wolfTLSv1_2_server_method() 関数は、アプリケーションがサーバであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```

#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_server_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

A.6.2.9 function wolfTLSv1_2_client_method

```

WOLFSSL_METHOD * wolfTLSv1_2_client_method(
    void
)

```

wolfTLSv1_2_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_client_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.10 function wolfDTLSv1_client_method

```
WOLFSSL_METHOD * wolfDTLSv1_client_method(
    void
)
```

wolfDTLSv1_client_method() 関数は、アプリケーションがクライアントであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポート付きでコンパイルされている場合 (-enable-dtls、または wolfSSL_DTLS を定義することによって) にのみ利用可能です。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、この呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます（通常は NULL で、errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_client_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.11 function wolfDTLSv1_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_server_method(
    void
)
```

wolfDTLSv1_server_method() 関数は、アプリケーションがサーバーであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です（-enable-dtls、または wolfSSL_DTLS を定義することによって）。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、この呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます（通常は NULL で、errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
```

```
method = wolfDTLSv1_server_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.12 function wolfDTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_server_method(
    void
)
```

wolfDTLSv1_3_server_method() 関数は、アプリケーションがサーバーであり、DTLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLSv1.3 サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls13、または wolfSSL_DTLS13 を定義することによって)。

Parameters:

- **none** パラメータはありません。

See: [wolfDTLSv1_3_client_method](#)

Return:

- * 成功した場合、この呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL_XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_server_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.13 function wolfDTLSv1_3_client_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_client_method(
    void
)
```

wolfDTLSv1_3_client_method() 関数は、アプリケーションがクライアントであり、DTLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLSv1.3 サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls13、または wolfSSL_DTLS13 を定義することによって)。

Parameters:

- **none** パラメータはありません。

See: [wolfDTLSv1_3_server_method](#)

Return:

- * 成功した場合、この呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL_XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます（通常は NULL で、errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_client_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.14 function wolfDTLS_server_method

```
WOLFSSL_METHOD * wolfDTLS_server_method(
    void
)
```

wolfDTLS_server_method() 関数は、アプリケーションがサーバーであり、利用可能な最新バージョンの DTLS および許可される最小バージョンまでのすべてのバージョンをサポートすることを示すために使用されます。デフォルトの許可される最小バージョンは、WOLFSSL_MIN_DTLS_DOWNGRADE の定義に基づいており、wolfSSL_SetMinVersion() を使用して実行時に変更できます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です（-enable-dtls、または wolfSSL_DTLS を定義することによって）。

Parameters:

- **none** パラメータはありません。

See:

- [wolfDTLS_client_method](#)
- [wolfSSL_SetMinVersion](#)

Return:

- * 成功した場合、この呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL_XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます（通常は NULL で、errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_server_method();
if (method == NULL) {
```



```
// メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.15 function wolfDTLS_client_method

```
WOLFSSL_METHOD * wolfDTLS_client_method(
    void
)
```

wolfDTLS_client_method() 関数は、アプリケーションがクライアントであり、利用可能な最新バージョンの DTLS および許可される最小バージョンまでのすべてのバージョンをサポートすることを示すために使用されます。デフォルトの許可される最小バージョンは、WOLFSSL_MIN_DTLS_DOWNGRADE の定義に基づいており、wolfSSL_SetMinVersion() を使用して実行時に変更できます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です（-enable-dtls、または wolfSSL_DTLS を定義することによって）。

Parameters:

- **none** パラメータはありません。

See:

- [wolfDTLS_server_method](#)
- [wolfSSL_SetMinVersion](#)

Return:

- * 成功した場合、この呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL_XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます（通常は NULL で、errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_client_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.16 function wolfSSL_use_old_poly

```
int wolfSSL_use_old_poly(
    WOLFSSL * ssl,
    int value
)
```

chacha-poly AEAD 構築の最初のリリースと新しいバージョンの間にはいくつかの違いがあるため、古いバージョンを使用するサーバー/クライアントと通信するためのオプションを追加しました。デフォルトでは、wolfSSL は新しいバージョンを使用します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **value** `poly1305` の情報を設定する際に古いバージョンを使用するかどうか。フラグ値 1 を渡すと古い `poly AEAD` を使用することを示し、新しいバージョンの使用に戻すにはフラグ値 0 を渡します。

See: none**Return:** 0 成功時*Example*

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_use_old_poly(ssl, 1);
if (ret != 0) {
    // poly1305 AEAD バージョンの設定に失敗しました
}
```

A.6.2.17 function wolfSSL_CTX_trust_peer_cert

```
int wolfSSL_CTX_trust_peer_cert(
    WOLFSSL_CTX * ctx,
    const char * file,
    int type
)
```

この関数は、TLS/SSL ハンドシェイクを実行する際にピアを検証するために使用する証明書を読み込みます。ハンドシェイク中に送信されるピア証明書は、利用可能な場合は SKID を使用し、署名を使用して比較されます。これら 2 つが一致しない場合は、読み込まれた CA が使用されます。この機能はマクロ `WOLFSSL_TRUST_PEER_CERT` を定義することで有効になります。適切な使用方法については、examples を参照してください。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された SSL コンテキストへのポインタ。
- **file** 証明書を含むファイル名へのポインタ。
- **type** 読み込まれる証明書のタイプ、すなわち `SSL_FILETYPE_ASN1` または `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:

- 成功時は `SSL_SUCCESS`。
- `ctx` が `NULL` の場合、または `file` と `type` の両方が無効な場合は `SSL_FAILURE` が返されます。
- ファイルの形式が間違っている場合は `SSL_BAD_FILETYPE` が返されます。
- ファイルが存在しない、読み取れない、または破損している場合は `SSL_BAD_FILE` が返されます。
- メモリ不足状態が発生した場合は `MEMORY_E` が返されます。

- ファイルに対する Base16 デコードが失敗した場合は ASN_INPUT_E が返されます。

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) {
    // 信頼されたピア証明書の読み込みエラー
}
...
```

A.6.2.18 function wolfSSL_CTX_get_verify_depth

```
long wolfSSL_CTX_get_verify_depth(
    WOLFSSL_CTX * ctx
)
```

この関数は、CTX 構造体を使用して証明書チェーンの深度を取得します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- `wolfSSL_CTX_use_certificate_chain_file`
- `wolfSSL_get_verify_depth`

Return:

- CTX 構造体が NULL でない場合は MAX_CHAIN_DEPTH が返されます。最大証明書チェーンピア深度の定数表現です。
- CTX 構造体が NULL の場合は BAD_FUNC_ARG が返されます。

Example

```
WOLFSSL_METHOD method; // プロトコルメソッド
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
...
long ret = wolfSSL_CTX_get_verify_depth(ctx);

if(ret == EXPECTED){
    // 期待値を取得しました
} else {
    // 予期しない深度を処理
}
```

A.6.2.19 function wolfSSL_CTX_new

```
WOLFSSL_CTX * wolfSSL_CTX_new(
    WOLFSSL_METHOD *
)
```

この関数は、入力として希望する SSL/TLS プロトコルメソッドを取り、新しい SSL コンテキストを作成します。

Parameters:

- **method** SSL コンテキストに使用する希望の WOLFSSL_METHOD へのポインタ。これは、SSL/TLS/DTLS プロトコルレベルを指定するための wolfSSLvXX_XXXX_method() 関数のいずれかを使用して作成されます。

See: [wolfSSL_new](#)

Return:

- pointer 成功した場合、呼び出しは新しく作成された WOLFSSL_CTX へのポインタを返します。
- NULL 失敗時。

Example

```
WOLFSSL_CTX*   ctx    = 0;
WOLFSSL_METHOD* method = 0;

method = wolfSSLv3_client_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキストの作成に失敗しました
}
```

A.6.2.20 function wolfSSL_new

```
WOLFSSL * wolfSSL_new(
    WOLFSSL_CTX *
)
```

この関数は、既に作成された SSL コンテキストを入力として取り、新しい SSL セッションを作成します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された SSL コンテキストへのポインタ。

See: [wolfSSL_CTX_new](#)

Return:

- * 成功した場合、呼び出しは新しく作成された wolfSSL 構造体へのポインタを返します。
- NULL 失敗時。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL*       ssl = NULL;
WOLFSSL_CTX*   ctx = 0;

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキストの作成に失敗しました
}

ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // SSL オブジェクトの作成に失敗しました
}
```

A.6.2.21 function wolfSSL_set_fd

```
int wolfSSL_set_fd(  
    WOLFSSL * ssl,  
    int fd  
)
```

この関数は、ファイルディスクリプタ (fd) を SSL 接続の入出力機能として割り当てます。通常、これはソケットファイルディスクリプタになります。

Parameters:

- **ssl** `wolfSSL_new()` で作成された SSL セッションへのポインタ。
- **fd** SSL/TLS 接続で使用するファイルディスクリプタ。

See:

- `wolfSSL_CTX_SetIOSend`
- `wolfSSL_CTX_SetIORecv`
- `wolfSSL_SetIOReadCtx`
- `wolfSSL_SetIOWriteCtx`

Return:

- `SSL_SUCCESS` 成功時。
- `BAD_FUNC_ARG` 失敗時。

Example

```
int sockfd;  
WOLFSSL* ssl = 0;  
...  
  
ret = wolfSSL_set_fd(ssl, sockfd);  
if (ret != SSL_SUCCESS) {  
    // SSL ファイルディスクリプタの設定に失敗しました  
}
```

A.6.2.22 function wolfSSL_set_dtls_fd_connected

```
int wolfSSL_set_dtls_fd_connected(  
    WOLFSSL * ssl,  
    int fd  
)
```

この関数は、ファイルディスクリプタ (fd) を SSL 接続の入出力機能として割り当てます。通常、これはソケットファイルディスクリプタになります。これは、ソケットが接続されていることをマークするため、DTLS 固有の API です。この fd に対する `recvfrom` および `sendto` 呼び出しは、`addr` および `addr_len` パラメータが `NULL` に設定されます。

Parameters:

- **ssl** `wolfSSL_new()` で作成された SSL セッションへのポインタ。
- **fd** SSL/TLS 接続で使用するファイルディスクリプタ。

See:

- `wolfSSL_CTX_SetIOSend`
- `wolfSSL_CTX_SetIORecv`
- `wolfSSL_SetIOReadCtx`
- `wolfSSL_SetIOWriteCtx`
- `wolfDTLS_SetChGoodCb`

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG 失敗時。

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
if (connect(sockfd, peer_addr, peer_addr_len) != 0) {
    // 接続エラーを処理
}
...
ret = wolfSSL_set_dtls_fd_connected(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // SSL ファイルディスクリプタの設定に失敗しました
}
```

A.6.2.23 function wolfDTLS_SetChGoodCb

```
int wolfDTLS_SetChGoodCb(
    WOLFSSL * ssl,
    ClientHelloGoodCb cb,
    void * user_ctx
)
```

正しく処理および検証された DTLS クライアント hello に対するコールバックを設定できます。クッキー交換メカニズム (DTLS 1.2 の HelloVerifyRequest またはクッキー拡張を伴う DTLS 1.3 の HelloRetryRequest) を使用する場合、このコールバックはクッキー交換が成功した後に呼び出されます。これは、1 つの WOLFSSL オブジェクトを新しい接続のリスナーとして使用し、ClientHello が検証された後 (クッキー交換を通じて、または ClientHello が正しい形式であることをチェックするだけで) に WOLFSSL オブジェクトを分離できるようにするのに役立ちます。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** `wolfSSL_new()` で作成された SSL セッションへのポインタ。
- **fd** SSL/TLS 接続で使用するファイルディスクリプタ。

See: `wolfSSL_set_dtls_fd_connected`

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG 失敗時。

Example

```
// 接続を検証したときに呼び出されます
static int chGoodCb(WOLFSSL* ssl, void* arg)
{
    // ピアとファイルディスクリプタを設定
}

if (wolfDTLS_SetChGoodCb(ssl, chGoodCb, NULL) != WOLFSSL_SUCCESS) {
    // コールバック設定エラー
}
```

A.6.2.24 function wolfSSL_set_using_nonblock

```
void wolfSSL_set_using_nonblock(  
    WOLFSSL * ssl,  
    int nonblock  
)
```

この関数は、WOLFSSL オブジェクトに対して、下層の I/O がノンブロッキングであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキングソケットと共に使用する場合は、wolfSSL_set_using_nonblock() を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、recvfrom 呼び出しがブロックすることを意味すると認識できます。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **nonblock** WOLFSSL オブジェクトのノンブロッキングフラグを設定するための値。ノンブロッキングを指定する場合は 1 を、そうでない場合は 0 を使用してください。

See:

- wolfSSL_get_using_nonblock
- wolfSSL_dtls_get_timeout
- wolfSSL_dtls_get_current_timeout

Return: none 戻り値はありません。

Example

```
WOLFSSL* ssl = 0;  
...  
wolfSSL_set_using_nonblock(ssl, 1);
```

A.6.2.25 function wolfSSL_CTX_free

```
void wolfSSL_CTX_free(  
    WOLFSSL_CTX * ctx  
)
```

この関数は、割り当てられた WOLFSSL_CTX オブジェクトを解放します。この関数は CTX 参照カウントをデクリメントし、参照カウントが 0 になった場合にのみコンテキストを解放します。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: none 戻り値はありません。

Example

```
WOLFSSL_CTX* ctx = 0;  
...  
wolfSSL_CTX_free(ctx);
```

A.6.2.26 function wolfSSL_free

```
void wolfSSL_free(  
    WOLFSSL * ssl  
)
```

この関数は、割り当てられた wolfSSL オブジェクトを解放します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)で作成された SSL オブジェクトへのポインタ。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_new](#)
- [wolfSSL_CTX_free](#)

Return: none 戻り値はありません。

Example

```
#include <wolfssl/ssl.h>  
  
WOLFSSL* ssl = 0;  
...  
wolfSSL_free(ssl);
```

A.6.2.27 function wolfSSL_set_session

```
int wolfSSL_set_session(  
    WOLFSSL * ssl,  
    WOLFSSL_SESSION * session  
)
```

この関数は、SSL オブジェクト `ssl` が SSL/TLS 接続を確立するために使用されるときに使用されるセッションを設定します。セッション再開の場合、セッションオブジェクトで `wolfSSL_shutdown()` を呼び出す前に、アプリケーションは `wolfSSL_get1_session()` を呼び出してオブジェクトからセッション ID を保存する必要があります。これはセッションへのポインタを返します。後で、アプリケーションは新しい WOLFSSL オブジェクトを作成し、`wolfSSL_set_session()` を使用して保存されたセッションを割り当てる必要があります。この時点で、アプリケーションは `wolfSSL_connect()` を呼び出すことができ、wolfSSL はセッションの再開を試みます。wolfSSL サーバーコードは、デフォルトでセッション再開を許可します。`wolfSSL_get1_session()` によって返されたオブジェクトは、アプリケーションが使用を終えた後、`wolfSSL_SESSION_free()` を呼び出して解放する必要があります。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)で作成された SSL オブジェクトへのポインタ。
- **session** ssl のセッションを設定するために使用される WOLFSSL_SESSION へのポインタ。

See: [wolfSSL_get1_session](#)**Return:**

- SSL_SUCCESS は、セッションの設定に成功した場合に返されます。
- SSL_FAILURE は、失敗時に返されます。これは、セッションキャッシュが無効になっている場合、またはセッションがタイムアウトした場合に発生する可能性があります。
- OPENSSL_EXTRA と WOLFSSL_ERROR_CODE_OPENSSL が定義されている場合、セッションがタイムアウトしても SSL_SUCCESS が返されます。

Example


```

int ret;
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get1_session(ssl);
if (session == NULL) {
    // ssl オブジェクトからセッションオブジェクトの取得に失敗しました
}
...
ret = wolfSSL_set_session(ssl, session);
if (ret != SSL_SUCCESS) {
    // SSL セッションの設定に失敗しました
}
wolfSSL_SESSION_free(session);
...

```

A.6.2.28 function wolfSSL_CTX_set_verify

```

void wolfSSL_CTX_set_verify(
    WOLFSSL_CTX * ctx,
    int mode,
    VerifyCallback verify_callback
)

```

この関数はリモートピアの検証方法を設定し、また検証コールバックを SSL コンテキストに登録できるようにします。検証コールバックは検証失敗が発生した場合にのみ呼び出されます。検証コールバックが不要な場合は、verify_callback に NULL ポインタを使用できます。ピア証明書の検証モードは論理 OR 演算されたフラグのリストです。可能なフラグ値は以下の通りです。SSL_VERIFY_NONE クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常通り続行されます。サーバーモード：サーバーはクライアントに証明書要求を送信しません。したがって、クライアント検証は有効になりません。SSL_VERIFY_PEER クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトで有効になっているため、このオプションを使用しても効果はありません。サーバーモード：サーバーはクライアントに証明書要求を送信し、受信したクライアント証明書を検証します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：クライアントが要求されたときに証明書の送信に失敗した場合（SSL サーバーで SSL_VERIFY_PEER を使用している場合）、サーバー側で検証が失敗します。SSL_VERIFY_FAIL_EXCEPT_PSK クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じですが、PSK 接続の場合を除きます。PSK 接続が行われている場合、接続はピア証明書なしで続行されます。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#) で作成された SSL コンテキストへのポインタ。
- **mode** ピアの証明書の検証モードを示すフラグ。
- **verify_callback** 検証が失敗したときに呼び出されるコールバック。コールバックが不要な場合は、verify_callback に NULL ポインタを使用できます。

See: [wolfSSL_set_verify](#)

Return: none 戻り値はありません。

Example

```

WOLFSSL_CTX*   ctx    = 0;
...
wolfSSL_CTX_set_verify(ctx, (WOLFSSL_VERIFY_PEER |
                             WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT), NULL);

```

A.6.2.29 function wolfSSL_set_verify

```
void wolfSSL_set_verify(
    WOLFSSL * ssl,
    int mode,
    VerifyCallback verify_callback
)
```

この関数はリモートピアの検証方法を設定し、また検証コールバックを SSL セッションに登録できるようにします。検証コールバックは検証失敗が発生した場合にのみ呼び出されます。検証コールバックが不要な場合は、verify_callback に NULL ポインタを使用できます。ピア証明書の検証モードは論理 OR 演算されたフラグのリストです。可能なフラグ値は以下の通りです。SSL_VERIFY_NONE クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常通り続行されます。サーバーモード：サーバーはクライアントに証明書要求を送信しません。したがって、クライアント検証は有効になりません。SSL_VERIFY_PEER クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトで有効になっているため、このオプションを使用しても効果はありません。サーバーモード：サーバーはクライアントに証明書要求を送信し、受信したクライアント証明書を検証します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：クライアントが要求されたときに証明書の送信に失敗した場合（SSL サーバーで SSL_VERIFY_PEER を使用している場合）、サーバー側で検証が失敗します。SSL_VERIFY_FAIL_EXCEPT_PSK クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じですが、PSK 接続の場合を除きます。PSK 接続が行われている場合、接続はピア証明書なしで続行されます。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **mode** ピアの証明書の検証モードを示すフラグ。
- **verify_callback** 検証が失敗したときに呼び出されるコールバック。コールバックが不要な場合は、verify_callback に NULL ポインタを使用できます。

See: wolfSSL_CTX_set_verify

Return: none 返り値はありません。

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_verify(ssl, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, 0);
```

A.6.2.30 function wolfSSL_CTX_set_session_cache_mode

```
long wolfSSL_CTX_set_session_cache_mode(
    WOLFSSL_CTX * ctx,
    long mode
)
```

この関数は SSL セッションキャッシングを有効または無効にします。動作は mode に使用される値に依存します。mode に使用できる値は以下の通りです。SSL_SESS_CACHE_OFF - セッションキャッシングを無効にします。セッションキャッシングはデフォルトで有効になっています。SSL_SESS_CACHE_NO_AUTO_CLEAR - セッションキャッシュの自動フラッシュを無効にします。自動フラッシュはデフォルトで有効になっています。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。
- **mode** セッションキャッシュの動作を変更するために使用される修飾子。

See:

- `wolfSSL_flush_sessions`
- `wolfSSL_get1_session`
- `wolfSSL_set_session`
- `wolfSSL_get_sessionID`
- `wolfSSL_CTX_set_timeout`

Return: SSL_SUCCESS 成功時に返されます。

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_session_cache_mode(ctx, SSL_SESS_CACHE_OFF);
if (ret != SSL_SUCCESS) {
    // SSL セッションキャッシングをオフにすることができませんでした。
}
```

A.6.2.31 function `wolfSSL_CTX_memrestore_cert_cache`

```
int wolfSSL_CTX_memrestore_cert_cache(
    WOLFSSL_CTX * ctx,
    const void * mem,
    int sz
)
```

この関数は、証明書キャッシュをメモリから復元します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **mem** 証明書キャッシュに復元される値を持つ void 型ポインタ。
- **sz** mem パラメータのサイズを表す int 型。

See: CM_MemRestoreCertCache

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなく実行された場合に返されます。
- BAD_FUNC_ARG ctx または mem パラメータが NULL の場合、または sz パラメータが 0 以下の場合に返されます。
- BUFFER_E 証明書キャッシュのメモリバッファが小さすぎる場合に返されます。
- CACHE_MATCH_ERROR 証明書キャッシュヘッダの不一致があった場合に返されます。
- BAD_MUTEX_E ロック mutex が失敗した場合に返されます。

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)){
    // 成功ケース
}
```

A.6.2.32 function `wolfSSL_CTX_set_cipher_list`

```
int wolfSSL_CTX_set_cipher_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

この関数は、指定された WOLFSSL_CTX に対する暗号スイートリストを設定します。設定したリストは、本コンテキストから新規に作成されるすべての WOLFSSL セッションのデフォルトリストとなり、文字列内の順序がそのまま優先順位（高い順から低い順）として用いられます。本関数を呼び出すたびに前回のリストは置き換えられます。リストは null 終端のテキスト文字列で、暗号スイート名および／または OpenSSL 互換のキーワードをコロンで区切って指定します（例: “TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-SHA256”）。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。
- **list** 指定された SSL コンテキストで使用する暗号スイートおよび／またはキーワードの null 終端テキスト文字列（コロン区切り）。

See:

- wolfSSL_set_cipher_list
- wolfSSL_get_ciphers
- wolfSSL_get_cipher_list
- wolfSSL_CTX_new

Return:

- WOLFSSL_SUCCESS 関数が正常に完了した場合に返されます。
- WOLFSSL_FAILURE 失敗した場合に返されます。

リスト中の各トークンには次のいずれかを指定できます。

1. 個別の暗号スイート名。wolfSSL は独自の短縮名と IANA 名（WOLFSSL_NO_ERROR_STRINGS が定義されていない場合）の両方を受理します。たとえば “ECDHE-RSA-AES128-GCM-SHA256” と “TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256” は等価です。受理される名前の完全なリストは src/internal.c の cipher_names[] 配列にあります。実行時に現在のビルドに含まれるスイートを列挙するには wolfSSL_get_ciphers() を使用してください。
2. OpenSSL 互換のキーワード（下表参照）。スイート群を一括で選択したり、特定クラスを on/off するために使用します。一部のキーワードには OPENSSL_EXTRA または OPENSSL_ALL の定義が必要です。
3. 否定キーワード “!keyword”（OPENSSL_EXTRA または OPENSSL_ALL が必要）。例: “HIGH:!aNULL”。

“+” 演算子（例: “ECDHE+AESGCM”）は、先頭の公開鍵ファミリトークン (“ECDHE”, “RSA”, “DHE”) の抽出にのみ使用されます。“+” の後の部分は wolfSSL では無視されます。特定のスイートを指定したい場合は完全なスイート名を直接指定してください。

OpenSSL 互換のグループキーワード:

キーワード	効果	必要なビルドオプション
DEFAULT / ALL	ビルドされている全スイートを含める。“ALL” は匿名 (aNULL) も有効化	OPENSSL_EXTRA / OPENSSL_ALL (他トークンを伴わない単独指定も可)
HIGH	静的・匿名・NULL を除く全スイート	OPENSSL_EXTRA / OPENSSL_ALL (単独指定も可)
LOW / MEDIUM	受理されるが鍵長による制限はかからず “RSA” 相当として扱われる	OPENSSL_EXTRA / OPENSSL_ALL
aNULL	匿名（認証なし）スイート	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体は HAVE_ANON が必要)
eNULL / NULL	NULL 暗号化スイート	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体は HAVE_NULL_CIPHER が必要)

キーワード	効果	必要なビルドオプション
kDH	静的 DH 鍵交換	OPENSSL_EXTRA / OPENSSL_ALL
DHE / EDH	エフェメラル DH 鍵交換	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体 は!NO_DH が必要)
ECDHE / ECDH	エフェメラル ECDH 鍵交換	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体 は HAVE_ECC が必要)
kRSA / RSA	静的 RSA 鍵交換	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体 は!NO_RSA が必要)
PSK	PSK スイート	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体 は!NO_PSK が必要)
DSS	黙って無視される (wolfSSL は DSA スイート非対応)	OPENSSL_EXTRA / OPENSSL_ALL
EXP / EXPORT	黙って無視される (export 等級 は非対応)	OPENSSL_EXTRA / OPENSSL_ALL
AES128 / SHA1 / RC4	否定指定 ("!AES128" 等) でそ のクラスを無効化	WOLFSSL_SYS_CRYPTOPOLICY
@SECLEVEL=n	OpenSSL 互換のセキュリティ レベル設定	WOLFSSL_SYS_CRYPTOPOLICY

代表的な TLS 1.3 暗号スイート名 (各々個別の BUILD_* マクロでガードされる。多くは -enable-tls13 で自動有効化) :

名前 (wolfSSL)	IANA 名	必要なビルドオプション
TLS13-AES128-GCM-SHA256	TLS_AES_128_GCM_SHA256	BUILD_TLS_AES_128_GCM_SHA256 (TLS 1.3 既定)
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_AECCCM)
TLS13-AES128-CCM-8-SHA256	TLS_AES_128_CCM_8_SHA256	BUILD_TLS_AES_128_CCM_8_SHA256 (HAVE_AECCCM)
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

代表的な TLS 1.2 暗号スイート名のファミリ (各スイートは IANA 名から派生する個別の BUILD_* マクロでガードされる) :

スイート名のファミリ／例	一般的な必要オプション
ECDHE-ECDSA-AES128-GCM-SHA256, ECDHE-ECDSA-AES256-GCM-SHA384, ECDHE-ECDSA-CHACHA20-POLY1305	HAVE_ECC, HAVE_AESGCM (または HAVE_CHACHA + HAVE_POLY1305)
ECDHE-RSA-AES128-GCM-SHA256, ECDHE-RSA-AES256-GCM-SHA384, ECDHE-RSA-CHACHA20-POLY1305	HAVE_ECC, !NO_RSA, HAVE_AESGCM (または HAVE_CHACHA + HAVE_POLY1305)
DHE-RSA-AES128-GCM-SHA256, DHE-RSA-AES256-GCM-SHA384, DHE-RSA-CHACHA20-POLY1305	!NO_DH, !NO_RSA, HAVE_AESGCM (または HAVE_CHACHA + HAVE_POLY1305)
AES128-SHA, AES256-SHA, AES128-SHA256, AES256-SHA256, AES128-GCM-SHA256, AES256-GCM-SHA384 (静的 RSA)	!NO_RSA, !NO_AES_CBC および／または HAVE_AESGCM
DES-CBC3-SHA, RC4-SHA, RC4-MD5 NULL-SHA, NULL-SHA256, NULL-MD5	レガシー: !NO_DES3 / !NO_RC4, !NO_OLD_TLS HAVE_NULL_CIPHER
PSK-AES128-CBC-SHA256, PSK-AES256-GCM-SHA384, ECDHE-PSK-AES128-CBC-SHA256, DHE-PSK-AES256-GCM-SHA384	!NO_PSK (ECDHE-PSK は HAVE_ECC、DHE-PSK は!NO_DH も必要)
ADH-AES128-SHA, ADH-AES256-SHA ECDHE-ECDSA-SM4-GCM-SM3, ECDHE-ECDSA-SM4-CCM-SM3	HAVE_ANON WOLFSSL_SM2, WOLFSSL_SM3, WOLFSSL_SM4_GCM/WOLFSSL_SM4_CCM

注意事項:

- TLS 1.3 と TLS 1.2 のスイート名は同一リスト内で混在指定できます。wolfSSL は内部でバージョンごとに振り分けます。
- DTLS では、リスト中の RC4 ベースのストリーム暗号は黙って除外されます。
- リスト全体が“DEFAULT”、“ALL”、“HIGH”、または空文字列の場合、wolfSSL は組み込みのデフォルトスイートリストを設定して成功を返し、それ以上のトークン解析は行いません。

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) {
    // 暗号スイートリストの設定に失敗しました。
}
```

A.6.2.33 function wolfSSL_set_cipher_list

```
int wolfSSL_set_cipher_list(
    WOLFSSL * ssl,
    const char * list
)
```

この関数は、指定された WOLFSSL セッション (SSL セッション) に対する暗号スイートリストを設定します。リストの形式、認識される暗号スイート名およびキーワードは `wolfSSL_CTX_set_cipher_list()` の説明を参照してください。本関数を呼び出すたびに、当該セッションのリストは置き換えられます。

Parameters:

- **ssl** `wolfSSL_new()` で作成された SSL セッションへのポインタ。

- **list** 指定された SSL セッションで使用する暗号スイートおよび／またはキーワードの null 終端テキスト文字列（コロン区切り）。

See:

- `wolfSSL_CTX_set_cipher_list`
- `wolfSSL_get_ciphers`
- `wolfSSL_get_cipher_list`
- `wolfSSL_new`

Return:

- WOLFSSL_SUCCESS 関数が正常に完了すると返されます。
- WOLFSSL_FAILURE 失敗時に返されます。

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) {
    // 暗号スイートリストの設定に失敗
}
```

A.6.2.34 function `wolfSSL_dtls13_set_send_more_acks`

```
void wolfSSL_dtls13_set_send_more_acks(
    WOLFSSL * ssl,
    int value
)
```

この関数は、中断を検出したときにライブラリが ACK を他のピアに即座に送信すべきかどうかを設定します。ACK を即座に送信すると最小のレイテンシが保証されますが、必要以上に帯域幅を消費する可能性があります。アプリケーションが自分でタイマーを管理し、このオプションが 0 に設定されている場合、アプリケーションコードは `wolfSSL_dtls13_use_quick_timeout()` を使用して、これらの遅延 ACK を送信するためにより短いタイムアウトを設定すべきかどうかを判定できます。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **value** オプションを設定する場合は 1、オプションを無効にする場合は 0。

See:

- `wolfSSL_dtls`
- `wolfSSL_dtls_get_peer`
- `wolfSSL_dtls_got_timeout`
- `wolfSSL_dtls_set_peer`
- `wolfSSL_dtls13_use_quick_timeout`

A.6.2.35 function `wolfSSL_dtls_set_timeout_init`

```
int wolfSSL_dtls_set_timeout_init(
    WOLFSSL * ssl,
    int timeout
)
```


この関数は dtls タイムアウトを設定します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **timeout** WOLFSSL 構造体の dtls_timeout_init メンバに設定される int 型の値。

See:

- wolfSSL_dtls_set_timeout_max
- wolfSSL_dtls_got_timeout

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。SSL の dtls_timeout_init および dtls_timeout メンバが設定されています。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL であるか、またはタイムアウトが 0 より大きくない場合に返されます。また、timeout 引数が許容される最大値を超える場合にも返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int timeout = TIMEOUT;
...
if(wolfSSL_dtls_set_timeout_init(ssl, timeout)){
    // dtls タイムアウトが設定された
} else {
    // DTLS タイムアウトの設定に失敗
}
```

A.6.2.36 function wolfSSL_get1_session

```
WOLFSSL_SESSION * wolfSSL_get1_session(
    WOLFSSL * ssl
)
```

この関数は WOLFSSL 構造体から WOLFSSL_SESSION を参照型として返します。これには wolfSSL_SESSION_free を呼び出してセッション参照を解放する必要があります。指される WOLFSSL_SESSION には、セッション再開を実行し、新しいハンドシェイクなしで接続を再確立するために必要なすべての情報が含まれています。セッション再開のために、セッションオブジェクトで wolfSSL_shutdown() を呼び出す前に、アプリケーションは wolfSSL_get1_session() の呼び出しでオブジェクトからセッション ID を保存する必要があります。これはセッションへのポインタを返します。後で、アプリケーションは新しい WOLFSSL オブジェクトを作成し、wolfSSL_set_session() で保存されたセッションを割り当てる必要があります。この時点で、アプリケーションは wolfSSL_connect() を呼び出すことができ、wolfSSL はセッションの再開を試みます。wolfSSL サーバコードはデフォルトでセッション再開を許可します。wolfSSL_get1_session() によって返されたオブジェクトは、アプリケーションが使用を終えた後、wolfSSL_SESSION_free() を呼び出すことによって解放する必要があります。

Parameters:

- **ssl** セッションを取得する WOLFSSL 構造体。

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_SESSION_free

Return:

- WOLFSSL_SESSION 成功時にセッションポインタを返します。

- NULL ssl が NULL、SSL セッションキャッシュが無効、wolfSSL がセッション ID を利用できない、または mutex 関数が失敗した場合に返されます。

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* ses;    // ハンドシェイクを試行/完了
wolfSSL_connect(ssl);
ses = wolfSSL_get1_session(ssl);
// ses 情報を確認
// 切断/新しい SSL インスタンスをセットアップ
wolfSSL_set_session(ssl, ses);
// ハンドシェイクを試行/再開
wolfSSL_SESSION_free(ses);
```

A.6.2.37 function wolfSSLv23_client_method

```
WOLFSSL_METHOD * wolfSSLv23_client_method(
    void
)
```

wolfSSLv23_client_method() 関数は、アプリケーションがクライアントであり、SSL 3.0 から TLS 1.3 までの間でサーバがサポートする最も高いプロトコルバージョンをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい WOLFSSL_METHOD 構造体のためのメモリを割り当て、初期化します。wolfSSL のクライアントとサーバの両方は、堅牢なバージョンダウングレード機能を持っています。どちらか一方で特定のプロトコルバージョンメソッドが使用された場合、そのバージョンのみがネゴシエートされるか、エラーが返されます。例えば、TLSv1 を使用するクライアントが SSLv3 のみのサーバに接続しようとする失敗し、同様に TLSv1.1 への接続も失敗します。この問題を解決するために、wolfSSLv23_client_method() 関数を使用するクライアントは、サーバがサポートする最も高いプロトコルバージョンを使用し、必要に応じて SSLv3 までダウングレードします。この場合、クライアントは SSLv3 から TLSv1.3 までを実行しているサーバに接続できます。

Parameters:

- none パラメータなし。

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 成功時、WOLFSSL_METHOD へのポインタ。
- Failure XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
method = wolfSSLv23_client_method();
if (method == NULL) {
    // メソッドを取得できませんでした
```

```

}

ctx = wolfSSL_CTX_new(method);
...

```

A.6.2.38 function wolfSSL_ASN1_INTEGER_to_BN

```

WOLFSSL_BIGNUM * wolfSSL_ASN1_INTEGER_to_BN(
    const WOLFSSL_ASN1_INTEGER * ai,
    WOLFSSL_BIGNUM * bn
)

```

この関数は、WOLFSSL_ASN1_INTEGER 値を WOLFSSL_BIGNUM 構造体にコピーするために使用されます。

Parameters:

- **ai** コピー元の WOLFSSL_ASN1_INTEGER 構造体。
- **bn** 既存の WOLFSSL_BIGNUM 構造体にコピーしたい場合は、そのポインタを渡します。オプションとして、これを NULL にすることで新しい WOLFSSL_BIGNUM 構造体を作成されます。

See: none

Return:

- pointer WOLFSSL_ASN1_INTEGER 値のコピーに成功すると、WOLFSSL_BIGNUM ポインタが返されます。
- Null 失敗時には Null が返されます。

Example

```

WOLFSSL_ASN1_INTEGER* ai;
WOLFSSL_BIGNUM* bn;
// ai を作成
bn = wolfSSL_ASN1_INTEGER_to_BN(ai, NULL);

// または既に bn を作成済みで構造体を再利用したい場合
// wolfSSL_ASN1_INTEGER_to_BN(ai, bn);
// bn または戻り値が NULL でないことを確認

```

A.6.2.39 function wolfSSL_CTX_add_extra_chain_cert

```

long wolfSSL_CTX_add_extra_chain_cert(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509 * x509
)

```

この関数は、WOLFSSL_CTX 構造体内で構築中の内部チェーンに証明書を追加します。

Parameters:

- **ctx** 証明書を追加する WOLFSSL_CTX 構造体。
- **x509** チェーンに追加する証明書。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)

Return:

- SSL_SUCCESS 証明書の追加に成功した後に返されます。

- SSL_FAILURE 証明書のチェーンへの追加に失敗した場合に返されます。

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL_X509* x509;
int ret;
// ctx を作成
ret = wolfSSL_CTX_add_extra_chain_cert(ctx, x509);
// ret 値を確認
```

A.6.2.40 function wolfSSL_CTX_get_read_ahead

```
int wolfSSL_CTX_get_read_ahead(
    WOLFSSL_CTX * ctx
)
```

この関数は、WOLFSSL_CTX 構造体からリードアヘッドフラグを取得して返します。

Parameters:

- **ctx** リードアヘッドフラグを取得する WOLFSSL_CTX 構造体。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)
- [wolfSSL_CTX_set_read_ahead](#)

Return:

- flag 成功時にはリードアヘッドフラグが返されます。
- SSL_FAILURE ctx が NULL の場合、SSL_FAILURE が返されます。

Example

```
WOLFSSL_CTX* ctx;
int flag;
// ctx をセットアップ
flag = wolfSSL_CTX_get_read_ahead(ctx);
// flag を確認
```

A.6.2.41 function wolfSSL_CTX_set_read_ahead

```
int wolfSSL_CTX_set_read_ahead(
    WOLFSSL_CTX * ctx,
    int v
)
```

この関数は、WOLFSSL_CTX 構造体内のリードアヘッドフラグを設定します。

Parameters:

- **ctx** リードアヘッドフラグを設定する WOLFSSL_CTX 構造体。
- **v** リードアヘッドフラグ。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)
- [wolfSSL_CTX_get_read_ahead](#)

Return:

- SSL_SUCCESS ctx のリード Ahead フラグが設定された場合。
- SSL_FAILURE ctx が NULL の場合、SSL_FAILURE が返されます。

Example

```
WOLFSSL_CTX* ctx;
int flag;
int ret;
// ctx をセットアップ
ret = wolfSSL_CTX_set_read_ahead(ctx, flag);
// 戻り値を確認
```

A.6.2.42 function wolfSSL_CTX_set_tlsext_status_arg

```
long wolfSSL_CTX_set_tlsext_status_arg(
    WOLFSSL_CTX * ctx,
    void * arg
)
```

この関数は、OCSP で使用するオプション引数を設定します。

Parameters:

- **ctx** ユーザー引数を設定する WOLFSSL_CTX 構造体。
- **arg** ユーザー引数。
- **ctx** WOLFSSL_CTX オブジェクト。
- **arg** コールバックに渡すユーザー引数。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)
- [wolfSSL_CTX_set_tlsext_status_cb](#)

Return:

- SSL_FAILURE ctx またはその証明書マネージャが NULL の場合。
- SSL_SUCCESS 正常に設定された場合。
- SSL_SUCCESS 成功時、それ以外の場合は SSL_FAILURE。

OCSP ステータスコールバックに渡される引数を設定します。

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// ctx をセットアップ
ret = wolfSSL_CTX_set_tlsext_status_arg(ctx, data);

//ret 値を確認
```

A.6.2.43 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
)
```

この関数は、PRF コールバックに渡されるオプション引数を設定します。

Parameters:

- **ctx** ユーザ引数を設定する WOLFSSL_CTX 構造体。
- **arg** ユーザ引数。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)

Return:

- SSL_FAILURE ctx が NULL の場合。
- SSL_SUCCESS 正常に設定された場合。

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// ctx をセットアップ
ret = wolfSSL_CTX_set_tlsext_opaques_prf_input_callback_arg(ctx, data);
//ret 値を確認
```

A.6.2.44 function wolfSSL_CTX_set_options

```
long wolfSSL_CTX_set_options(
    WOLFSSL_CTX * ctx,
    long opt
)
```

この関数は、指定された WOLFSSL_CTX のオプションマスクに opt のビットを OR で追加します。設定したオプションマスクは、本コンテキストから後に作成されるすべての WOLFSSL セッションに継承されます。ビットは累積されるため、特定のオプションを解除するには wolfSSL_CTX_clear_options() を使用してください。OpenSSL 形式の "SSL_OP_" マクロは対応する "WOLFSSL_OP_" 値の別名であり、どちらの表記でも使用できます。

Parameters:

- **ctx** オプションマスクを設定する WOLFSSL_CTX 構造体。
- **opt** 現在のマスクに OR で追加する SSL_OP_* / WOLFSSL_OP_* フラグのビットマスク。

See:

- [wolfSSL_CTX_clear_options](#)
- [wolfSSL_CTX_get_options](#)
- [wolfSSL_set_options](#)
- [wolfSSL_get_options](#)

Return:

- BAD_FUNC_ARG ctx が NULL の場合。
- 成功時、ctx に格納された更新後のオプションマスク値。

実効を持つオプション:

マクロ	効果
SSL_OP_NO_SSLv2	SSLv2 を無効化 (wolfSSL は SSLv2 を一切サポートしない。OpenSSL 互換のため受け付けるのみ)
SSL_OP_NO_SSLv3	SSLv3 を無効化
SSL_OP_NO_TLSv1	TLS 1.0 を無効化
SSL_OP_NO_TLSv1_1	TLS 1.1 を無効化

マクロ	効果
SSL_OP_NO_TLSv1_2	TLS 1.2 を無効化
SSL_OP_NO_TLSv1_3	TLS 1.3 を無効化 (WOLFSSL_TLS13 が必要)
SSL_OP_NO_COMPRESSION	レコード層圧縮を無効化 (HAVE_LIBZ 未定義時は何もしない)
SSL_OP_NO_TICKET	RFC 5077 セッションチケットを無効化 (TLS 1.2 のみ。TLS 1.3 では本フラグは無視される)。 HAVE_SESSION_TICKET と OPENSSL_EXTRA / HAVE_WEBSERVER / WOLFSSL_WPAS_SMALL のいずれかが必要
SSL_OP_NO_RENEGOTIATION	相手主導の再ネゴシエーションを拒否
SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION	再ネゴシエーション時のセッション再利用を無効化
SSL_OP_COOKIE_EXCHANGE	HelloVerifyRequest クッキー交換を有効化 (DTLS では既定で有効)
SSL_OP_NO_QUERY_MTU	DTLS: 経路 MTU を問い合わせない
SSL_OP_CIPHER_SERVER_PREFERENCE	サーバ側の暗号スイート優先順位をクライアントの順序より優先
SSL_OP_SINGLE_DH_USE	ハンドシェイクごとに新しい DH 鍵を生成
SSL_OP_SINGLE_ECDH_USE	ハンドシェイクごとに新しい ECDH 鍵を生成
SSL_OP_EPHEMERAL_RSA	エフェメラル RSA を使用 (レガシー。OpenSSL 互換のため受け付けるのみ)
SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS	レコード前の空フラグメント挿入を抑制 (CBC BEAST 回避策)
SSL_OP_PKCS1_CHECK_1 / _2	OpenSSL 互換のため受け付けるのみ
SSL_OP_LEGACY_SERVER_CONNECT	安全でない初回接続を常に許可。定義値は 0 のため実質無効化 (openssl/ssl.h 互換ヘッダが必要)

便宜マクロ・バグ回避フラグ (すべて SSL_OP_ALL の構成要素。OpenSSL 互換のため受け付けるが wolfSSL では実質 no-op) :

- SSL_OP_ALL (下記バグ回避フラグのビット OR)
- 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

複合マクロ:

- 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 には ctx に蓄積されたオプション値が反映される

```

A.6.2.45 function wolfSSL_set_options

```

long wolfSSL_set_options(
    WOLFSSL * s,
    long op
)

```

この関数は、指定された WOLFSSL セッションのオプションマスクに op のビットを OR で追加します。認識される "SSL_OP_" / "WOLFSSL_OP_" フラグの集合は [wolfSSL_CTX_set_options\(\)](#) の説明を参照してください。親 WOLFSSL_CTX から継承したフラグはそのまま保持されます。本関数で SSL_OP_NO_TLSv1_3 を設定した場合、当該セッションのネゴシエーション可能なバージョンは TLS 1.2 まで引き下げられます。

Parameters:

- **s** オプションマスクを設定する WOLFSSL セッション。
- **op** 現在のマスクに OR で追加する SSL_OP_* / WOLFSSL_OP_* フラグのビットマスク。

See:

- [wolfSSL_CTX_set_options](#)
- [wolfSSL_clear_options](#)
- [wolfSSL_get_options](#)
- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return: 成功時、s に格納された更新後のオプションマスク値。s が NULL の場合は 0 を返します。

Example

```

WOLFSSL* ssl;
long mask;
mask = wolfSSL_set_options(ssl, SSL_OP_NO_TLSv1_3);
// mask には ssl に蓄積されたオプション値が反映される

```

A.6.2.46 function wolfSSL_get_options

```

long wolfSSL_get_options(
    const WOLFSSL * s
)

```

この関数は、現在のオプションマスクを返します。

Parameters:

- **s** オプションマスクを取得する WOLFSSL 構造体。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)
- [wolfSSL_set_options](#)

Return: val ssl に格納されているマスク値を返します。

Example

```

WOLFSSL* ssl;
unsigned long mask;

```

```
mask = wolfSSL_get_options(ssl);  
// mask を確認
```

A.6.2.47 function wolfSSL_set_tlsext_debug_arg

```
long wolfSSL_set_tlsext_debug_arg(  
    WOLFSSL * s,  
    void * arg  
)
```

これは、渡されるデバッグ引数を設定するために使用されます。

Parameters:

- **s** 引数を設定する WOLFSSL 構造体。
- **arg** 使用する引数。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- SSL_SUCCESS 引数の設定に成功した場合。
- SSL_FAILURE NULL の ssl が渡された場合。

Example

```
WOLFSSL* ssl;  
void* args;  
int ret;  
// ssl オブジェクトを作成  
ret = wolfSSL_set_tlsext_debug_arg(ssl, args);  
// ret 値を確認
```

A.6.2.48 function wolfSSL_get_verify_result

```
long wolfSSL_get_verify_result(  
    const WOLFSSL * ssl  
)
```

これは、ピアの証明書の検証を試みた後の結果を取得するために使用されます。

Parameters:

- **ssl** 検証結果を取得する WOLFSSL 構造体。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- X509_V_OK 検証が成功した場合。
- SSL_FAILURE NULL の ssl が渡された場合。

Example

```
WOLFSSL* ssl;  
long ret;  
// ハンドシェイクを試行/完了
```



```
ret = wolfSSL_get_verify_result(ssl);
// ret 値を確認
```

A.6.2.49 function wolfSSL_CTX_allow_anon_cipher

```
int wolfSSL_CTX_allow_anon_cipher(
    WOLFSSL_CTX * ctx
)
```

この関数は、コンパイル時に HAVE_ANON が定義されている場合、CTX 構造体の havAnon メンバを有効にします。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See: none

Return:

- SSL_SUCCESS 関数が正常に実行され、CTX の haveAnon メンバが 1 に設定された場合に返されます。
- SSL_FAILURE CTX 構造体が NULL の場合に返されます。

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){
        // 失敗ケース
    }
}
#endif
```

A.6.2.50 function wolfSSLv23_server_method

```
WOLFSSL_METHOD * wolfSSLv23_server_method(
    void
)
```

wolfSSLv23_server_method() 関数は、アプリケーションがサーバであり、SSL 3.0 から TLS 1.3 までのプロトコルバージョンで接続するクライアントをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい WOLFSSL_METHOD 構造体のためのメモリを割り当て、初期化します。

Parameters:

- **none** パラメータなし。

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 成功時、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- Failure XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv23_server_method();
if (method == NULL) {
    // メソッドを取得できませんでした
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.51 function wolfSSL_state

```
int wolfSSL_state(
    WOLFSSL * ssl
)
```

これは、WOLFSSL 構造体の内部エラー状態を取得するために使用されます。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- wolfssl_error ssl エラー状態を返します。通常は負の値です。
- BAD_FUNC_ARG ssl が NULL の場合。
- ssl 状態を取得する WOLFSSL 構造体。

Example

```
WOLFSSL* ssl;
int ret;
// ssl オブジェクトを作成
ret = wolfSSL_state(ssl);
// ret 値を確認
```

A.6.2.52 function wolfSSL_check_domain_name

```
int wolfSSL_check_domain_name(
    WOLFSSL * ssl,
    const char * dn
)
```

wolfSSL はデフォルトでピア証明書の有効な日付範囲と検証済み署名をチェックします。wolfSSL_connect() または wolfSSL_accept() の前にこの関数を呼び出すと、実行するチェックのリストにドメイン名チェックが追加されます。dn は、受信したピア証明書に対してチェックするドメイン名を保持します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **dn** 受信したピア証明書に対してチェックするドメイン名。

See: none

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE メモリエラーが発生した場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
char* domain = (char*) "www.yassl.com";
...

ret = wolfSSL_check_domain_name(ssl, domain);
if (ret != SSL_SUCCESS) {
    // ドメイン名チェックの有効化に失敗しました
}
```

A.6.2.53 function wolfSSL_check_ip_address

```
int wolfSSL_check_ip_address(
    WOLFSSL * ssl,
    const char * ipaddr
)
```

`wolfSSL_connect()` の前にこの関数を呼ぶと、ピア証明書の SAN iAddress エントリに対する IP アドレス検証を追加します。

Parameters:

- **ssl** `wolfSSL_new()` で作成された WOLFSSL 構造体へのポインタ。
- **ipaddr** 検証する NULL 終端 ASCII IP アドレス文字列。

See: `wolfSSL_check_domain_name`

Return:

- SSL_SUCCESS 成功。
- SSL_FAILURE パラメータ不正またはメモリ確保失敗。

A.6.2.54 function wolfSSL_set_compression

```
int wolfSSL_set_compression(
    WOLFSSL * ssl
)
```

SSL 接続で圧縮を使用する機能をオンにします。両側で圧縮をオンにする必要があります。そうでない場合、圧縮は使用されません。zlib ライブラリが実際のデータ圧縮を実行します。ライブラリにコンパイルするには、configure システムで `-with-libz` を使用し、HAVE_LIBZ を定義してください。送信前にデータを圧縮すると送受信されるメッセージの実際のサイズは減少しますが、圧縮によって節約されるデータの量は、最も遅いネットワークを除いて、生データで送信するよりも解析に時間がかかることに注意してください。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL セッションへのポインタ。

See: none

Return:

- SSL_SUCCESS 成功時。
- NOT_COMPILED_IN 圧縮サポートがライブラリにビルドされていない場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_compression(ssl);
if (ret == SSL_SUCCESS) {
    // SSL セッションの圧縮を正常に有効化しました
}
```

A.6.2.55 function wolfSSL_set_timeout

```
int wolfSSL_set_timeout(
    WOLFSSL * ssl,
    unsigned int to
)
```

この関数は SSL セッションのタイムアウト値を秒単位で設定します。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL オブジェクトへのポインタ。
- **to** SSL セッションタイムアウトの設定に使用される秒単位の値。

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- SSL_SUCCESS セッションの設定に成功した場合に返されます。
- BAD_FUNC_ARG ssl が NULL の場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_timeout(ssl, 500);
if (ret != SSL_SUCCESS) {
    // セッションタイムアウト値の設定に失敗しました
}
...
```

A.6.2.56 function wolfSSL_CTX_set_timeout

```
int wolfSSL_CTX_set_timeout(
    WOLFSSL_CTX * ctx,
    unsigned int to
)
```

この関数は、指定された SSL コンテキストの SSL セッションのタイムアウト値を秒単位で設定します。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。

- **to** セッションタイムアウト値 (秒単位)。

See:

- `wolfSSL_flush_sessions`
- `wolfSSL_get1_session`
- `wolfSSL_set_session`
- `wolfSSL_get_sessionID`
- `wolfSSL_CTX_set_session_cache_mode`

Return:

- WOLFSSL_ERROR_CODE_OPENSSL が定義されている場合、成功時に以前のタイムアウト値が返されます。定義されていない場合、SSL_SUCCESS が返されます。
- BAD_FUNC_ARG 入力コンテキスト (ctx) が null の場合に返されます。

Example

```
WOLFSSL_CTX*   ctx   = 0;
...
ret = wolfSSL_CTX_set_timeout(ctx, 500);
if (ret != SSL_SUCCESS) {
    // セッションタイムアウト値の設定に失敗しました
}
```

A.6.2.57 function wolfSSL_CTX_UnloadCAs

```
int wolfSSL_CTX_UnloadCAs(
    WOLFSSL_CTX * ctx
)
```

この関数は、CA 署名者リストをアンロードし、署名者テーブル全体を解放します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- `wolfSSL_CertManagerUnloadCAs`
- `LockMutex`
- `UnlockMutex`

Return:

- SSL_SUCCESS 関数の実行が成功した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合、またはサブルーチンで許可されない引数値が渡された場合に返されます。
- BAD_MUTEX_E mutex エラーが発生した場合に返されます。LockMutex() が 0 を返しませんでした。

Example

```
WOLFSSL_METHOD method = wolfTLsv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
...
if(wolfSSL_CTX_UnloadCAs(ctx) != SSL_SUCCESS){
    // 関数は CA をアンロードしませんでした
}
```

A.6.2.58 function wolfSSL_CTX_UnloadIntermediateCerts

```
int wolfSSL_CTX_UnloadIntermediateCerts(
    WOLFSSL_CTX * ctx
)
```

この関数は、CA 署名者リストに追加された中間証明書をアンロードし、解放します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- `wolfSSL_CTX_UnloadCAs`
- `wolfSSL_CertManagerUnloadIntermediateCerts`

Return:

- SSL_SUCCESS 関数の実行が成功した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合、またはサブルーチンで許可されない引数値が渡された場合に返されます。
- BAD_STATE_E WOLFSSL_CTX の参照カウントが 1 より大きい場合に返されます。
- BAD_MUTEX_E mutex エラーが発生した場合に返されます。LockMutex() が 0 を返しませんでした。

Example

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
...
if(wolfSSL_CTX_UnloadIntermediateCerts(ctx) != NULL){
    // 関数は CA をアンロードしませんでした
}
```

A.6.2.59 function wolfSSL_CTX_Unload_trust_peers

```
int wolfSSL_CTX_Unload_trust_peers(
    WOLFSSL_CTX * ctx
)
```

この関数は以前にロードされたすべての信頼されたピア証明書をアンロードするために使用されます。この機能は、マクロ WOLFSSL_TRUST_PEER_CERT を定義することで有効になります。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された SSL コンテキストへのポインタ。

See:

- `wolfSSL_CTX_trust_peer_buffer`
- `wolfSSL_CTX_trust_peer_cert`

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG ctx が NULL の場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
```

```
ret = wolfSSL_CTX_Unload_trust_peers(ctx);
if (ret != SSL_SUCCESS) {
    // 信頼されたピア証明書のアンロードエラー
}
...
```

A.6.2.60 function wolfSSL_CTX_trust_peer_buffer

```
int wolfSSL_CTX_trust_peer_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、TLS/SSL ハンドシェイクを実行する際にピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されるピア証明書は、利用可能な場合は SKID と署名を使用して比較されます。これら 2 つが一致しない場合は、ロードされた CA が使用されます。ファイルではなくバッファからの入力である点を除いて、wolfSSL_CTX_trust_peer_cert と同じ機能です。この機能は、マクロ WOLFSSL_TRUST_PEER_CERT を定義することで有効になります。適切な使用方法については例を参照してください。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。
- **buffer** 証明書を含むバッファへのポインタ。
- **sz** 入力バッファの長さ。
- **type** ロードされる証明書のタイプ、すなわち SSL_FILETYPE_ASN1 または 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 成功時。
- SSL_FAILURE ctx が NULL、または file と type の両方が無効な場合に返されます。
- SSL_BAD_FILETYPE ファイルの形式が間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...

ret = wolfSSL_CTX_trust_peer_buffer(ctx, bufferPtr, bufferSz,
    SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
```

```

    // 信頼されたピア証明書のロードエラー
}
...

```

A.6.2.61 function wolfSSL_CTX_set_group_messages

```

int wolfSSL_CTX_set_group_messages(
    WOLFSSL_CTX * ctx
)

```

この関数は、可能な場合にハンドシェイクメッセージのグループ化をオンにします。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された SSL コンテキストへのポインタ。
- **ctx** WOLFSSL_CTX 構造体へのポインタ。

See:

- [wolfSSL_set_group_messages](#)
- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_clear_group_messages](#)
- [wolfSSL_set_group_messages](#)
- [wolfSSL_clear_group_messages](#)

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 入力コンテキストが null の場合に返されます。
- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG ctx が NULL の場合。

指定された WOLFSSL_CTX コンテキストに対してハンドシェイクメッセージグループ化を有効にします。

Example

```

WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_group_messages(ctx);
if (ret != SSL_SUCCESS) {
    // ハンドシェイクメッセージのグループ化の設定に失敗
}

```

この関数は、指定されたコンテキストから作成されたすべての SSL オブジェクトに対してハンドシェイクメッセージグループ化をオンにします。

Example

```

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
wolfSSL_CTX_set_group_messages(ctx);

```

A.6.2.62 function wolfSSL_set_group_messages

```

int wolfSSL_set_group_messages(
    WOLFSSL * ssl
)

```

この関数は、可能な場合にハンドシェイクメッセージのグループ化をオンにします。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)で作成された SSL セッションへのポインタ。

- **ssl** WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_CTX_set_group_messages`
- `wolfSSL_new`
- `wolfSSL_clear_group_messages`
- `wolfSSL_CTX_set_group_messages`
- `wolfSSL_CTX_clear_group_messages`

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 入力コンテキストが null の場合に返されます。
- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG ssl が NULL の場合。

指定された WOLFSSL オブジェクトに対してハンドシェイクメッセージグループ化を有効にします。

Example

```
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_group_messages(ssl);
if (ret != SSL_SUCCESS) {
    // ハンドシェイクメッセージのグループ化の設定に失敗
}
```

この関数は、指定された SSL オブジェクトに対してハンドシェイクメッセージグループ化をオンにします。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_set_group_messages(ssl);
```

A.6.2.63 function wolfSSL_CTX_SetMinVersion

```
int wolfSSL_CTX_SetMinVersion(
    WOLFSSL_CTX * ctx,
    int version
)
```

この関数は、許可される最小ダウングレードバージョンを設定します。ダウングレードを許可する接続 (wolfSSLv23_client_method または wolfSSLv23_server_method) を使用する場合にのみ適用されます。

Parameters:

- **ctx** `wolfSSL_CTX_new()` を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **version** 最小値として設定されるバージョンの整数表現: WOLFSSL_SSLV3 = 0、WOLFSSL_TLSV1 = 1、WOLFSSL_TLSV1_1 = 2、または WOLFSSL_TLSV1_2 = 3。

See: SetMinVersionHelper

Return:

- SSL_SUCCESS 関数がエラーなく戻り、最小バージョンが設定された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合、または最小バージョンがサポートされていない場合に返されます。

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
```

```
int version; // マクロ表現
...
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
    // 最小バージョンの設定に失敗
}
```

A.6.2.64 function wolfSSL_SetVersion

```
int wolfSSL_SetVersion(
    WOLFSSL * ssl,
    int version
)
```

この関数は、version で指定されたバージョンを使用して、指定された SSL セッション (WOLFSSL オブジェクト) の SSL/TLS プロトコルバージョンを設定します。これにより、SSL セッション (ssl) のプロトコル設定が上書きされます。これは元々 SSL コンテキスト (wolfSSL_CTX_new()) のメソッドタイプによって定義および設定されたものです。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **version** SSL/TLS プロトコルバージョン。指定可能な値には、WOLFSSL_SSLV3、WOLFSSL_TLSV1、WOLFSSL_TLSV1_1、WOLFSSL_TLSV1_2 があります。

See: [wolfSSL_CTX_new](#)

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG 入力された SSL オブジェクトが NULL の場合、または version に不正なプロトコルバージョンが指定された場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_SetVersion(ssl, WOLFSSL_TLSV1);
if (ret != SSL_SUCCESS) {
    // SSL セッションのプロトコルバージョンの設定に失敗しました。
}
```

A.6.2.65 function wolfSSL_UseALPN

```
int wolfSSL_UseALPN(
    WOLFSSL * ssl,
    char * protocol_name_list,
    unsigned int protocol_name_listSz,
    unsigned char options
)
```

wolfSSL セッションで ALPN の使用をセットアップします。

Parameters:

- **ssl** 使用する wolfSSL セッション。
- **protocol_name_list** 使用するプロトコル名のリスト。カンマ区切りの文字列が必要です。
- **protocol_name_listSz** プロトコル名リストのサイズ。
- **options** WOLFSSL_ALPN_CONTINUE_ON_MISMATCH または WOLFSSL_ALPN_FAILED_ON_MISMATCH。

See: TLSX_UseALPN

Return:

- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG ssl または protocol_name_list が null、または protocol_name_listSz が大きすぎる、または options がサポートされていない何かを含む場合に返されます。
- MEMORY_ERROR プロトコルリストのメモリ割り当てエラー。
- SSL_FAILURE 失敗時。

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // 任意の wolfSSL メソッド
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)
{
    // セッションチケット設定エラー
}
```

A.6.2.66 function wolfSSL_CTX_UseSessionTicket

```
int wolfSSL_CTX_UseSessionTicket(
    WOLFSSL_CTX * ctx
)
```

この関数は、wolfSSL コンテキストにセッションチケットを使用するよう設定します。

Parameters:

- ctx 使用する WOLFSSL_CTX 構造体。

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS 関数が正常に実行されました。
- BAD_FUNC_ARG ctx が NULL の場合に返されます。
- MEMORY_E 内部関数でメモリ割り当てエラー。

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL_METHOD method = // 何らかの wolfSSL メソッド ;
ctx = wolfSSL_CTX_new(method);

if(wolfSSL_CTX_UseSessionTicket(ctx) != SSL_SUCCESS)
{
    // セッションチケットの設定エラー
}
```

A.6.2.67 function wolfSSL_check_private_key

```
int wolfSSL_check_private_key(
    const WOLFSSL * ssl
```

)

この関数は、秘密鍵が使用されている証明書と一致していることを確認します。

Parameters:

- **ssl** 確認する WOLFSSL 構造体。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- SSL_SUCCESS 一致に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。
- <0 SSL_FAILURE 以外のすべてのエラーケースは負の値です。

Example

```
WOLFSSL* ssl;  
int ret;  
// ssl を作成してセットアップ  
ret = wolfSSL_check_private_key(ssl);  
// ret 値を確認
```

A.6.2.68 function wolfSSL_use_certificate

```
int wolfSSL_use_certificate(  
    WOLFSSL * ssl,  
    WOLFSSL_X509 * x509  
)
```

ハンドシェイク中に使用する WOLFSSL 構造体の証明書を設定するために使用されます。

Parameters:

- **ssl** 証明書を設定する WOLFSSL 構造体。
- **x509** 使用する証明書。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- SSL_SUCCESS 引数の設定に成功した場合。
- SSL_FAILURE NULL 引数が渡された場合。

Example

```
WOLFSSL* ssl;  
WOLFSSL_X509* x509  
int ret;  
// ssl オブジェクトと x509 を作成  
ret = wolfSSL_use_certificate(ssl, x509);  
// ret 値を確認
```

A.6.2.69 function wolfSSL_use_certificate_ASN1

```
int wolfSSL_use_certificate_ASN1(  
    WOLFSSL * ssl,  
    const unsigned char * der,  
    int derSz  
)
```

ハンドシェイク中に使用する WOLFSSL 構造体の証明書を設定するために使用されます。DER 形式のバッファが必要です。

Parameters:

- **ssl** 証明書を設定する WOLFSSL 構造体。
- **der** 使用する DER 証明書。
- **derSz** 渡された DER バッファのサイズ。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- SSL_SUCCESS 引数の設定に成功した場合。
- SSL_FAILURE NULL 引数が渡された場合。

Example

```
WOLFSSL* ssl;  
unsigned char* der;  
int derSz;  
int ret;  
// ssl オブジェクトを作成して DER 変数を設定  
ret = wolfSSL_use_certificate_ASN1(ssl, der, derSz);  
// ret 値を確認
```

A.6.2.70 function wolfSSL_SESSION_get_master_key

```
int wolfSSL_SESSION_get_master_key(  
    const WOLFSSL_SESSION * ses,  
    unsigned char * out,  
    int outSz  
)
```

ハンドシェイク完了後にマスターキーを取得するために使用されます。

Parameters:

- **ses** マスターシークレットバッファを取得する WOLFSSL_SESSION 構造体。
- **out** データを保持するバッファ。
- **outSz** 渡された out バッファのサイズ (0 の場合、関数は必要な最大バッファサイズを返します)。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- > 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムデータバッファがない、またはエラー状態の場合は 0 を返します。
- max 渡された outSz が 0 の場合、必要な最大バッファサイズが返されます。

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// ハンドシェイクを完了してセッション構造体を取得
bufferSz = wolfSSL_SESSION_get_master_secret(ses, NULL, 0);
buffer = malloc(bufferSz);
ret = wolfSSL_SESSION_get_master_secret(ses, buffer, bufferSz);
// ret 値を確認
```

A.6.2.71 function wolfSSL_SESSION_get_master_key_length

```
int wolfSSL_SESSION_get_master_key_length(
    const WOLFSSL_SESSION * ses
)
```

マスターシークレットキーの長さを取得するために使用されます。

Parameters:

- **ses** マスターシークレットバッファを取得する WOLFSSL_SESSION 構造体。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return: size マスターシークレットキーのサイズを返します。

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// ハンドシェイクを完了してセッション構造体を取得
bufferSz = wolfSSL_SESSION_get_master_secret_length(ses);
buffer = malloc(bufferSz);
// ret 値を確認
```

A.6.2.72 function wolfSSL_CTX_set_cert_store

```
void wolfSSL_CTX_set_cert_store(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509_STORE * str
)
```

ctx の WOLFSSL_X509_STORE 構造体の setter 関数です。

Parameters:

- **ctx** 証明書ストアポインタを設定する WOLFSSL_CTX 構造体へのポインタ。
- **str** ctx に設定する WOLFSSL_X509_STORE へのポインタ。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)

Return: none 戻り値なし。

Example

```
WOLFSSL_CTX ctx;  
WOLFSSL_X509_STORE* st;  
// ctx と st をセットアップ  
st = wolfSSL_CTX_set_cert_store(ctx, st);  
// st を使用
```

A.6.2.73 function wolfSSL_CTX_get_cert_store

```
WOLFSSL_X509_STORE * wolfSSL_CTX_get_cert_store(  
    WOLFSSL_CTX * ctx  
)
```

ctx の WOLFSSL_X509_STORE 構造体の getter 関数です。

Parameters:

- **ctx** 証明書ストアポインタを取得する WOLFSSL_CTX 構造体へのポインタ。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)
- [wolfSSL_CTX_set_cert_store](#)

Return:

- WOLFSSL_X509_STORE* ポインタの取得に成功した場合。
- NULL NULL 引数が渡された場合に返されます。

Example

```
WOLFSSL_CTX ctx;  
WOLFSSL_X509_STORE* st;  
// ctx をセットアップ  
st = wolfSSL_CTX_get_cert_store(ctx);  
// st を使用
```

A.6.2.74 function wolfSSL_get_server_random

```
size_t wolfSSL_get_server_random(  
    const WOLFSSL * ssl,  
    unsigned char * out,  
    size_t outlen  
)
```

ハンドシェイク中にサーバーから送信されたランダムデータを取得するために使用されます。

Parameters:

- **ssl** クライアントのランダムデータバッファを取得する WOLFSSL 構造体。
- **out** ランダムデータを保持するバッファ。
- **outSz** 渡された out バッファのサイズ (0 の場合、関数は必要な最大バッファサイズを返します)。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- > 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムデータバッファがない場合、またはエラー状態の場合、0 を返します。
- max 渡された outSz が 0 の場合、必要な最大バッファサイズを返します。

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);
// ret 値を確認
```

A.6.2.75 function wolfSSL_get_client_random

```
size_t wolfSSL_get_client_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outSz
)
```

ハンドシェイク中にクライアントから送信されたランダムデータを取得するために使用されます。

Parameters:

- **ssl** クライアントのランダムデータバッファを取得する WOLFSSL 構造体。
- **out** ランダムデータを保持するバッファ。
- **outSz** 渡された out バッファのサイズ (0 の場合、関数は必要な最大バッファサイズを返します)。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- > 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムデータバッファがない場合、またはエラー状態の場合、0 を返します。
- max 渡された outSz が 0 の場合、必要な最大バッファサイズを返します。

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);
// ret 値を確認
```

A.6.2.76 function wolfSSL_CTX_get_default_passwd_cb

```
wc_pem_password_cb * wolfSSL_CTX_get_default_passwd_cb(
    WOLFSSL_CTX * ctx
)
```

ctx 内に設定されたパスワードコールバックの getter 関数です。

Parameters:

- **ctx** コールバックを取得する WOLFSSL_CTX 構造体。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)

Return:

- func 成功時にはコールバック関数を返します。
- NULL ctx が NULL の場合、NULL を返します。

Example

```
WOLFSSL_CTX* ctx;  
wc_pem_password_cb cb;  
// ctx をセットアップ  
cb = wolfSSL_CTX_get_default_passwd_cb(ctx);  
//cb を使用
```

A.6.2.77 function wolfSSL_CTX_get_default_passwd_cb_userdata

```
void * wolfSSL_CTX_get_default_passwd_cb_userdata(  
    WOLFSSL_CTX * ctx  
)
```

ctx 内に設定されたパスワードコールバックユーザーデータの getter 関数です。

Parameters:

- **ctx** ユーザーデータを取得する WOLFSSL_CTX 構造体。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)

Return:

- pointer 成功時にはユーザーデータポインタを返します。
- NULL ctx が NULL の場合、NULL を返します。

Example

```
WOLFSSL_CTX* ctx;  
void* data;  
// ctx をセットアップ  
data = wolfSSL_CTX_get_default_passwd_cb_userdata(ctx);  
//data を使用
```

A.6.2.78 function wolfSSL_CTX_clear_options

```
long wolfSSL_CTX_clear_options(  
    WOLFSSL_CTX * ctx,  
    long opt  
)
```

この関数は、WOLFSSL_CTX オブジェクトのオプションビットをリセットします。

Parameters:

- **ctx** SSL コンテキストへのポインタ。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return: option 新しいオプションビット。

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_clear_options(ctx, SSL_OP_NO_TLSv1);
```

A.6.2.79 function wolfSSL_set_msg_callback

```
int wolfSSL_set_msg_callback(
    WOLFSSL * ssl,
    SSL_Msg_Cb cb
)
```

この関数は、ssl 内にコールバックを設定します。コールバックはハンドシェイクメッセージを監視するためのものです。cb の NULL 値はコールバックをリセットします。

Parameters:

- **ssl** コールバック引数を設定する WOLFSSL 構造体。

See: [wolfSSL_set_msg_callback_arg](#)

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE NULL の ssl が渡された場合。

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);
// ret を確認
```

A.6.2.80 function wolfSSL_set_msg_callback_arg

```
int wolfSSL_set_msg_callback_arg(
    WOLFSSL * ssl,
    void * arg
)
```

この関数は、ssl 内に関連するコールバックコンテキスト値を設定します。値はコールバック引数に渡されます。

Parameters:

- **ssl** コールバック引数を設定する WOLFSSL 構造体。

See: [wolfSSL_set_msg_callback](#)

Return: none 戻り値はありません。

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);
// ret を確認
wolfSSL_set_msg_callback(ssl, arg);
```

A.6.2.81 function wolfSSL_send_hrr_cookie

```
int wolfSSL_send_hrr_cookie(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

この関数は、HelloRetryRequest メッセージに Cookie を含める必要があることを示すために、サーバ側で呼び出されます。また、プロトコル DTLS v1.3 を使用する場合は、ハンドシェイクには常に Cookie 交換が含まれることを示します。プロトコル DTLS v1.3 を使用する場合は、Cookie 交換はデフォルトで有効になっていることに注意してください。Cookie は現在のトランスクリプトのハッシュを保持しており、別のサーバプロセスが応答の ClientHello を処理できるようにします。secret は、Cookie データの完全性チェックを生成する際に使用されます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **secret** secret を保持するバッファへのポインタ。NULL を渡すと、新しいランダムな secret を生成することを示します。
- **secretSz** secret のサイズ (バイト単位)。0 を渡すと、デフォルトサイズを使用することを示します: WC_SHA256_DIGEST_SIZE (SHA-256 が利用できない場合は WC_SHA_DIGEST_SIZE)。

See:

- wolfSSL_new
- wolfSSL_disable_hrr_cookie

Return:

- BAD_FUNC_ARG ssl が NULL または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。
- WOLFSSL_SUCCESS 成功した場合。
- MEMORY_ERROR secret を保存するための動的メモリの割り当てに失敗した場合。
- Another 内部エラーの場合は他の負の値。

Example

```
int ret;
WOLFSSL* ssl;
char secret[32];
...
ret = wolfSSL_send_hrr_cookie(ssl, secret, sizeof(secret));
if (ret != WOLFSSL_SUCCESS) {
    // Cookie と secret の使用設定に失敗しました。
}
```

A.6.2.82 function wolfSSL_disable_hrr_cookie

```
int wolfSSL_disable_hrr_cookie(
    WOLFSSL * ssl
)
```

この関数は、HelloRetryRequest メッセージに Cookie を含めてはならないこと、およびプロトコル DTLS v1.3 を使用している場合、ハンドシェイクに Cookie 交換を含めないことを示すために、サーバ側で呼び出されます。プロトコル DTLS v1.3 を使用する際に Cookie 交換を行わないと、サーバが DoS/増幅攻撃に対して脆弱になる可能性があることに注意してください。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_send_hrr_cookie`

Return:

- WOLFSSL_SUCCESS 成功した場合。
- BAD_FUNC_ARG ssl が NULL または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。

A.6.2.83 function wolfSSL_CTX_no_ticket_TLSv13

```
int wolfSSL_CTX_no_ticket_TLSv13(  
    WOLFSSL_CTX * ctx  
)
```

この関数は、ハンドシェイクが完了した後、再開セッションチケットの送信を停止するために、サーバで呼び出されます。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX 構造体へのポインタ。

See: `wolfSSL_no_ticket_TLSv13`

Return:

- BAD_FUNC_ARG ctx が NULL または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。
- 0 成功した場合。

Example

```
int ret;  
WOLFSSL_CTX* ctx;  
...  
ret = wolfSSL_CTX_no_ticket_TLSv13(ctx);  
if (ret != 0) {  
    // no ticket の設定に失敗しました。  
}
```

A.6.2.84 function wolfSSL_no_ticket_TLSv13

```
int wolfSSL_no_ticket_TLSv13(  
    WOLFSSL * ssl  
)
```

この関数は、ハンドシェイクが完了した後、再開セッションチケットの送信を停止するために、サーバで呼び出されます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_CTX_no_ticket_TLSv13`

Return:

- BAD_FUNC_ARG ssl が NULL または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。
- 0 成功した場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_ticket_TLsv13(ssl);
if (ret != 0) {
    // no ticket の設定に失敗しました。
}
```

A.6.2.85 function wolfSSL_CTX_no_dhe_psk

```
int wolfSSL_CTX_no_dhe_psk(
    WOLFSSL_CTX * ctx
)
```

この関数は、事前共有鍵を認証に使用するハンドシェイクの際に、Diffie-Hellman(DH) スタイルの鍵交換を禁止するために、TLS v1.3 wolfSSL コンテキストで呼び出されます。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された WOLFSSL_CTX 構造体へのポインタ。

See: [wolfSSL_no_dhe_psk](#)

Return:

- BAD_FUNC_ARG ctx が NULL または TLS v1.3 を使用していない場合。
- 0 成功した場合。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_dhe_psk(ctx);
if (ret != 0) {
    // PSK ハンドシェイクの DHE 無効化の設定に失敗しました。
}
```

A.6.2.86 function wolfSSL_no_dhe_psk

```
int wolfSSL_no_dhe_psk(
    WOLFSSL * ssl
)
```

この関数は、事前共有鍵を認証に使用するハンドシェイクの際に、Diffie-Hellman(DH) スタイルの鍵交換を禁止するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See: [wolfSSL_CTX_no_dhe_psk](#)

Return:

- BAD_FUNC_ARG ssl が NULL または TLS v1.3 を使用していない場合。

- 0 成功した場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_dhe_psk(ssl);
if (ret != 0) {
    // PSK ハンドシェイクの DHE 無効化の設定に失敗しました。
}
```

A.6.2.87 function wolfSSL_CTX_allow_post_handshake_auth

```
int wolfSSL_CTX_allow_post_handshake_auth(
    WOLFSSL_CTX * ctx
)
```

この関数は、サーバからの要求に応じてクライアント証明書をハンドシェイク後に送信できるようにするために、TLS v1.3 クライアント wolfSSL コンテキストで呼び出されます。これは、クライアント認証が必要なページとそうでないページを持つ Web サーバに接続する際に便利です。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- `wolfSSL_allow_post_handshake_auth`
- `wolfSSL_request_certificate`

Return:

- BAD_FUNC_ARG ctx が NULL または TLS v1.3 を使用していない場合。
- SIDE_ERROR サーバで呼び出された場合。
- 0 成功した場合。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_allow_post_handshake_auth(ctx);
if (ret != 0) {
    // ハンドシェイク後認証の許可に失敗しました。
}
```

A.6.2.88 function wolfSSL_allow_post_handshake_auth

```
int wolfSSL_allow_post_handshake_auth(
    WOLFSSL * ssl
)
```

この関数は、サーバからの要求に応じてクライアント証明書をハンドシェイク後に送信できるようにするために、TLS v1.3 クライアント wolfSSL で呼び出されます。Post-Handshake Client Authentication 拡張が ClientHello で送信されます。これは、クライアント認証が必要なページとそうでないページを持つ Web サーバに接続する際に便利です。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_CTX_allow_post_handshake_auth](#)
- [wolfSSL_request_certificate](#)

Return:

- BAD_FUNC_ARG ssl が NULL または TLS v1.3 を使用していない場合。
- SIDE_ERROR サーバで呼び出された場合。
- 0 成功した場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_allow_post_handshake_auth(ssl);
if (ret != 0) {
    // ハンドシェイク後認証の許可に失敗しました。
}
```

A.6.2.89 function wolfSSL_CTX_set1_sigalgs_list

```
int wolfSSL_CTX_set1_sigalgs_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

この関数は、wolfSSL コンテキストで優先順位に従って許可する署名アルゴリズムのリストを設定します。リストは null 終端のテキスト文字列で、コロン区切りのリストです。各要素は “+” の形式（例: “RSA-PSS+SHA256:ECDSA+SHA384”）で指定します。Edwards 曲線アルゴリズムである “ED25519” および “ED448” はハッシュがアルゴリズムに内包されるため、サフィックスを付けずに記述します。コンテキストに既に保持されているリストは置き換えられます。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された WOLFSSL_CTX 構造体へのポインタ。
- **list** “+” 要素（または digest を伴わない “ED25519” / “ED448”）のコロン区切りリスト。

See:

- [wolfSSL_set1_sigalgs_list](#)
- [wolfSSL_CTX_set1_groups_list](#)

Return:

- WOLFSSL_SUCCESS 成功時。
- WOLFSSL_FAILURE ポインタパラメータが NULL の場合、suites 構造体の確保に失敗した場合、またはリスト中のいずれかのトークンが認識されないか現在のビルドでサポートされていない場合。

認識される公開鍵トークンとハッシュトークン、および各トークンを利用するために必要なビルドオプションを以下の表に示します。

公開鍵トークン:

トークン	必要なビルドオプション
RSA	!NO_RSA
RSA-PSS / PSS	!NO_RSA かつ WC_RSA_PSS
ECDSA	HAVE_ECC
ED25519	HAVE_ED25519（ハッシュ指定不要）

トークン	必要なビルドオプション
ED448	HAVE_ED448 (ハッシュ指定不要)
DSA	!NO_DSA
SM2	WOLFSSL_SM2 かつ WOLFSSL_SM3 (ハッシュは SM3 固定)

ハッシュトークン:

トークン	必要なビルドオプション
SHA256	!NO_SHA256
SHA384	WOLFSSL_SHA384
SHA512	WOLFSSL_SHA512
SHA224	WOLFSSL_SHA224
SM3	WOLFSSL_SM3
SHA1	!NO_SHA かつ (!NO_OLD_TLS または WOLFSSL_ALLOW_TLS_SHA1)

TLS 1.3 における注意事項: RFC 8446 の規定により、RSA PKCS#1 v1.5、DSA、SHA-1、SHA-224 はハンドシェイク署名としては使用できず、リストに含まれていてもネゴシエーション時に除外されます。“RSA-PSS+SHAxxx”を指定した場合は `rsa_pss_rsae_shaxxx` と `rsa_pss_pss_shaxxx` の両方のスキームが追加されます。Brainpool ECDSA 署名スキーム (RFC 8734) は本文列インタフェースでは選択できず、HAVE_ECC_BRAINPOOL が有効な場合に自動でネゴシエーションされます。

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) {
    // 署名アルゴリズムリストの設定に失敗しました
}
```

A.6.2.90 function wolfSSL_set1_sigalgs_list

```
int wolfSSL_set1_sigalgs_list(
    WOLFSSL * ssl,
    const char * list
)
```

この関数は、wolfSSL セッションで優先順位に従って許可する署名アルゴリズムのリストを設定します。リストの形式、認識される公開鍵トークンおよびハッシュトークンは `wolfSSL_CTX_set1_sigalgs_list()` の説明を参照してください。セッションに既に保持されているリストは置き換えられます。

Parameters:

- `ssl` `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- `list` “+” 要素 (または digest を伴わない “ED25519” / “ED448”) のコロン区切りリスト。

See:

- `wolfSSL_CTX_set1_sigalgs_list`
- `wolfSSL_set1_groups_list`

Return:

- WOLFSSL_SUCCESS 成功時。

- WOLFSSL_FAILURE ポインタパラメータが NULL の場合、suites 構造体の確保に失敗した場合、またはリスト中のいずれかのトークンが認識されないか現在のビルドでサポートされていない場合。

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) {
    // 署名アルゴリズムリストの設定に失敗しました
}
```

A.6.2.91 function wolfSSL_CTX_set1_groups_list

```
int wolfSSL_CTX_set1_groups_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

この関数は、wolfSSL コンテキストで優先順位に従って許可する鍵交換グループ（名前付き楕円曲線および KEM）のリストを設定します。リストは null 終端のテキスト文字列で、グループ名をコロンで区切って指定します（例: “P-384:P-256:X25519”）。TLS v1.3 接続で使用する鍵交換パラメータを設定するために本関数を呼び出します（本関数は HAVE_ECC、WOLFSSL_TLS13、HAVE_SUPPORTED_CURVES がすべて定義された場合にのみ提供されます）。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された WOLFSSL_CTX 構造体へのポインタ。
- **list** 鍵交換グループ名のコロン区切りリストである文字列。

See:

- wolfSSL_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return:

- WOLFSSL_FAILURE ポインタパラメータが NULL の場合、グループが WOLFSSL_MAX_GROUP_COUNT を超える場合、グループ名が認識されない場合、または TLS v1.3 を使用していない場合。
- WOLFSSL_SUCCESS 成功時。

認識されるグループ名と必要なビルドオプションを以下に示します。名前は表とケースセンシティブに比較されます。

NIST / SEC 曲線（HAVE_ECC が必要）：

名前	曲線 / グループ
P-160	secp160r1
P-160-2	secp160r2
P-192	secp192r1 (prime192v1)
P-224	secp224r1
P-256	secp256r1 (prime256v1) — “prime256v1” / “secp256r1” でも指定可
P-384	secp384r1 — “secp384r1” でも指定可
P-521	secp521r1 — “secp521r1” でも指定可

名前	曲線 / グループ
K-160	secp160k1
K-192	secp192k1
K-224	secp224k1
K-256	secp256k1

Brainpool 曲線 (HAVE_ECC に加えて WOLFSSL_CUSTOM_CURVES と HAVE_ECC_BRAINPOOL が必要。通常は `-enable-eccustcurves=all` で有効化) :

名前	曲線
B-256	brainpoolP256r1
B-384	brainpoolP384r1
B-512	brainpoolP512r1

Edwards / Montgomery 曲線:

名前	必要なビルドオプション
X25519	HAVE_CURVE25519
X448	HAVE_CURVE448

SM2 (WOLFSSL_SM2 が必要) :

名前	グループ
SM2	sm2p256v1
sm2p256v1	sm2p256v1 (別名)

ML-KEM (耐量子) グループ (WOLFSSL_HAVE_MLKEM かつ!WOLFSSL_NO_ML_KEM が必要) :

名前
ML_KEM_512
ML_KEM_768
ML_KEM_1024

ML-KEM ハイブリッドグループは、上記に加えて HAVE_ECC、および WOLFSSL_WC_MLKEM または HAVE_LIBOQS、さらに WOLFSSL_PQC_HYBRIDS (“extra” セットには WOLFSSL_EXTRA_PQC_HYBRIDS) が必要です:

名前	必要なハイブリッドフラグ
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

名前	必要なハイブリッドフラグ
X448MLKEM768	WOLFSSL_EXTRA_PQC_HYBRIDS

レガシー Kyber グループ (WOLFSSL_MLKEM_KYBER が必要。ハイブリッドはさらに HAVE_ECC と、WOLFSSL_WC_MLKEM または HAVE_LIBOQS が必要) :

名前
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

上記の名前に加え、HAVE_FIPS および HAVE_SELFTEST が定義されていないビルドでは、wolfCrypt に登録されている任意の曲線名 (wc_ecc_get_curve_idx_from_name() で参照、例: "brainpoolP256r1") も受理されます。

リストに指定した名前の順序はそのまま保持され、TLS 1.3 における KeyShare の優先順位として使用されます。

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) {
    // グループリストの設定に失敗しました
}
```

A.6.2.92 function wolfSSL_set1_groups_list

```
int wolfSSL_set1_groups_list(
    WOLFSSL * ssl,
    const char * list
)
```

この関数は、wolfSSL セッションで優先順位に従って許可する鍵交換グループ (名前付き楕円曲線および KEM) のリストを設定します。リストの形式および認識されるグループ名は `wolfSSL_CTX_set1_groups_list()` の説明を参照してください。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **list** 鍵交換グループのコロン区切りリストである文字列。

See:

- `wolfSSL_CTX_set1_groups_list`

- `wolfSSL_CTX_set_groups`
- `wolfSSL_set_groups`
- `wolfSSL_UseKeyShare`
- `wolfSSL_preferred_group`

Return:

- WOLFSSL_FAILURE ポインタパラメータが NULL の場合、グループが WOLFSSL_MAX_GROUP_COUNT を超える場合、グループ名が認識されない場合、または TLS v1.3 を使用していない場合。
- WOLFSSL_SUCCESS 成功時。

Example

```
int ret;
WOLFSSL* ssl;
const char* list = "P-384:P-256";
...
ret = wolfSSL_CTX_set1_groups_list(ssl, list);
if (ret != WOLFSSL_SUCCESS) {
    // グループリストの設定に失敗しました
}
```

A.6.2.93 function wolfSSL_CTX_set_groups

```
int wolfSSL_CTX_set_groups(
    WOLFSSL_CTX * ctx,
    int * groups,
    int count
)
```

この関数は、wolfSSL コンテキストで優先順位に従って許可する鍵交換グループ（名前付き楕円曲線および KEM）のリストを設定します。リストは名前付きグループ識別子（下表参照）の配列で、count に配列内の識別子数を指定します。TLS v1.3 接続で使用する鍵交換パラメータを設定するために本関数を呼び出します。配列の順序がそのまま KeyShare の優先順位として用いられます。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された WOLFSSL_CTX 構造体へのポインタ。
- **groups** 識別子による鍵交換グループのリスト。
- **count** groups 内の識別子数（WOLFSSL_MAX_GROUP_COUNT を超えてはならない）。

See:

- `wolfSSL_set_groups`
- `wolfSSL_UseKeyShare`
- `wolfSSL_CTX_set1_groups_list`
- `wolfSSL_set1_groups_list`
- `wolfSSL_preferred_group`

Return:

- BAD_FUNC_ARG ポインタパラメータが NULL の場合、count が WOLFSSL_MAX_GROUP_COUNT を超える場合、または基底メソッドが TLS メソッドでない場合。
- WOLFSSL_SUCCESS 成功時。

認識される識別子と必要なビルドオプションを以下に示します。識別子は <wolfssl/ssl.h> 内の無名 enum で定義されています。

NIST / SEC 曲線（HAVE_ECC が必要）：

識別子	曲線 / グループ
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 曲線 (HAVE_ECC に加えて WOLFSSL_CUSTOM_CURVES と HAVE_ECC_BRAINPOOL が必要。通常は `-enable-eccustcurves=all` で有効化) :

識別子	曲線	備考
WOLFSSL_ECC_BRAINPOOLP256R1	brainpoolP256r1	TLS 1.2 グループ ID 26
WOLFSSL_ECC_BRAINPOOLP384R1	brainpoolP384r1	TLS 1.2 グループ ID 27
WOLFSSL_ECC_BRAINPOOLP512R1	brainpoolP512r1	TLS 1.2 グループ 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 曲線:

識別子	必要なビルドオプション
WOLFSSL_ECC_X25519	HAVE_CURVE25519
WOLFSSL_ECC_X448	HAVE_CURVE448

SM2 (WOLFSSL_SM2 が必要) :

識別子	グループ
WOLFSSL_ECC_SM2P256V1	sm2p256v1

有限体 DH (RFC 7919) グループ (HAVE_FFDHE および各サイズに対応する HAVE_FFDHE_NNNN マクロが必要) :

識別子	グループ
WOLFSSL_FFDHE_2048	ffdhe2048
WOLFSSL_FFDHE_3072	ffdhe3072
WOLFSSL_FFDHE_4096	ffdhe4096
WOLFSSL_FFDHE_6144	ffdhe6144
WOLFSSL_FFDHE_8192	ffdhe8192

ML-KEM (耐量子) グループ (HAVE_PQC、WOLFSSL_HAVE_MLKEM、!WOLFSSL_NO_ML_KEM が必要) :

識別子

WOLFSSL_ML_KEM_512
WOLFSSL_ML_KEM_768
WOLFSSL_ML_KEM_1024

ML-KEM ハイブリッドグループは、上記に加えて HAVE_ECC、および WOLFSSL_WC_MLKEM または HAVE_LIBOQS、さらに WOLFSSL_PQC_HYBRIDS (“extra” セットには WOLFSSL_EXTRA_PQC_HYBRIDS) が必要です:

識別子	必要なハイブリッドフラグ
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

レガシー Kyber グループ (HAVE_PQC と WOLFSSL_MLKEM_KYBER が必要。ハイブリッドはさらに HAVE_ECC と、WOLFSSL_WC_MLKEM または HAVE_LIBOQS が必要):

識別子

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) {
    // グループリストの設定に失敗しました
}
```

A.6.2.94 function wolfSSL_set_groups

```
int wolfSSL_set_groups(
    WOLFSSL * ssl,
```

```

    int * groups,
    int count
)

```

この関数は、wolfSSL セッションで優先順位に従って許可する鍵交換グループ（名前付き楕円曲線および KEM）のリストを設定します。配列の形式および認識される識別子は `wolfSSL_CTX_set_groups()` の説明を参照してください。

Parameters:

- `ssl` `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- `groups` 識別子による鍵交換グループのリスト。
- `count` `groups` 内の識別子数（WOLFSSL_MAX_GROUP_COUNT を超えてはならない）。

See:

- `wolfSSL_CTX_set_groups`
- `wolfSSL_UseKeyShare`
- `wolfSSL_CTX_set1_groups_list`
- `wolfSSL_set1_groups_list`
- `wolfSSL_preferred_group`

Return:

- BAD_FUNC_ARG ポインタパラメータが NULL の場合、`count` が WOLFSSL_MAX_GROUP_COUNT を超える場合、識別子のいずれかが認識されない場合、または基底メソッドが TLS メソッドでない場合。
- WOLFSSL_SUCCESS 成功時。

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) {
    // グループリストの設定に失敗しました
}

```

A.6.2.95 function wolfSSL_CTX_set_max_early_data

```

int wolfSSL_CTX_set_max_early_data(
    WOLFSSL_CTX * ctx,
    unsigned int sz
)

```

この関数は、TLS v1.3 クライアントまたはサーバが wolfSSL コンテキストを使用して交換する意思のある早期データの最大量を設定します。リプレイ攻撃を軽減するために処理する早期データの量を制限するには、この関数を呼び出してください。早期データは、セッションチケットが送信された接続の鍵から派生した鍵によって保護されるため、セッションチケットが再開に使用されるたびに同じになります。この値は再開用のセッションチケットに含まれます。サーバの値がゼロの場合、セッションチケットを使用してクライアントが早期データを送信しないことを示します。クライアントの値がゼロの場合、クライアントが早期データを送信しないことを示します。早期データのバイト数は、アプリケーションで実用的に可能な限り低く保つことをお勧めします。

Parameters:

- `ctx` `wolfSSL_CTX_new()` で作成された WOLFSSL_CTX 構造体へのポインタ。
- `sz` 受け入れる早期データの量 (バイト単位)。

See:

- [wolfSSL_set_max_early_data](#)
- [wolfSSL_write_early_data](#)
- [wolfSSL_read_early_data](#)

Return:

- BAD_FUNC_ARG ctx が NULL の場合、または TLS v1.3 を使用していない場合。
- 0 成功時。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_set_max_early_data(ctx, 128);
if (ret != WOLFSSL_SUCCESS) {
    // グループリストの設定に失敗しました
}
```

A.6.2.96 function wolfSSL_set_max_early_data

```
int wolfSSL_set_max_early_data(
    WOLFSSL * ssl,
    unsigned int sz
)
```

この関数は、TLS v1.3 クライアントまたはサーバが交換する意思のある早期データの最大量を設定します。リプレイ攻撃を軽減するために処理する早期データの量を制限するには、この関数を呼び出してください。早期データは、セッションチケットが送信された接続の鍵から派生した鍵によって保護されるため、セッションチケットが再開に使用されるたびに同じになります。この値は再開用のセッションチケットに含まれます。サーバの値がゼロの場合、セッションチケットを使用してクライアントが早期データを送信しないことを示します。クライアントの値がゼロの場合、クライアントが早期データを送信しないことを示します。早期データのバイト数は、アプリケーションで実用的に可能な限り低く保つことをお勧めします。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **sz** クライアントから受け入れる早期データの量 (バイト単位)。

See:

- [wolfSSL_CTX_set_max_early_data](#)
- [wolfSSL_write_early_data](#)
- [wolfSSL_read_early_data](#)

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- 0 成功時。 *Example*

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_set_max_early_data(ssl, 128);
if (ret != WOLFSSL_SUCCESS) {
    // グループリストの設定に失敗しました
}
```


A.6.2.97 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  
)
```

この関数は TLS v1.3 接続のための事前共有鍵 (PSK) クライアント側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL_CTX 構造体の client_psk_tls13_cb メンバーを設定します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された WOLFSSL_CTX 構造体へのポインタ。
- **cb** TLS 1.3 クライアント用の事前共有鍵 (PSK) コールバック。

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);
```

A.6.2.98 function wolfSSL_set_psk_client_tls13_callback

```
void wolfSSL_set_psk_client_tls13_callback(  
    WOLFSSL * ssl,  
    wc_psk_client_tls13_callback cb  
)
```

この関数は TLS v1.3 接続のための事前共有鍵 (PSK) クライアント側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL 構造体の options フィールドの client_psk_tls13_cb メンバーを設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** TLS 1.3 クライアント用の事前共有鍵 (PSK) コールバック。

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);
```

A.6.2.99 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  
)
```

この関数は TLS v1.3 接続のための事前共有鍵 (PSK) サーバー側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL_CTX 構造体の server_psk_tls13_cb メンバーを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された WOLFSSL_CTX 構造体へのポインタ。
- **cb** TLS 1.3 サーバー用の事前共有鍵 (PSK) コールバック。

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);
```

A.6.2.100 function wolfSSL_set_psk_server_tls13_callback

```
void wolfSSL_set_psk_server_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_server_tls13_callback cb
)
```

この関数は TLS v1.3 接続のための事前共有鍵 (PSK) サーバー側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL 構造体の options フィールドの server_psk_tls13_cb メンバーを設定します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** TLS 1.3 サーバー用の事前共有鍵 (PSK) コールバック。

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);
```

A.6.2.101 function wolfSSL_UseKeyShare

```
int wolfSSL_UseKeyShare(
    WOLFSSL * ssl,
    word16 group
)
```

この関数は鍵ペアの生成を含む、グループから鍵共有エントリを作成します。KeyShare 拡張には鍵交換用に生成されたすべての公開鍵が含まれます。この関数が呼び出されると、指定されたグループのみが含まれます。サーバーに対して優先グループが以前に確立されている場合に、この関数を呼び出します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **group** 鍵交換グループ識別子。

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` ssl が NULL の場合。
- `MEMORY_E` 動的メモリ割り当てが失敗した場合。
- `WOLFSSL_SUCCESS` 成功した場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_UseKeyShare(ssl, WOLFSSL_ECC_X25519);
if (ret != WOLFSSL_SUCCESS) {
    // 鍵共有の設定に失敗しました
}
```

A.6.2.102 function `wolfSSL_NoKeyShares`

```
int wolfSSL_NoKeyShares(
    WOLFSSL * ssl
)
```

この関数は ClientHello で鍵共有が送信されないようにするために呼び出されます。これにより、ハンドシェイクで鍵交換が必要な場合、サーバーは HelloRetryRequest で応答することになります。期待される鍵交換グループが不明で、不要な鍵の生成を避けたい場合にこの関数を呼び出します。鍵交換が必要な場合、ハンドシェイクを完了するために追加のラウンドトリップが必要になることに注意してください。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_UseKeyShare`

Return:

- `BAD_FUNC_ARG` ssl が NULL の場合。
- `SIDE_ERROR` サーバーで呼び出された場合。
- `WOLFSSL_SUCCESS` 成功した場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_NoKeyShares(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // 鍵共有なしの設定に失敗しました
}
```

A.6.2.103 function wolfTLSv1_3_server_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method_ex(
    void * heap
)
```

この関数はアプリケーションがサーバーであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。

Parameters:

- **heap** 静的メモリアロケータが動的メモリ割り当て時に使用するバッファへのポインタ。

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:

- 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC の呼び出し時にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method_ex(NULL);
if (method == NULL) {
    // メソッドの取得ができません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.104 function wolfTLSv1_3_client_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method_ex(
    void * heap
)
```

この関数はアプリケーションがクライアントであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。

Parameters:

- **heap** 静的メモリアロケータが動的メモリ割り当て時に使用するバッファへのポインタ。

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:

- 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC の呼び出し時にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>
```

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;    method = wolfTLSv1_3_client_method_ex(NULL);
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.105 function wolfTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method(
    void
)
```

この関数は、アプリケーションがサーバーであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体用のメモリを割り当て、初期化します。

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:

- 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC 呼び出し時にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常、errno が ENOMEM に設定された NULL)。

Example

```
#include <wolfssl/ssl.h>
```

```
WOLFSSL_METHOD* method;
```

```

WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

A.6.2.106 function wolfTLSv1_3_client_method

```

WOLFSSL_METHOD * wolfTLSv1_3_client_method(
    void
)

```

この関数は、アプリケーションがクライアントであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体用のメモリを割り当て、初期化します。

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:

- 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL_XMALLOC 呼び出し時にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常、errno が ENOMEM に設定された NULL)。

Example

```

#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_client_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

A.6.2.107 function wolfTLSv1_3_method_ex

```

WOLFSSL_METHOD * wolfTLSv1_3_method_ex(
    void * heap
)

```

この関数は、どちら側 (サーバー/クライアント) であるかがまだ決定されていないことを除いて、wolfTLSv1_3_client_method と同様の WOLFSSL_METHOD を返します。

Parameters:

- **heap** 静的メモリアロケータが動的メモリ割り当て中に使用するバッファへのポインタ。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- WOLFSSL_METHOD 正常に作成された場合、WOLFSSL_METHOD ポインタを返します。
- NULL メモリ割り当てエラーまたはメソッドの作成に失敗した場合は Null。

Example

```
WOLFSSL* ctx;  
ctx = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));  
// ret 値を確認
```

A.6.2.108 function wolfTLSv1_3_method

```
WOLFSSL_METHOD * wolfTLSv1_3_method(  
    void  
)
```

この関数は、どちら側 (サーバー/クライアント) であるかがまだ決定されていないことを除いて、wolfTLSv1_3_client_method と同様の WOLFSSL_METHOD を返します。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- WOLFSSL_METHOD 正常に作成された場合、WOLFSSL_METHOD ポインタを返します。
- NULL メモリ割り当てエラーまたはメソッドの作成に失敗した場合は Null。

Example

```
WOLFSSL* ctx;  
ctx = wolfSSL_CTX_new(wolfTLSv1_3_method());  
// ret 値を確認
```

A.6.2.109 function wolfSSL_CTX_set_client_cert_type

```
int wolfSSL_CTX_set_client_cert_type(  
    WOLFSSL_CTX * ctx,  
    const char * buf,  
    int len  
)
```

この関数がクライアント側で呼び出された場合、ピアに送信できる証明書タイプを設定します。サーバー側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が “Raw public key” 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクトポインタ。
- **buf** 証明書タイプが格納されるバッファ。
- **len** buf サイズ (バイト単位)(含まれる証明書タイプの数と同じ)。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 証明書タイプが正常に設定された場合。
- BAD_FUNC_ARG ctx に NULL が渡された場合、証明書タイプとして不正な値が指定された場合、buf サイズが MAX_CLIENT_CERT_TYPE_CNT を超えた場合、または buf 内に重複する値が見つかった場合。

A.6.2.110 function wolfSSL_CTX_set_server_cert_type

```
int wolfSSL_CTX_set_server_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

この関数がサーバー側で呼び出された場合、ピアに送信できる証明書タイプを設定します。クライアント側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が "Raw public key" 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクトポインタ。
- **buf** 証明書タイプが格納されるバッファ。
- **len** buf サイズ (バイト単位)(含まれる証明書タイプの数と同じ)。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 証明書タイプが正常に設定された場合。
- BAD_FUNC_ARG ctx に NULL が渡された場合、証明書タイプとして不正な値が指定された場合、buf サイズが MAX_SERVER_CERT_TYPE_CNT を超えた場合、または buf 内に重複する値が見つかった場合。

A.6.2.111 function wolfSSL_set_client_cert_type

```
int wolfSSL_set_client_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

この関数がクライアント側で呼び出された場合、ピアに送信できる証明書タイプを設定します。サーバー側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が “Raw public key” 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **buf** 証明書タイプが格納されるバッファ。
- **len** buf サイズ (バイト単位)(含まれる証明書タイプの数と同じ)。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 証明書タイプが正常に設定された場合。
- BAD_FUNC_ARG ctx に NULL が渡された場合、証明書タイプとして不正な値が指定された場合、buf サイズが MAX_CLIENT_CERT_TYPE_CNT を超えた場合、または buf 内に重複する値が見つかった場合。

A.6.2.112 function wolfSSL_set_server_cert_type

```
int wolfSSL_set_server_cert_type(
```

```

    WOLFSSL * ssl,
    const char * buf,
    int len
)

```

この関数がサーバー側で呼び出された場合、ピアに送信できる証明書タイプを設定します。クライアント側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が “Raw public key” 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクトポインタ。
- **buf** 証明書タイプが格納されるバッファ。
- **len** buf サイズ (バイト単位)(含まれる証明書タイプの数と同じ)。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 証明書タイプが正常に設定された場合。
- BAD_FUNC_ARG ctx に NULL が渡された場合、証明書タイプとして不正な値が指定された場合、buf サイズが MAX_SERVER_CERT_TYPE_CNT を超えた場合、または buf 内に重複する値が見つかった場合。

A.6.2.113 function wolfSSL_CTX_clear_group_messages

```

int wolfSSL_CTX_clear_group_messages(
    WOLFSSL_CTX * ctx
)

```

指定された WOLFSSL_CTX コンテキストに対してハンドシェイクメッセージグループ化を無効にします。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ。

See:

- [wolfSSL_CTX_set_group_messages](#)
- [wolfSSL_set_group_messages](#)
- [wolfSSL_clear_group_messages](#)

Return:

- WOLFSSL_SUCCESS 成功時。

- BAD_FUNC_ARG ctx が NULL の場合。

この関数は、指定されたコンテキストから作成されたすべての SSL オブジェクトに対してハンドシェイクメッセージグループ化をオフにします。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());  
wolfSSL_CTX_clear_group_messages(ctx);
```

A.6.2.114 function wolfSSL_clear_group_messages

```
int wolfSSL_clear_group_messages(  
    WOLFSSL * ssl  
)
```

指定された WOLFSSL オブジェクトに対してハンドシェイクメッセージグループ化を無効にします。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_set_group_messages](#)
- [wolfSSL_CTX_set_group_messages](#)
- [wolfSSL_CTX_clear_group_messages](#)

Return:

- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG ssl が NULL の場合。

この関数は、指定された SSL オブジェクトに対してハンドシェイクメッセージグループ化をオフにします。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);  
wolfSSL_clear_group_messages(ssl);
```

A.6.2.115 function wolfSSL_GetCookieCtx

```
void * wolfSSL_GetCookieCtx(  
    WOLFSSL * ssl  
)
```

この関数は WOLFSSL 構造体の IOCB_CookieCtx メンバーを返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_SetCookieCtx](#)
- [wolfSSL_CTX_SetGenCookie](#)

Return:

- pointer 関数は IOCB_CookieCtx に保存されている void 型ポインタ値を返します。
- NULL WOLFSSL 構造体が NULL の場合。

Example

```

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
void* cookie;
...
cookie = wolfSSL_GetCookieCtx(ssl);
if(cookie != NULL){
// cookie を取得しました
}

```

A.6.2.116 function wolfSSL_SetIO_ISOTP

```

int wolfSSL_SetIO_ISOTP(
    WOLFSSL * ssl,
    isotp_wolfssl_ctx * ctx,
    can_rcv_fn rcv_fn,
    can_snd_fn send_fn,
    can_delay_fn delay_fn,
    word32 receive_delay,
    char * receive_buffer,
    int receive_buffer_size,
    void * arg
)

```

この関数は、wolfSSL が WOLFSSL_ISOTP でコンパイルされている場合に使用するために、wolfSSL の ISO-TP コンテキストを設定します。

Parameters:

- **ssl** wolfSSL コンテキスト。
- **ctx** この関数が初期化するユーザー作成の ISOTP コンテキスト。
- **rcv_fn** ユーザーの CAN バス受信コールバック。
- **send_fn** ユーザーの CAN バス送信コールバック。
- **delay_fn** ユーザーのマイクロ秒粒度遅延関数。
- **receive_delay** 各 CAN バスパケットを遅延させる設定マイクロ秒数。
- **receive_buffer** データを受信するためのユーザー提供バッファ、ISOTP_DEFAULT_BUFFER_SIZE バイトに割り当ててことを推奨。
- **receive_buffer_size** - receive_buffer のサイズ。
- **arg** rcv_fn と send_fn に送信される任意のポインタ。

Return: 0 成功時、WOLFSSL_CBIO_ERR_GENERAL 失敗時。

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);

```

A.6.2.117 function wolfSSL_SSLEnableRead

```

void wolfSSL_SSLEnableRead(
    WOLFSSL * ssl
)

```

この関数は IO レイヤーからの読み取りを無効にします。

Parameters:

- **ssl** wolfSSL コンテキスト。

See:

- [wolfSSL_CTX_SetIORecv](#)
- [wolfSSL_SSLSetIORecv](#)
- [wolfSSL_SSEnableRead](#)

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLDisableRead(ssl);
```

A.6.2.118 function wolfSSL_SSEnableRead

```
void wolfSSL_SSEnableRead(
    WOLFSSL * ssl
)
```

この関数は IO レイヤーからの読み取りを有効にします。読み取りはデフォルトで有効になっており、wolfSSL_SSLDisableRead() を元に戻すために使用する必要があります。

Parameters:

- **ssl** wolfSSL コンテキスト。

See:

- [wolfSSL_CTX_SetIORecv](#)
- [wolfSSL_SSLSetIORecv](#)
- [wolfSSL_SSEnableRead](#)

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLDisableRead(ssl);
...
wolfSSL_SSEnableRead(ssl);
```

A.7 wolfSSL エラー処理と報告

A.7.1 Functions

	Name
int	wolfSSL_Debugging_ON (void) ビルド時にロギングが有効になっている場合、この関数は実行時にロギングを有効にします。ビルド時にロギングを有効にするには、 <code>-enable-debug</code> を使用するか、 <code>DEBUG_WOLFSSL</code> を定義します。
void	wolfSSL_Debugging_OFF (void) この関数は、実行時のロギングメッセージを無効にします。既に無効になっている場合、何も行われません。

	Name
int	wolfSSL_get_error (WOLFSSL * ssl, int ret) この関数は、前回の API 関数呼び出し (wolfSSL_connect、wolfSSL_accept、wolfSSL_read、wolfSSL_write 等) がエラーリターンコード (SSL_FAILURE) になった理由を説明する一意のエラーコードを返します。前回の関数の戻り値は、ret を通じて wolfSSL_get_error に渡されます。wolfSSL_get_error が呼び出されて一意のエラーコードを返した後、wolfSSL_ERR_error_string() を呼び出して、人間が読めるエラー文字列を取得できます。詳細については、wolfSSL_ERR_error_string() を参照してください。
void	wolfSSL_load_error_strings (void) この関数は OpenSSL 互換性 (SSL_load_error_string) のためのものであり、何も動作を行いません。
char *	wolfSSL_ERR_error_string (unsigned long errNumber, char * data) この関数は wolfSSL_get_error() によって返されたエラーコードを、より人間が読みやすいエラー文字列に変換します。errNumber は wolfSSL_get_error() によって返されたエラーコードであり、data はエラー文字列が配置される格納バッファです。data の最大長はデフォルトで 80 文字であり、wolfssl/wolfcrypt/error.h の MAX_ERROR_SZ で定義されています。
void	wolfSSL_ERR_error_string_n (unsigned long e, char * buf, unsigned long len) この関数は wolfSSL_ERR_error_string() のバージョンであり、len は buf に書き込まれる最大文字数を指定します。wolfSSL_ERR_error_string() と同様に、この関数は wolfSSL_get_error() から返されたエラーコードをより人間が読みやすいエラー文字列に変換します。人間が読みやすい文字列は buf に配置されます。
void	wolfSSL_ERR_print_errors_fp (XFILE fp, int err) この関数は、wolfSSL_get_error() によって返されたエラーコードを、より人間が読みやすいエラー文字列に変換し、その文字列を出力ファイル fp に出力します。err は wolfSSL_get_error() によって返されたエラーコードであり、fp はエラー文字列が配置されるファイルです。
void	wolfSSL_ERR_print_errors_cb (int)(const char str, size_t len, void u) cb, void u) この関数は、提供されたコールバックを使用してエラーレポートを処理します。コールバック関数は、各エラー行に対して実行されます。文字列、長さ、およびユーザデータがコールバックパラメータに渡されます。

	Name
int	wolfSSL_want_read (WOLFSSL * ssl) この関数は、wolfSSL_get_error() を呼び出して SSL_ERROR_WANT_READ が返される場合と同様です。基礎となるエラー状態が SSL_ERROR_WANT_READ の場合、この関数は 1 を返し、それ以外の場合は 0 を返します。
int	wolfSSL_want_write (WOLFSSL * ssl) この関数は、wolfSSL_get_error() を呼び出して SSL_ERROR_WANT_WRITE が返される場合と同様です。基礎となるエラー状態が SSL_ERROR_WANT_WRITE の場合、この関数は 1 を返し、それ以外の場合は 0 を返します。
unsigned long	wolfSSL_ERR_peek_last_error (void) この関数は、WOLFSSL_ERROR から発生した最後のエラーの絶対値を返します。

A.7.2 Functions Documentation

A.7.2.1 function wolfSSL_Debugging_ON

```
int wolfSSL_Debugging_ON(
    void
)
```

ビルド時にロギングが有効になっている場合、この関数は実行時にロギングを有効にします。ビルド時にロギングを有効にするには、-enable-debug を使用するか、DEBUG_WOLFSSL を定義します。

Parameters:

- **none** パラメータなし。

See:

- **wolfSSL_Debugging_OFF**
- **wolfSSL_SetLoggingCb**

Return:

- 0 成功時。
- NOT_COMPILED_IN このビルドでロギングが有効になっていない場合に返されるエラーです。

Example

```
wolfSSL_Debugging_ON();
```

A.7.2.2 function wolfSSL_Debugging_OFF

```
void wolfSSL_Debugging_OFF(
    void
)
```

この関数は、実行時のロギングメッセージを無効にします。既に無効になっている場合、何も行われません。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_Debugging_ON](#)
- [wolfSSL_SetLoggingCb](#)

Return: none 戻り値なし。

Example

```
wolfSSL_Debugging_OFF();
```

A.7.2.3 function wolfSSL_get_error

```
int wolfSSL_get_error(  
    WOLFSSL * ssl,  
    int ret  
)
```

この関数は、前回の API 関数呼び出し (wolfSSL_connect、wolfSSL_accept、wolfSSL_read、wolfSSL_write 等) がエラーリターンコード (SSL_FAILURE) になった理由を説明する一意のエラーコードを返します。前回の関数の戻り値は、ret を通じて wolfSSL_get_error に渡されます。wolfSSL_get_error が呼び出されて一意のエラーコードを返した後、wolfSSL_ERR_error_string() を呼び出して、人間が読めるエラー文字列を取得できます。詳細については、wolfSSL_ERR_error_string() を参照してください。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) で作成された SSL オブジェクトへのポインタ。
- **ret** エラーリターンコードになった前回の関数の戻り値。

See:

- [wolfSSL_ERR_error_string](#)
- [wolfSSL_ERR_error_string_n](#)
- [wolfSSL_ERR_print_errors_fp](#)
- [wolfSSL_load_error_strings](#)

Return:

- 正常に完了した場合、この関数は前回の API 関数が失敗した理由を説明する一意のエラーコードを返します。
- SSL_ERROR_NONE は、ret > 0 の場合に返されます。ret <= 0 の場合、前回の API がエラーコードを返したように見えても実際にはエラーが発生していなかった場合に、この値が返されることがあります。例えば、sz パラメータがゼロの wolfSSL_read() を呼び出した場合です。wolfSSL_read() からの 0 の戻り値は通常エラーを示しますが、この場合はエラーは発生していません。その後 wolfSSL_get_error() を呼び出すと、SSL_ERROR_NONE が返されます。

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);
```

A.7.2.4 function wolfSSL_load_error_strings

```
void wolfSSL_load_error_strings(  
    void  
)
```


この関数は OpenSSL 互換性 (`SSL_load_error_string`) のためのものであり、何も動作を行いません。

Parameters:

- **none** パラメータはありません。

See:

- `wolfSSL_get_error`
- `wolfSSL_ERR_error_string`
- `wolfSSL_ERR_error_string_n`
- `wolfSSL_ERR_print_errors_fp`
- `wolfSSL_load_error_strings`

Return: none 戻り値はありません。

Example

```
wolfSSL_load_error_strings();
```

A.7.2.5 function wolfSSL_ERR_error_string

```
char * wolfSSL_ERR_error_string(  
    unsigned long errNumber,  
    char * data  
)
```

この関数は `wolfSSL_get_error()` によって返されたエラーコードを、より人間が読みやすいエラー文字列に変換します。`errNumber` は `wolfSSL_get_error()` によって返されたエラーコードであり、`data` はエラー文字列が配置される格納バッファです。`data` の最大長はデフォルトで 80 文字であり、`wolfssl/wolfcrypt/error.h` の `MAX_ERROR_SZ` で定義されています。

Parameters:

- **errNumber** `wolfSSL_get_error()` によって返されたエラーコード。
- **data** `errNumber` に一致する人間が読みやすいエラー文字列を含む出力バッファ。

See:

- `wolfSSL_get_error`
- `wolfSSL_ERR_error_string_n`
- `wolfSSL_ERR_print_errors_fp`
- `wolfSSL_load_error_strings`

Return:

- success 正常に完了した場合、この関数は `data` で返されるのと同じ文字列を返します。
- failure 失敗時には、この関数は適切な失敗理由 `msg` を含む文字列を返します。

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);
```

A.7.2.6 function wolfSSL_ERR_error_string_n

```
void wolfSSL_ERR_error_string_n(  
    unsigned long e,  
    char * buf,  
    unsigned long len  
)
```

この関数は wolfSSL_ERR_error_string() のバージョンであり、len は buf に書き込まれる最大文字数を指定します。wolfSSL_ERR_error_string() と同様に、この関数は wolfSSL_get_error() から返されたエラーコードをより人間が読みやすいエラー文字列に変換します。人間が読みやすい文字列は buf に配置されます。

Parameters:

- **e** wolfSSL_get_error()によって返されたエラーコード。
- **buff** e に一致する人間が読みやすいエラー文字列を含む出力バッファ。
- **len** buf に書き込まれる最大文字数。

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: none 戻り値はありません。

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);
```

A.7.2.7 function wolfSSL_ERR_print_errors_fp

```
void wolfSSL_ERR_print_errors_fp(  
    XFILE fp,  
    int err  
)
```

この関数は、wolfSSL_get_error() によって返されたエラーコードを、より人間が読みやすいエラー文字列に変換し、その文字列を出力ファイル fp に出力します。err は wolfSSL_get_error() によって返されたエラーコードであり、fp はエラー文字列が配置されるファイルです。

Parameters:

- **fp** 人間が読みやすいエラー文字列が書き込まれる出力ファイル。
- **err** wolfSSL_get_error()によって返されたエラーコード。

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: none 戻り値なし。

Example

```
int err = 0;
WOLFSSL* ssl;
FILE* fp = ...
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_print_errors_fp(fp, err);
```

A.7.2.8 function wolfSSL_ERR_print_errors_cb

```
void wolfSSL_ERR_print_errors_cb(
    int (*)(const char *str, size_t len, void *u) cb,
    void * u
)
```

この関数は、提供されたコールバックを使用してエラーレポートを処理します。コールバック関数は、各エラー一行に対して実行されます。文字列、長さ、およびユーザデータがコールバックパラメータに渡されます。

Parameters:

- **cb** コールバック関数。
- **u** コールバック関数に渡すユーザデータ。

See:

- [wolfSSL_get_error](#)
- [wolfSSL_ERR_error_string](#)
- [wolfSSL_ERR_error_string_n](#)
- [wolfSSL_load_error_strings](#)

Return: none 戻り値なし。

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);
```

A.7.2.9 function wolfSSL_want_read

```
int wolfSSL_want_read(
    WOLFSSL * ssl
)
```

この関数は、wolfSSL_get_error() を呼び出して SSL_ERROR_WANT_READ が返される場合と同様です。基礎となるエラー状態が SSL_ERROR_WANT_READ の場合、この関数は 1 を返し、それ以外の場合は 0 を返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) で作成された SSL セッションへのポインタ。

See:

- [wolfSSL_want_write](#)
- [wolfSSL_get_error](#)

Return:

- 1 [wolfSSL_get_error\(\)](#) が SSL_ERROR_WANT_READ を返す場合。基礎となる I/O に読み取り可能なデータがあります。

- 0 SSL_ERROR_WANT_READ エラー状態がない場合。

Example

```
int ret;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_want_read(ssl);
if (ret == 1) {
    // 基礎となる I/O に読み取り可能なデータがあります (SSL_ERROR_WANT_READ)
}
```

A.7.2.10 function wolfSSL_want_write

```
int wolfSSL_want_write(
    WOLFSSL * ssl
)
```

この関数は、wolfSSL_get_error() を呼び出して SSL_ERROR_WANT_WRITE が返される場合と同様です。基礎となるエラー状態が SSL_ERROR_WANT_WRITE の場合、この関数は 1 を返し、それ以外の場合は 0 を返します。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。

See:

- wolfSSL_want_read
- wolfSSL_get_error

Return:

- 1 wolfSSL_get_error() が SSL_ERROR_WANT_WRITE を返す場合。基礎となる SSL 接続で進行するために、基礎となる I/O にデータを書き込む必要があります。
- 0 SSL_ERROR_WANT_WRITE エラー状態がない場合。

Example

```
int ret;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_want_write(ssl);
if (ret == 1) {
    // 基礎となる I/O にデータを書き込む必要があります (SSL_ERROR_WANT_WRITE)
}
```

A.7.2.11 function wolfSSL_ERR_peek_last_error

```
unsigned long wolfSSL_ERR_peek_last_error(
    void
)
```

この関数は、WOLFSSL_ERROR から発生した最後のエラーの絶対値を返します。

Parameters:

- **none** パラメータはありません。

See: `wolfSSL_ERR_print_errors_fp`

Return: error 最後のエラーの絶対値を返します。

Example

```
unsigned long err;
...
err = wolfSSL_ERR_peek_last_error();
// err 値を検査
```

A.8 wolfSSL の初期化/シャットダウン

A.8.1 Functions

	Name
int	wolfSSL_shutdown (WOLFSSL * ssl) この関数は、SSL セッション ssl を使用してアクティブな SSL/TLS 接続をシャットダウンします。この関数は、ピアに「close notify」アラートを送信しようと試みます。呼び出し側のアプリケーションは、ピアからの応答として「close notify」アラートが送信されるのを待つか、wolfSSL_shutdown() を直接呼び出した後に基盤となる接続をシャットダウンするか (リソースを節約するため) を選択できます。TLS 仕様では、どちらのオプションも許可されています。基盤となる接続を将来再び使用する場合は、ピア間の同期を維持するために、完全な双方向シャットダウン手順を実行する必要があります。wolfSSL_shutdown() は、ブロッキング I/O とノンブロッキング I/O の両方で動作します。基盤となる I/O がノンブロッキングの場合、基盤となる I/O が wolfSSL_shutdown() の継続に必要な要求を満たせなかった場合、wolfSSL_shutdown() はエラーを返します。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、基盤となる I/O の準備ができたときに、wolfSSL_shutdown() の呼び出しを繰り返す必要があります。
int	wolfSSL_SetServerID (WOLFSSL * ssl, const unsigned char * id, int len, int newSession) この関数は、クライアントセッションをサーバー ID に関連付けます。newSession フラグがオンの場合、既存のセッションは再利用されません。
int	wolfSSL_library_init (void) この関数は wolfSSL_CTX_new() 内で内部的に呼び出されます。この関数は wolfSSL_Init() のラッパーであり、wolfSSL が OpenSSL 互換性レイヤーでコンパイルされた場合の OpenSSL 互換性 (SSL_library_init) のために存在します。wolfSSL_Init() は、より一般的に使用される wolfSSL 初期化関数です。

	Name
int	wolfSSL_get_shutdown (const WOLFSSL * ssl) この関数は Options 構造体の closeNotify、connReset、または sentNotify メンバのシャットダウン条件をチェックします。Options 構造体は WOLFSSL 構造体内にあります。
int	wolfSSL_is_init_finished (const WOLFSSL * ssl) この関数は接続が確立されているかどうかをチェックします。
int	wolfSSL_Init (void) 使用するために wolfSSL ライブラリを初期化します。アプリケーションごとに 1 回、ライブラリへの他の呼び出しの前に呼び出す必要があります。
int	wolfSSL_Cleanup (void)wolfSSL ライブラリをこれ以上使用しないように初期化を解除します。呼び出す必要はありませんが、ライブラリが使用したリソースを解放します。
int	wolfSSL_SetMinVersion (WOLFSSL * ssl, int version) この関数は、許可される最小ダウングレードバージョンを設定します。ダウングレードを許可する接続 (wolfSSLv23_client_method または wolfSSLv23_server_method) を使用する場合にはのみ適用されます。
int	wolfSSL_ALPN_GetProtocol (WOLFSSL * ssl, char ** protocol_name, unsigned short * size) この関数は、サーバーによって設定されたプロトコル名を取得します。
int	wolfSSL_ALPN_GetPeerProtocol (WOLFSSL * ssl, char ** list, unsigned short * listSz) この関数は、SSL オブジェクトから alpn_client_list データをバッファにコピーします。
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) この関数は、cr と sr の値をコピーし、wc_PRF(疑似乱数関数) に渡し、その値を返します。
int	wolfSSL_preferred_group (WOLFSSL * ssl) この関数は、TLS v1.3 ハンドシェイクでクライアントが優先的に使用したい鍵交換グループを返します。ハンドシェイク完了後にこの関数を呼び出して、サーバーが優先するグループを判定し、この情報を将来の接続で使用して鍵交換用の鍵ペアを事前生成できます。
void	**wolfSSL_CTX_set_client_CA_list (WOLFSSL_X509_NAME * names) サーバ側では、この関数は証明書要求でクライアントに送信される CA 名のリストを設定します。これは、サーバーがサポートする CA のヒントとして機能します。

	Name
WOLFSSL_STACK *	wolfSSL_CTX_get_client_CA_list (const WOLFSSL_CTX * ctx) これは、wolfSSL_CTX_set_client_CA_list を介して以前に設定されたリストを取得します。リストが設定されていない場合は NULL を返します。
void	**wolfSSL_set_client_CA_list (WOLFSSL_X509_NAME * names)wolfSSL_CTX_set_client_CA_list と同じですが、セッション固有です。CA リストがコンテキストとセッションの両方に設定されている場合、セッション上のリストが使用されます。
WOLFSSL_STACK *	wolfSSL_get_client_CA_list (const WOLFSSL * ssl) サーバ側では、wolfSSL_set_client_CA_list を介して以前に設定されたリストを取得します。何も設定されていない場合は、wolfSSL_CTX_set_client_CA_list を介して以前に設定されたリストを返します。リストが全く設定されていない場合は、NULL を返します。
void	**wolfSSL_CTX_set0_CA_list (WOLFSSL_X509_NAME * names) この関数は、ピアの認証でサポートされている CA のヒントとしてピアに送信される CA 名のリストを設定します。
WOLFSSL_STACK *	wolfSSL_CTX_get0_CA_list (const WOLFSSL_CTX * ctx) これは、wolfSSL_CTX_set0_CA_list を介して以前に設定されたリストを取得します。リストが設定されていない場合は NULL を返します。
void	**wolfSSL_set0_CA_list (WOLFSSL_X509_NAME * names)wolfSSL_CTX_set0_CA_list と同じですが、セッション固有です。CA リストがコンテキストとセッションの両方に設定されている場合、セッション上のリストが使用されます。
WOLFSSL_STACK *	wolfSSL_get0_CA_list (const WOLFSSL * ssl) これは、wolfSSL_set0_CA_list を介して以前に設定されたリストを取得します。何も設定されていない場合は、wolfSSL_CTX_set0_CA_list を介して以前に設定されたリストを返します。リストが全く設定されていない場合は、NULL を返します。
WOLFSSL_STACK *	wolfSSL_get0_peer_CA_list (const WOLFSSL * ssl) これは、ピアから受信した CA リストを返します。
void	wolfSSL_CTX_set_cert_cb (WOLFSSL_CTX * ctx, int()(WOLFSSL , void) cb, void arg) この関数は、証明書が使用される直前に呼び出されるコールバックを設定し、アプリケーションが証明書を検査、設定、またはクリアできるようにします。例えば、ピアから送信された CA リストに反応することができます。

	Name
int	wolfSSL_get_client_suites_sigalgs (const WOLFSSL * ssl, const byte ** suites, word16 * suiteSz, const byte ** hashSigAlgo, word16 * hashSigAlgoSz) この関数は、クライアントが提供する暗号スイートと署名アルゴリズムの生のリストを返します。リストは、wolfSSL_CTX_set_cert_cb() で設定されたコールバック内でのみ保存され、返されます。これは、利用可能な暗号スイートと署名アルゴリズムに基づいて証明書と鍵を動的にロードできるようにするのに便利です。
WOLFSSL_CIPHERSUITE_INFO	wolfSSL_get_ciphersuite_info (byte first, byte second) これは、生の暗号スイートバイトから直接暗号スイートに関する情報を返します。
int	wolfSSL_get_sigalg_info (byte first, byte second, int * hashAlgo, int * sigAlgo) これは、生の暗号スイートバイトから直接ハッシュおよび署名アルゴリズムに関する情報を返します。

A.8.2 Functions Documentation

A.8.2.1 function wolfSSL_shutdown

```
int wolfSSL_shutdown(
    WOLFSSL * ssl
)
```

この関数は、SSL セッション ssl を使用してアクティブな SSL/TLS 接続をシャットダウンします。この関数は、ピアに「close notify」アラートを送信しようと試みます。呼び出し側のアプリケーションは、ピアからの応答として「close notify」アラートが送信されるのを待つか、wolfSSL_shutdown() を直接呼び出した後に基盤となる接続をシャットダウンするか (リソースを節約するため) を選択できます。TLS 仕様では、どちらのオプションも許可されています。基盤となる接続を将来再び使用する場合は、ピア間の同期を維持するために、完全な双方向シャットダウン手順を実行する必要があります。wolfSSL_shutdown() は、ブロッキング I/O とノンブロッキング I/O の両方で動作します。基盤となる I/O がノンブロッキングの場合、基盤となる I/O が wolfSSL_shutdown() の継続に必要な要求を満たせなかった場合、wolfSSL_shutdown() はエラーを返します。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、基盤となる I/O の準備ができたときに、wolfSSL_shutdown() の呼び出しを繰り返す必要があります。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。

See:

- wolfSSL_free
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS が成功時に返されます。
- SSL_SHUTDOWN_NOT_DONE は、シャットダウンが完了していない場合に返され、関数を再度呼び出す必要があります。
- SSL_FATAL_ERROR は、失敗時に返されます。より具体的なエラーコードについては、wolfSSL_get_error() を呼び出してください。

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_shutdown(ssl);
if (ret != 0) {
    // SSL 接続のシャットダウンに失敗しました
}
```

A.8.2.2 function wolfSSL_SetServerID

```
int wolfSSL_SetServerID(
    WOLFSSL * ssl,
    const unsigned char * id,
    int len,
    int newSession
)
```

この関数は、クライアントセッションをサーバー ID に関連付けます。newSession フラグがオンの場合、既存のセッションは再利用されません。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **id** WOLFSSL_SESSION 構造体の serverID メンバーにコピーされる定数バイトポインタ。
- **len** セッション id パラメータの長さを表す int 型。
- **newSession** セッションを再利用するかどうかを示すフラグを表す int 型。

See: [wolfSSL_set_session](#)

Return:

- SSL_SUCCESS は、関数がエラーなく実行された場合に返されます。
- BAD_FUNC_ARG は、WOLFSSL 構造体または id パラメータが NULL の場合、または len が 0 より大きくない場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol );
WOLFSSL* ssl = WOLFSSL_new(ctx);
const byte id[MAX_SIZE]; // または動的にスペースを作成
int len = 0; // 長さを初期化
int newSession = 0; // 許可するフラグ
...
int ret = wolfSSL_SetServerID(ssl, id, len, newSession);

if (ret == WOLFSSL_SUCCESS) {
    // ID が正常に設定されました
}
```

A.8.2.3 function wolfSSL_library_init

```
int wolfSSL_library_init(
    void
)
```

この関数は wolfSSL_CTX_new() 内で内部的に呼び出されます。この関数は wolfSSL_Init() のラッパーであり、wolfSSL が OpenSSL 互換性レイヤーでコンパイルされた場合の OpenSSL 互換性 (SSL_library_init) のために存在します。wolfSSL_Init() は、より一般的に使用される wolfSSL 初期化関数です。

Parameters:

- **none** パラメータはありません。

See:

- [wolfSSL_Init](#)
- [wolfSSL_Cleanup](#)

Return:

- SSL_SUCCESS 呼び出しが成功した場合に返されます。
- SSL_FATAL_ERROR 失敗時に返されます。

Example

```
int ret = 0;
ret = wolfSSL_library_init();
if (ret != SSL_SUCCESS) {
    // wolfSSL の初期化に失敗しました
}
...
```

A.8.2.4 function wolfSSL_get_shutdown

```
int wolfSSL_get_shutdown(
    const WOLFSSL * ssl
)
```

この関数は Options 構造体の closeNotify、connReset、または sentNotify メンバのシャットダウン条件をチェックします。Options 構造体は WOLFSSL 構造体内にあります。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体への定数ポインタ。

See: wolfSSL_SESSION_free

Return:

- 1 SSL_SENT_SHUTDOWN が返されます。
- 2 SSL_RECEIVED_SHUTDOWN が返されます。

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 {
```

```
// 致命的エラー。
}
```

A.8.2.5 function wolfSSL_is_init_finished

```
int wolfSSL_is_init_finished(
    const WOLFSSL * ssl
)
```

この関数は接続が確立されているかどうかをチェックします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_set_accept_state`
- `wolfSSL_get_keys`
- `wolfSSL_set_shutdown`

Return:

- 0 接続が確立されていない場合に返されます、つまり WOLFSSL 構造体が NULL またはハンドシェイクが完了していない場合。
- 1 接続が確立されている場合に返されます、つまり WOLFSSL ハンドシェイクが完了している場合。

EXAMPLE

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_is_init_finished(ssl)){
    // ハンドシェイクが完了し、接続が確立されています
}
```

A.8.2.6 function wolfSSL_Init

```
int wolfSSL_Init(
    void
)
```

使用するために wolfSSL ライブラリを初期化します。アプリケーションごとに 1 回、ライブラリへの他の呼び出しの前に呼び出す必要があります。

See: `wolfSSL_Cleanup`

Return:

- `SSL_SUCCESS` 成功した場合に返されます。
- `BAD_MUTEX_E` 返される可能性のあるエラーです。
- `WC_INIT_E` wolfCrypt 初期化エラーが返されます。

Example

```
int ret = 0;
ret = wolfSSL_Init();
if (ret != SSL_SUCCESS) {
    wolfSSL ライブラリの初期化に失敗しました
}
```

A.8.2.7 function wolfSSL_Cleanup

```
int wolfSSL_Cleanup(  
    void  
)
```

wolfSSL ライブラリをこれ以上使用しないように初期化を解除します。呼び出す必要はありませんが、ライブラリが使用したリソースを解放します。

See: [wolfSSL_Init](#)

Return:

- SSL_SUCCESS エラーなしで返されます。
- BAD_MUTEX_E mutex エラーが返されます。

Example

```
wolfSSL_Cleanup();
```

A.8.2.8 function wolfSSL_SetMinVersion

```
int wolfSSL_SetMinVersion(  
    WOLFSSL * ssl,  
    int version  
)
```

この関数は、許可される最小ダウングレードバージョンを設定します。ダウングレードを許可する接続 (wolfSSLv23_client_method または wolfSSLv23_server_method) を使用する場合にのみ適用されます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) を使用して作成された WOLFSSL 構造体へのポインタ。
- **version** 最小値として設定されるバージョンの整数表現: WOLFSSL_SSLV3 = 0、WOLFSSL_TLSV1 = 1、WOLFSSL_TLSV1_1 = 2、または WOLFSSL_TLSV1_2 = 3。

See: [SetMinVersionHelper](#)

Return:

- SSL_SUCCESS この関数とそのサブルーチンがエラーなく実行された場合に返されます。
- BAD_FUNC_ARG SSL オブジェクトが NULL の場合に返されます。サブルーチンでは、バージョンマッチが良好でない場合にこのエラーがスローされます。

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(protocol method);  
WOLFSSL* ssl = WOLFSSL_new(ctx);  
int version; // マクロ表現  
...  
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){  
    // 最小バージョンの設定に失敗  
}
```

A.8.2.9 function wolfSSL_ALPN_GetProtocol

```
int wolfSSL_ALPN_GetProtocol(  
    WOLFSSL * ssl,  
    char ** protocol_name,  
    unsigned short * size  
)
```

この関数は、サーバーによって設定されたプロトコル名を取得します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **protocol_name** プロトコル名を表し、ALPN 構造体に保持される char へのポインタ。
- **size** `protocol_name` のサイズを表す word16 型。

See:

- `TLSX_ALPN_GetRequest`
- `TLSX_Find`

Return:

- `SSL_SUCCESS` エラーがスローされずに正常に実行された場合に返されます。
- `SSL_FATAL_ERROR` 拡張が見つからなかった場合、またはピアとのプロトコルマッチがなかった場合に返されます。また、受け入れられたプロトコル名が複数ある場合にもエラーがスローされます。
- `SSL_ALPN_NOT_FOUND` ピアとのプロトコルマッチが見つからなかったことを示して返されます。
- `BAD_FUNC_ARG` 関数に NULL 引数が渡された場合に返されます。

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){
    // ALPN プロトコルを送信
}
```

A.8.2.10 function `wolfSSL_ALPN_GetPeerProtocol`

```
int wolfSSL_ALPN_GetPeerProtocol(
    WOLFSSL * ssl,
    char ** list,
    unsigned short * listSz
)
```

この関数は、SSL オブジェクトから `alpn_client_list` データをバッファにコピーします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **list** バッファへのポインタ。SSL オブジェクトからのデータがここにコピーされます。
- **listSz** バッファサイズ。

See: `wolfSSL_UseALPN`

Return:

- `SSL_SUCCESS` 関数がエラーなく実行された場合に返されます。SSL オブジェクトの `alpn_client_list` メンバが `list` パラメータにコピーされました。
- `BAD_FUNC_ARG` `list` または `listSz` パラメータが NULL の場合に返されます。
- `BUFFER_ERROR` `list` バッファに問題がある場合に返されます (NULL であるか、サイズが 0 の場合)。
- `MEMORY_ERROR` メモリの動的割り当てに問題があった場合に返されます。

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){
    // クライアントが送信したプロトコル名のリスト
}
```

A.8.2.11 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
)
```

この関数は、cr と sr の値をコピーし、wc_PRF(疑似乱数関数) に渡し、その値を返します。

Parameters:

- **ms** Arrays 構造体に保持されているマスターシークレット。
- **msLen** マスターシークレットの長さ。
- **pms** Arrays 構造体に保持されているプリマスターシークレット。
- **pmsLen** プリマスターシークレットの長さ。
- **cr** クライアントランダム。
- **sr** サーバランダム。
- **tls1_2** バージョンが少なくとも TLS バージョン 1.2 であることを示します。
- **hash_type** ハッシュタイプを示します。

See:

- wc_PRF
- MakeTlsMasterSecret

Return:

- 0 成功時。
- BUFFER_E バッファのサイズでエラーが発生する場合に返されます。
- MEMORY_E サブルーチンが動的メモリの割り当てに失敗した場合に返されます。

Example

```
WOLFSSL* ssl;
```

MakeTlsMasterSecret で呼び出され、以下のように必要な情報を取得します:

```
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;
}

```

A.8.2.12 function wolfSSL_preferred_group

```

int wolfSSL_preferred_group(
    WOLFSSL * ssl
)

```

この関数は、TLS v1.3 ハンドシェイクでクライアントが優先的に使用したい鍵交換グループを返します。ハンドシェイク完了後にこの関数を呼び出して、サーバが優先するグループを判定し、この情報を将来の接続で使用して鍵交換用の鍵ペアを事前生成できます。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_UseKeyShare`
- `wolfSSL_CTX_set_groups`
- `wolfSSL_set_groups`
- `wolfSSL_CTX_set1_groups_list`
- `wolfSSL_set1_groups_list`

Return:

- `BAD_FUNC_ARG` ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- `SIDE_ERROR` サーバで呼び出された場合。
- `NOT_READY_ERROR` ハンドシェイク完了前に呼び出された場合。
- Group identifier 成功時、グループ識別子。

Example

```

int ret;
int group;
WOLFSSL* ssl;
...
ret = wolfSSL_CTX_set1_groups_list(ssl)
if (ret < 0) {
    // グループの取得に失敗しました
}
group = ret;

```

A.8.2.13 function wolfSSL_CTX_set_client_CA_list

```

void wolfSSL_CTX_set_client_CA_list(
    WOLFSSL_CTX * ctx,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)

```

サーバ側では、この関数は証明書要求でクライアントに送信される CA 名のリストを設定します。これは、サーバがサポートする CA のヒントとして機能します。

Parameters:

- **ctx** wolfSSL コンテキストへのポインタ。
- **names** 設定される名前のリスト。

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](#)

クライアント側では、この関数は効果がありません。

A.8.2.14 function wolfSSL_CTX_get_client_CA_list

```
WOLFSSL_STACK * wolfSSL_CTX_get_client_CA_list(  
    const WOLFSSL_CTX * ctx  
)
```

これは、wolfSSL_CTX_set_client_CA_list を介して以前に設定されたリストを取得します。リストが設定されていない場合は NULL を返します。

Parameters:

- **ctx** wolfSSL コンテキストへのポインタ。

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: CA 名を含む WOLFSSL_X509_NAME のスタック。

A.8.2.15 function wolfSSL_set_client_CA_list

```
void wolfSSL_set_client_CA_list(  
    WOLFSSL * ssl,  
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names  
)
```

wolfSSL_CTX_set_client_CA_list と同じですが、セッション固有です。CA リストがコンテキストとセッションの両方に設定されている場合、セッション上のリストが使用されます。

Parameters:

- **ssl** WOLFSSL オブジェクトへのポインタ。
- **names** 設定する名前のリスト。

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`

A.8.2.16 function `wolfSSL_get_client_CA_list`

```
WOLFSSL_STACK * wolfSSL_get_client_CA_list(
    const WOLFSSL * ssl
)
```

サーバ側では、`wolfSSL_set_client_CA_list` を介して以前に設定されたリストを取得します。何も設定されていない場合は、`wolfSSL_CTX_set_client_CA_list` を介して以前に設定されたリストを返します。リストが全く設定されていない場合は、NULL を返します。

Parameters:

- **ssl** WOLFSSL オブジェクトへのポインタ。

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: CA 名を含む WOLFSSL_X509_NAME のスタック。

クライアント側では、サーバから受信したリストを取得します。何も受信していない場合は NULL を返します。`wolfSSL_CTX_set_cert_cb` を使用して、サーバから証明書要求を受信したときに証明書を動的にロードするコールバックを登録できます。

A.8.2.17 function `wolfSSL_CTX_set0_CA_list`

```
void wolfSSL_CTX_set0_CA_list(
    WOLFSSL_CTX * ctx,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

この関数は、ピアの認証でサポートされている CA のヒントとしてピアに送信される CA 名のリストを設定します。

Parameters:

- **ctx** wolfSSL コンテキストへのポインタ。
- **names** 設定する名前のリスト。

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`

TLS >= 1.3 では、これはクライアントとサーバ間の両方向でサポートされています。サーバ側では、CA 名は `CertificateRequest` の一部として送信されるため、この関数は `*_set_client_CA_list` と同等です。クライアント側では、これらは `ClientHello` の一部として送信されます。

TLS < 1.3 では、クライアントからサーバへの CA 名の送信はサポートされていないため、この関数は `wolfSSL_CTX_set_client_CA_list` と同等です。

`_set_client_CA_list` と `*_set0_CA_list` を介して設定されたリストは内部的に別々であることに注意してください。つまり、`*_get_client_CA_list` を呼び出しても `*_set0_CA_list` を介して設定されたリストは取得されず、その逆も同様です。両方が設定されている場合、サーバはクライアントに CA 名を送信する際に `*_set0_CA_list` を無視します。

A.8.2.18 function `wolfSSL_CTX_get0_CA_list`

```
WOLFSSL_STACK * wolfSSL_CTX_get0_CA_list(
    const WOLFSSL_CTX * ctx
)
```

これは、`wolfSSL_CTX_set0_CA_list` を介して以前に設定されたリストを取得します。リストが設定されていない場合は `NULL` を返します。

Parameters:

- **ctx** wolfSSL コンテキストへのポインタ。

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: CA 名を含む `WOLFSSL_X509_NAME` のスタック。

A.8.2.19 function `wolfSSL_set0_CA_list`

```
void wolfSSL_set0_CA_list(
    WOLFSSL * ssl,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

`wolfSSL_CTX_set0_CA_list` と同じですが、セッション固有です。CA リストがコンテキストとセッションの両方に設定されている場合、セッション上のリストが使用されます。

Parameters:

- **ssl** WOLFSSL オブジェクトへのポインタ。
- **names** 設定する名前のリスト。

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`

A.8.2.20 function `wolfSSL_get0_CA_list`

```
WOLFSSL_STACK * wolfSSL_get0_CA_list(  
    const WOLFSSL * ssl  
)
```

これは、`wolfSSL_set0_CA_list` を介して以前に設定されたリストを取得します。何も設定されていない場合は、`wolfSSL_CTX_set0_CA_list` を介して以前に設定されたリストを返します。リストが全く設定されていない場合は、NULL を返します。

Parameters:

- **ssl** WOLFSSL オブジェクトへのポインタ。

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: CA 名を含む WOLFSSL_X509_NAME のスタック。

A.8.2.21 function `wolfSSL_get0_peer_CA_list`

```
WOLFSSL_STACK * wolfSSL_get0_peer_CA_list(  
    const WOLFSSL * ssl  
)
```

これは、ピアから受信した CA リストを返します。

Parameters:

- **ssl** WOLFSSL オブジェクトへのポインタ。

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: CA 名を含む WOLFSSL_X509_NAME のスタック。

クライアント側では、これはサーバが CertificateRequest で送信したリストであり、この関数は wolfSSL_get_client_CA_list と同等です。

サーバ側では、これは TLS >= 1.3 でクライアントが ClientHello メッセージで送信したリストです。TLS < 1.3 では、この関数はサーバ側で常に NULL を返します。

wolfSSL_CTX_set_cert_cb を使用して、ピアから CA リストを受信したときに証明書を動的にロードするコールバックを登録できます。

A.8.2.22 function wolfSSL_CTX_set_cert_cb

```
void wolfSSL_CTX_set_cert_cb(  
    WOLFSSL_CTX * ctx,  
    int (*)(WOLFSSL *, void *) cb,  
    void * arg  
)
```

この関数は、証明書が使用される直前に呼び出されるコールバックを設定し、アプリケーションが証明書を検査、設定、またはクリアできるようにします。例えば、ピアから送信された CA リストに反応することができます。

Parameters:

- **ctx** wolfSSL コンテキストへのポインタ。
- **cb** コールバックへの関数ポインタ。
- **arg** コールバックに渡されるポインタ。

See:

- [wolfSSL_get0_peer_CA_list](#)
- [wolfSSL_get_client_CA_list](#)

A.8.2.23 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  
)
```

この関数は、クライアントが提供する暗号スイートと署名アルゴリズムの生のリストを返します。リストは、wolfSSL_CTX_set_cert_cb() で設定されたコールバック内でのみ保存され、返されます。これは、利用可能な暗号スイートと署名アルゴリズムに基づいて証明書と鍵を動的にロードできるようにするのに便利です。

Parameters:

- **ssl** リストを抽出する WOLFSSL オブジェクト。
- **suites** クライアント暗号スイートの生のフィルタリングされていないリスト。利用可能なスイートがない場合は NULL になることがあります。
- **suiteSz** suites のサイズ (バイト単位)。
- **hashSigAlgo** クライアント署名アルゴリズムの生のフィルタリングされていないリスト。提供されない場合は NULL になることがあります。
- **hashSigAlgoSz** hashSigAlgo のサイズ (バイト単位)。

See:

- [wolfSSL_get_ciphersuite_info](#)
- [wolfSSL_get_sigalg_info](#)

Return:

- WOLFSSL_SUCCESS スイートが利用可能な場合。
- WOLFSSL_FAILURE スイートが利用できない場合。

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);

    // 暗号スイートと sigalgs に基づいてロードする証明書を選択
}

WOLFSSL* ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
wolfSSL_CTX_set_cert_cb(ctx, certCB, NULL);
```

A.8.2.24 function wolfSSL_get_ciphersuite_info

```
WOLFSSL_CIPHERSUITE_INFO wolfSSL_get_ciphersuite_info(
    byte first,
    byte second
)
```

これは、生の暗号スイートバイトから直接暗号スイートに関する情報を返します。

Parameters:

- **first** 暗号スイートの最初のバイト。
- **second** 暗号スイートの 2 番目のバイト。

See:

- [wolfSSL_get_client_suites_sigalgs](#)
- [wolfSSL_get_sigalg_info](#)

Return: WOLFSSL_CIPHERSUITE_INFO 暗号スイートで使用される認証のタイプに関する情報を含む構造体。

Example

```
WOLFSSL_CIPHERSUITE_INFO info =
    wolfSSL_get_ciphersuite_info(suites[0], suites[1]);
if (info.rsaAuth)
    haveRSA = 1;
else if (info.eccAuth)
    haveECC = 1;
```

A.8.2.25 function wolfSSL_get_sigalg_info

```
int wolfSSL_get_sigalg_info(  
    byte first,  
    byte second,  
    int * hashAlgo,  
    int * sigAlgo  
)
```

これは、生の暗号スイートバイトから直接ハッシュおよび署名アルゴリズムに関する情報を返します。

Parameters:

- **first** ハッシュおよび署名アルゴリズムの最初のバイト。
- **second** ハッシュおよび署名アルゴリズムの 2 番目のバイト。
- **hashAlgo** MAC アルゴリズムの enum wc_HashType。
- **sigAlgo** 認証アルゴリズムの enum Key_Sum。

See:

- [wolfSSL_get_client_suites_sigalgs](#)
- [wolfSSL_get_ciphersuite_info](#)

Return:

- 0 情報が正しく設定された場合。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合、またはバイトが認識される sigalg スイートでない場合。

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;
```

B wolfCrypt API リファレンス

B.1 ASN.1

B.1.1 Functions

	Name
int	wc_InitCert (Cert * cert) この関数は、デフォルトのオプションでデフォルトの証明書を初期化します：version = 3 (0x2)、serial = 0、sigType = SHA_WITH_RSA、issuer = 空白、daysValid = 500、selfSigned = 1 (true) 発行者として subject を使用、subject = 空白
Cert *	wc_CertNew (void * heap) この関数は、証明書操作中に使用するための新しい Cert 構造体を割り当てます。アプリケーションが構造体自体を割り当てる必要はありません。Cert 構造体もこの関数によって初期化されるため、wc_InitCert() を呼び出す必要がなくなります。アプリケーションが割り当てられた Cert 構造体の使用を終了したら、wc_CertFree() を呼び出す必要があります。
void	wc_CertFree (Cert * cert) この関数は、wc_CertNew() への以前の呼び出しによって cert 構造体に割り当てられたメモリを解放します。
int	wc_MakeCert (Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng) CA 署名付き証明書を作成するために使用されます。subject 情報が入力された後に呼び出されます。この関数は、cert 入力から x509 証明書 v3 RSA または ECC を作成します。次に、この証明書を derBuffer に書き込みます。証明書を生成するために、rsaKey または eccKey のいずれかを受け取ります。このメソッドを呼び出す前に、証明書を wc_InitCert で初期化する必要があります。
int	wc_MakeCertReq (Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey) この関数は、入力証明書を使用して証明書署名要求を作成し、出力を derBuffer に書き込みます。証明書要求を生成するために、rsaKey または eccKey のいずれかを受け取ります。証明書署名要求に署名するには、この関数の後に wc_SignCert() を呼び出す必要があります。この関数の使用例については、wolfCrypt テストアプリケーション (./wolfcrypt/test/test.c) を参照してください。

	Name
int	wc_SignCert (int requestSz, int sigType, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng) この関数は buffer に署名し、署名を buffer の末尾に追加します。署名タイプを受け取ります。CA 署名付き証明書を作成する場合は、wc_MakeCert() または wc_MakeCertReq() の後に呼び出す必要があります。
int	wc_MakeSelfCert (Cert * cert, byte * derBuffer, word32 derSz, RsaKey * key, WC_RNG * rng) この関数は、自己署名用の前の 2 つの関数、wc_MakeCert と wc_SignCert の組み合わせです (前の関数は CA 要求に使用できます)。証明書を生成してから署名し、自己署名証明書を生成します。
int	wc_SetIssuer (Cert * cert, const char * issuerFile) この関数は、証明書の発行者を、提供された pem issuerFile の発行者に設定します。また、証明書の自己署名属性を false に変更します。issuerFile で指定された発行者は、cert 発行者を設定する前に検証されます。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetSubject (Cert * cert, const char * subjectFile) この関数は、証明書の subject を、提供された pem subjectFile の subject に設定します。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetSubjectRaw (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの subject から証明書の生の subject を設定します。このメソッドは、署名前に生の subject フィールドを設定するために使用されます。
int	wc_GetSubjectRaw (byte ** subjectRaw, Cert * cert) この関数は、証明書構造体から生の subject を取得します。
int	wc_SetAltNames (Cert * cert, const char * file) この関数は、証明書の代替名を、提供された pem ファイル内の代替名に設定します。これは、同じ証明書で複数のドメインを保護したい場合に便利です。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetIssuerBuffer (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの発行者から証明書の発行者を設定します。また、証明書の自己署名属性を false に変更します。このメソッドは、署名前にフィールドを設定するために使用されます。

	Name
int	wc_SetIssuerRaw (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの発行者から証明書の生の発行者を設定します。このメソッドは、署名前に生の発行者フィールドを設定するために使用されます。
int	wc_SetSubjectBuffer (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの subject から証明書の subject を設定します。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetAltNamesBuffer (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの代替名から証明書の代替名を設定します。これは、同じ証明書で複数のドメインを保護したい場合に便利です。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetDatesBuffer (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの日付範囲から証明書の日付を設定します。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetAuthKeyIdFromPublicKey (Cert * cert, RsaKey * rsaKey, ecc_key * eckey)RSA または ECC 公開鍵から AKID を設定します。注: rsaKey または eckey のいずれか一方のみを設定し、両方は設定しないでください。
int	wc_SetAuthKeyIdFromCert (Cert * cert, const byte * der, int derSz)DER エンコードされた証明書から AKID を設定します。
int	wc_SetAuthKeyId (Cert * cert, const char * file)PEM 形式の証明書ファイルから AKID を設定します。
int	wc_SetSubjectKeyIdFromPublicKey (Cert * cert, RsaKey * rsaKey, ecc_key * eckey)RSA または ECC 公開鍵から SKID を設定します。
int	wc_SetSubjectKeyId (Cert * cert, const char * file)PEM 形式の公開鍵ファイルから SKID を設定します。両方の引数が必要です。
int	wc_PemPubKeyToDer (const char * fileName, unsigned char * derBuf, int derSz) ファイルから PEM キーを読み込み、DER エンコードされたバッファに変換します。
int	wc_PubKeyPemToDer (const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz)PEM エンコードされた公開鍵を DER に変換します。バッファに書き込まれたバイト数、またはエラーの場合は負の値を返します。
int	wc_PemCertToDer (const char * fileName, unsigned char * derBuf, int derSz) この関数は、pem 証明書を der 証明書に変換し、結果の証明書を提供された derBuf バッファに配置します。

	Name
int	wc_DerToPem (const byte * der, word32 derSz, byte * output, word32 outSz, int type) この関数は、der バッファに含まれる der 形式の入力証明書を、output バッファに含まれる pem 形式の出力証明書に変換します。これはインプレース変換ではなく、pem 形式の出力を格納するために別のバッファを使用する必要があることに注意してください。
int	wc_DerToPemEx (const byte * der, word32 derSz, byte * output, word32 outSz, byte * cipher_info, int type) この関数は、der 形式の入力証明書を変換します、der バッファに含まれる、pem 形式の出力証明書に変換し、output バッファに格納します。これはインプレース変換ではなく、pem 形式の出力を格納するために別のバッファを使用する必要があることに注意してください。暗号情報の設定を許可します。
int	wc_EccPrivateKeyDecode (const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz) この関数は、入力バッファ input から ECC 秘密鍵を読み取り、秘密鍵を解析し、それを使用して ecc_key オブジェクトを生成し、key に格納します。
int	wc_EccKeyToDer (ecc_key * key, byte * output, word32 inLen) この関数は、秘密 ECC キーを der 形式に書き込みます。
int	wc_EccPublicKeyDecode (const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz) 入力バッファから ECC 公開鍵をデコードします。ECC キーを取得するために ASN シーケンスを解析します。
int	wc_EccPublicKeyToDer (ecc_key * key, byte * output, word32 inLen, int with_AlgCurve) この関数は、ECC 公開鍵を DER 形式に変換します。使用されたバッファのサイズを返します。DER 形式の ECC 公開鍵は出力バッファに格納されます。with_AlgCurve フラグは、アルゴリズムと曲線情報を持つヘッダーを含めます
int	wc_EccPublicKeyToDer_ex (ecc_key * key, byte * output, word32 inLen, int with_AlgCurve, int comp) この関数は、ECC 公開鍵を DER 形式に変換します。使用されたバッファのサイズを返します。DER 形式の ECC 公開鍵は出力バッファに格納されます。with_AlgCurve フラグは、アルゴリズムと曲線情報を持つヘッダーを含めます。comp パラメータは、公開鍵を圧縮形式でエクスポートするかどうかを決定します。
int	wc_Curve25519PrivateKeyDecode (const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz) この関数は、DER エンコードされたバッファから Curve25519 秘密鍵 (のみ) をデコードします

	Name
int	wc_Curve25519PublicKeyDecode (const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz) この関数は、DER エンコードされたバッファから Curve25519 公開鍵 (のみ) をデコードします。
int	wc_Curve25519KeyDecode (const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz) この関数は、DER エンコードされたバッファから Curve25519 キーをデコードします。秘密鍵、公開鍵、または両方をデコードできます。
int	wc_Curve25519PrivateKeyToDer (curve25519_key * key, byte * output, word32 inLen) この関数は、Curve25519 秘密鍵を DER 形式にエンコードします。入力キー構造体に公開鍵が含まれている場合、それは無視されます。
int	wc_Curve25519PublicKeyToDer (curve25519_key * key, byte * output, word32 inLen, int withAlg) この関数は、Curve25519 公開鍵を DER 形式にエンコードします。入力キー構造体に秘密鍵が含まれている場合、それは無視されます。
int	wc_Curve25519KeyToDer (curve25519_key * key, byte * output, word32 inLen, int withAlg) この関数は、Curve25519 キーを DER 形式にエンコードします。秘密鍵、公開鍵、または両方をエンコードできます。
word32	wc_EncodeSignature (byte * out, const byte * digest, word32 digSz, int hashOID) この関数は、デジタル署名を出力バッファにエンコードし、作成されたエンコードされた署名のサイズを返します。
int	wc_GetCTC_HashOID (int type) この関数は、ハッシュタイプに対応するハッシュ OID を返します。例えば、WC_SHA512 タイプが与えられた場合、この関数は SHA512 ハッシュに対応する識別子 SHA512h を返します。
void	wc_SetCert_Free (Cert * cert) この関数は、証明書構造体のデコードされた証明書キャッシュによって使用されるメモリとリソースをクリーンアップします。WOLFSSL_CERT_GEN_CACHE が定義されている場合、デコードされた証明書構造体は証明書構造体にキャッシュされます。これにより、証明書設定関数への後続の呼び出しで、各呼び出しでデコードされた証明書を解析することを回避できます。
int	wc_GetPkcs8TraditionalOffset (byte * input, word32 * inOutIdx, word32 sz) この関数は、PKCS#8 暗号化されていないバッファ内の従来の秘密鍵の先頭を見つけます。

	Name
int	wc_CreatePKCS8Key (byte * out, word32 * outSz, byte * key, word32 keySz, int algoID, const byte * curveOID, word32 oidSz) この関数は、DER 秘密鍵を受け取り、PKCS#8 形式に変換します。PKCS#12 縮小キーバッグの作成にも使用されます。RFC 5208 を参照してください。
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) この関数は、暗号化されていない PKCS#8 DER キー（例：wc_CreatePKCS8Key によって作成されたもの）を受け取り、PKCS#8 暗号化形式に変換します。結果の暗号化されたキーは、wc_DecryptPKCS8Key を使用して復号できます。RFC 5208 を参照してください。
int	wc_DecryptPKCS8Key (byte * input, word32 sz, const char * password, int passwordSz) この関数は、暗号化された PKCS#8 DER キーを受け取り、PKCS#8 暗号化されていない DER に復号します。wc_EncryptPKCS8Key によって行われた暗号化を元に戻します。RFC5208 を参照してください。入力バッファは復号されたデータで上書きされます。
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) この関数は、従来の DER キーを受け取り、PKCS#8 形式に変換し、暗号化します。これを行うために wc_CreatePKCS8Key と wc_EncryptPKCS8Key を使用します。
void	wc_InitDecodedCert (struct DecodedCert * cert, const byte * source, word32 inSz, void * heap) この関数は、“cert” パラメータが指す DecodedCert を初期化します。長さ “inSz” の DER エンコードされた証明書への “source” ポインタを保存します。この証明書は、wc_ParseCert への後続の呼び出しによって解析できます。
int	wc_ParseCert (DecodedCert * cert, int type, int verify, void * cm) この関数は、DecodedCert オブジェクトに保存された DER エンコードされた証明書を解析し、そのオブジェクトのフィールドを設定します。DecodedCert は、wc_InitDecodedCert への事前の呼び出しで初期化されている必要があります。この関数は、CertificateManager オブジェクトへのオプションのポインタを取り、CA が CertificateManager で見つかった場合、DecodedCert の証明機関情報を設定するために使用されます。

	Name
void	wc_FreeDecodedCert (struct DecodedCert * cert) この関数は、wc_InitDecodedCert で以前に初期化された DecodedCert を解放します。
int	wc_SetTimeCb (wc_time_cb f) この関数は、wolfSSL が現在時刻を取得する必要があるときに使用される時刻コールバックを登録します。コールバックのプロトタイプは、C 標準ライブラリの“time” 関数と同じである必要があります。
time_t	wc_Time (time_t * t) この関数は、現在時刻を取得します。デフォルトでは、プラットフォーム間で異なる XTIME マクロを使用します。ユーザーは、wc_SetTimeCb 関数を介して任意の関数を使用できます。
int	wc_SetCustomExtension (Cert * cert, int critical, const char * oid, const byte * der, word32 derSz) この関数は、X.509 証明書にカスタム拡張を挿入します。注：ポインタであるパラメータのいずれかが指すアドレスのコンテンツは、証明書が生成されて der 出力が得られるまで変更してはいけません。この関数はコンテンツを別のバッファにコピーしません。
int	wc_SetUnknownExtCallback (DecodedCert * cert, wc_UnknownExtCallback cb) この関数は、wolfSSL が証明書の解析中に証明書内の不明な X.509 拡張に遭遇したときに使用されるコールバックを登録します。コールバックのプロトタイプは次のようにする必要があります：
int	wc_CheckCertSigPubKey (const byte * cert, word32 certSz, void * heap, const byte * pubKey, word32 pubKeySz, int pubKeyOID) この関数は、X.509 証明書の der 形式の署名を公開鍵に対して検証します。公開鍵は、der 形式の完全なサブジェクト公開鍵情報であることが期待されます。
int	wc_Asn1PrintOptions_Init (Asn1PrintOptions * opts) この関数は、ASN.1 プリントオプションを初期化します。
int	wc_Asn1PrintOptions_Set (Asn1PrintOptions * opts, enum Asn1PrintOpt opt, word32 val) この関数は、ASN.1 プリントオプションオブジェクトにプリントオプションを設定します。
int	wc_Asn1_Init (Asn1 * asn1) この関数は、ASN.1 解析オブジェクトを初期化します。
int	wc_Asn1_SetFile (Asn1 * asn1, XFILE file) この関数は、ASN.1 解析オブジェクトへのプリント時に使用するファイルを設定します。
int	wc_Asn1_PrintAll (Asn1 * asn1, Asn1PrintOptions * opts, unsigned char * data, word32 len) すべての ASN.1 項目をプリントします。

B.1.2 Functions Documentation

B.1.2.1 function wc_InitCert

```
int wc_InitCert(  
    Cert * cert  
)
```

この関数は、デフォルトのオプションでデフォルトの証明書を初期化します：version = 3 (0x2)、serial = 0、sigType = SHA_WITH_RSA、issuer = 空白、daysValid = 500、selfSigned = 1 (true) 発行者として subject を使用、subject = 空白

Parameters:

- **cert** 初期化する未初期化の cert 構造体へのポインタ

See:

- [wc_MakeCert](#)
- [wc_MakeCertReq](#)

Return: none 返り値なし。

Example

```
Cert myCert;  
wc_InitCert(&myCert);
```

B.1.2.2 function wc_CertNew

```
Cert * wc_CertNew(  
    void * heap  
)
```

この関数は、証明書操作中に使用するための新しい Cert 構造体を割り当てます。アプリケーションが構造体自体を割り当てる必要はありません。Cert 構造体もこの関数によって初期化されるため、wc_InitCert() を呼び出す必要がなくなります。アプリケーションが割り当てられた Cert 構造体の使用を終了したら、wc_CertFree() を呼び出す必要があります。

Parameters:

- **A** 動的割り当てに使用されるヒープへのポインタ。NULL でも可。

See:

- [wc_InitCert](#)
- [wc_MakeCert](#)
- [wc_CertFree](#)

Return:

- pointer 成功した場合、呼び出しは新しく割り当てられ初期化された Cert へのポインタを返します。
- NULL メモリ割り当て失敗時。

Example

```
Cert* myCert;  
  
myCert = wc_CertNew(NULL);  
if (myCert == NULL) {  
    // Cert 作成失敗  
}
```


B.1.2.3 function wc_CertFree

```
void wc_CertFree(  
    Cert * cert  
)
```

この関数は、wc_CertNew() への以前の呼び出しによって cert 構造体に割り当てられたメモリを解放します。

Parameters:

- **A** 解放する cert 構造体へのポインタ。

See:

- [wc_InitCert](#)
- [wc_MakeCert](#)
- [wc_CertNew](#)

Return: None.

Example

```
Cert*    myCert;  
  
myCert = wc_CertNew(NULL);  
  
// 証明書操作を実行。  
  
wc_CertFree(myCert);
```

B.1.2.4 function wc_MakeCert

```
int wc_MakeCert(  
    Cert * cert,  
    byte * derBuffer,  
    word32 derSz,  
    RsaKey * rsaKey,  
    ecc_key * eccKey,  
    WC_RNG * rng  
)
```

CA 署名付き証明書を作成するために使用されます。subject 情報が入力された後に呼び出されます。この関数は、cert 入力から x509 証明書 v3 RSA または ECC を作成します。次に、この証明書を derBuffer に書き込みます。証明書を生成するために、rsaKey または eccKey のいずれかを受け取ります。このメソッドを呼び出す前に、証明書を wc_InitCert で初期化する必要があります。

Parameters:

- **cert** 初期化された cert 構造体へのポインタ
- **derBuffer** 生成された証明書を保持するバッファへのポインタ
- **derSz** 証明書を格納するバッファのサイズ
- **rsaKey** 証明書の生成に使用される rsa キーを含む RsaKey 構造体へのポインタ
- **eccKey** 証明書の生成に使用される ecc キーを含む EccKey 構造体へのポインタ
- **rng** 証明書を生成するために使用される乱数生成器へのポインタ

See:

- [wc_InitCert](#)
- [wc_MakeCertReq](#)

Return:

- Success 指定された入力 cert から x509 証明書を正常に作成すると、生成された証明書のサイズを返します。
- MEMORY_E XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 提供された derBuffer が生成された証明書を格納するには小さすぎる場合に返されます
- Others 証明書の生成が成功しない場合、追加のエラーメッセージが返される可能性があります。

Example

```
Cert myCert;
wc_InitCert(&myCert);
WC_RNG rng;
// rng を初期化;
RsaKey key;
// key を初期化;
byte * derCert = malloc(FOURK_BUF);
word32 certSz;
certSz = wc_MakeCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

B.1.2.5 function wc_MakeCertReq

```
int wc_MakeCertReq(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey
)
```

この関数は、入力証明書を使用して証明書署名要求を作成し、出力を derBuffer に書き込みます。証明書要求を生成するために、rsaKey または eccKey のいずれかを受け取ります。証明書署名要求に署名するには、この関数の後に wc_SignCert() を呼び出す必要があります。この関数の使用例については、wolfCrypt テストアプリケーション (./wolfcrypt/test/test.c) を参照してください。

Parameters:

- **cert** 初期化された cert 構造体へのポインタ
- **derBuffer** 生成された証明書要求を保持するバッファへのポインタ
- **derSz** 証明書要求を格納するバッファのサイズ
- **rsaKey** 証明書要求の生成に使用される rsa キーを含む RsaKey 構造体へのポインタ
- **eccKey** 証明書要求の生成に使用される ecc キーを含む EccKey 構造体へのポインタ

See:

- [wc_InitCert](#)
- [wc_MakeCert](#)

Return:

- Success 指定された入力 cert から X.509 証明書要求を正常に作成すると、生成された証明書要求のサイズを返します。
- MEMORY_E XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 提供された derBuffer が生成された証明書を格納するには小さすぎる場合に返されます
- Other 証明書要求の生成が成功しない場合、追加のエラーメッセージが返される可能性があります。

Example

```
Cert myCert;
// myCert を初期化
EccKey key;
// key を初期化;
```



```
byte* derCert = (byte*)malloc(FOURK_BUF);

word32 certSz;
certSz = wc_MakeCertReq(&myCert, derCert, FOURK_BUF, NULL, &key);
```

B.1.2.6 function wc_SignCert

```
int wc_SignCert(
    int requestSz,
    int sigType,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

この関数は buffer に署名し、署名を buffer の末尾に追加します。署名タイプを受け取ります。CA 署名付き証明書を作成する場合は、wc_MakeCert() または wc_MakeCertReq() の後に呼び出す必要があります。

Parameters:

- **requestSz** 署名を要求している証明書本文のサイズ
- **sigType** 作成する署名のタイプ。有効なオプションは：CTC_MD5wRSA、CTC_SHAwRSA、CTC_SHAwECDSA、CTC_SHA256wECDSA、および CTC_SHA256wRSA
- **buffer** 署名される証明書を含むバッファへのポインタ。成功時：新しく署名された証明書を保持します
- **buffSz** 新しく署名された証明書を格納するバッファの（合計）サイズ
- **rsaKey** 証明書に署名するために使用される rsa キーを含む RsaKey 構造体へのポインタ
- **eccKey** 証明書に署名するために使用される ecc キーを含む EccKey 構造体へのポインタ
- **rng** 証明書に署名するために使用される乱数生成器へのポインタ

See:

- [wc_InitCert](#)
- [wc_MakeCert](#)

Return:

- Success 証明書の署名に成功すると、証明書の新しいサイズ（署名を含む）を返します。
- MEMORY_E XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 提供されたバッファが生成された証明書を格納するには小さすぎる場合に返されます
- Other 証明書の生成が成功しない場合、追加のエラーメッセージが返される可能性があります。

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// myCert、derCert を初期化
RsaKey key;
// key を初期化;
WC_RNG rng;
// rng を初期化

word32 certSz;
certSz = wc_SignCert(myCert.bodySz, myCert.sigType, derCert, FOURK_BUF,
    &key, NULL,
    &rng);
```

B.1.2.7 function wc_MakeSelfCert

```
int wc_MakeSelfCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * key,
    WC_RNG * rng
)
```

この関数は、自己署名用の前の 2 つの関数、wc_MakeCert と wc_SignCert の組み合わせです（前の関数は CA 要求に使用できます）。証明書を作成してから署名し、自己署名証明書を生成します。

Parameters:

- **cert** 作成して署名する証明書へのポインタ
- **buffer** 署名された証明書を保持するバッファへのポインタ
- **buffSz** 署名された証明書を格納するバッファのサイズ
- **key** 証明書に署名するために使用される rsa キーを含む RsaKey 構造体へのポインタ
- **rng** 証明書の生成と署名に使用される乱数生成器へのポインタ

See:

- [wc_InitCert](#)
- [wc_MakeCert](#)
- [wc_SignCert](#)

Return:

- Success 証明書の署名に成功すると、証明書の新しいサイズを返します。
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 提供されたバッファが生成された証明書を格納するには小さすぎる場合に返されます
- Other 証明書の生成が成功しない場合、追加のエラーメッセージが返される可能性があります。

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// myCert、derCert を初期化
RsaKey key;
// key を初期化;
WC_RNG rng;
// rng を初期化

word32 certSz;
certSz = wc_MakeSelfCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

B.1.2.8 function wc_SetIssuer

```
int wc_SetIssuer(
    Cert * cert,
    const char * issuerFile
)
```

この関数は、証明書の発行者を、提供された pem issuerFile の発行者に設定します。また、証明書の自己署名属性を false に変更します。issuerFile で指定された発行者は、cert 発行者を設定する前に検証されます。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 発行者を設定する証明書へのポインタ

- **issuerFile** pem 形式の証明書を含むファイルのパス

See:

- [wc_InitCert](#)
- [wc_SetSubject](#)
- [wc_SetIssuerBuffer](#)

Return:

- 0 証明書の発行者を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
if(wc_SetIssuer(&myCert, "../path/to/ca-cert.pem") != 0) {
    // 発行者設定エラー
}
```

B.1.2.9 function wc_SetSubject

```
int wc_SetSubject(
    Cert * cert,
    const char * subjectFile
)
```

この関数は、証明書の subject を、提供された pem subjectFile の subject に設定します。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 発行者を設定する証明書へのポインタ
- **subjectFile** pem 形式の証明書を含むファイルのパス

See:

- [wc_InitCert](#)
- [wc_SetIssuer](#)

Return:

- 0 証明書の発行者を正常に設定した場合に返されます

- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
    // subject 設定エラー
}
```

B.1.2.10 function wc_SetSubjectRaw

```
int wc_SetSubjectRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は、提供された der バッファの subject から証明書の生の subject を設定します。このメソッドは、署名前に生の subject フィールドを設定するために使用されます。

Parameters:

- **cert** 生の subject を設定する証明書へのポインタ
- **der** subject を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** subject を取得する der 形式の証明書を含むバッファのサイズ

See:

- [wc_InitCert](#)
- [wc_SetSubject](#)

Return:

- 0 証明書の subject を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます

- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);
// der を初期化
if(wc_SetSubjectRaw(&myCert, der, FOURK_BUF) != 0) {
    // subject 設定エラー
}
```

B.1.2.11 function wc_GetSubjectRaw

```
int wc_GetSubjectRaw(
    byte ** subjectRaw,
    Cert * cert
)
```

この関数は、証明書構造体から生の subject を取得します。

Parameters:

- **subjectRaw** 正常に返された場合の生の subject へのポインタのポインタ
- **cert** 生の subject を取得する証明書へのポインタ

See:

- [wc_InitCert](#)
- [wc_SetSubjectRaw](#)

Return:

- 0 証明書から subject を正常に取得した場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます

Example

```
Cert myCert;
byte *subjRaw;
// myCert を初期化
if(wc_GetSubjectRaw(&subjRaw, &myCert) != 0) {
    // subject 設定エラー
}
```

B.1.2.12 function wc_SetAltNames

```
int wc_SetAltNames(
```

```

    Cert * cert,
    const char * file
)

```

この関数は、証明書の代替名を、提供された pem ファイル内の代替名に設定します。これは、同じ証明書で複数のドメインを保護したい場合に便利です。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 代替名を設定する証明書へのポインタ
- **file** pem 形式の証明書を含むファイルのパス

See:

- [wc_InitCert](#)
- [wc_SetIssuer](#)

Return:

- 0 証明書の代替名を正常に設定した場合に返されます
- MEMORY_E XMALLOc でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```

Cert myCert;
// myCert を初期化
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
    // 代替名設定エラー
}

```

B.1.2.13 function wc_SetIssuerBuffer

```

int wc_SetIssuerBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)

```

この関数は、提供された der バッファの発行者から証明書の発行者を設定します。また、証明書の自己署名属性を false に変更します。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 発行者を設定する証明書へのポインタ
- **der** 発行者を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** 発行者を取得する der 形式の証明書を含むバッファのサイズ

See:

- [wc_InitCert](#)
- [wc_SetIssuer](#)

Return:

- 0 証明書の発行者を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);
// der を初期化
if(wc_SetIssuerBuffer(&myCert, der, FOURK_BUF) != 0) {
    // issuer 設定エラー
}
```

B.1.2.14 function wc_SetIssuerRaw

```
int wc_SetIssuerRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は、提供された der バッファの発行者から証明書の生の発行者を設定します。このメソッドは、署名前に生の発行者フィールドを設定するために使用されます。

Parameters:

- **cert** 生の発行者を設定する証明書へのポインタ
- **der** subject を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** subject を取得する der 形式の証明書を含むバッファのサイズ

See:

- `wc_InitCert`
- `wc_SetIssuer`

Return:

- 0 証明書の発行者を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);
// der を初期化
if(wc_SetIssuerRaw(&myCert, der, FOURK_BUF) != 0) {
    // subject 設定エラー
}
```

B.1.2.15 function wc_SetSubjectBuffer

```
int wc_SetSubjectBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は、提供された der バッファの subject から証明書の subject を設定します。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** subject を設定する証明書へのポインタ
- **der** subject を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** subject を取得する der 形式の証明書を含むバッファのサイズ

See:

- `wc_InitCert`
- `wc_SetSubject`

Return:

- 0 証明書の subject を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);
// der を初期化
if(wc_SetSubjectBuffer(&myCert, der, FOURK_BUF) != 0) {
    // subject 設定エラー
}
```

B.1.2.16 function wc_SetAltNamesBuffer

```
int wc_SetAltNamesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は、提供された der バッファの代替名から証明書の代替名を設定します。これは、同じ証明書で複数のドメインを保護したい場合に便利です。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 代替名を設定する証明書へのポインタ
- **der** 代替名を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** 代替名を取得する der 形式の証明書を含むバッファのサイズ

See:

- [wc_InitCert](#)
- [wc_SetAltNames](#)

Return:

- 0 証明書の代替名を正常に設定した場合に返されます

- MEMORY_E_XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E_ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);
// der を初期化
if(wc_SetAltNamesBuffer(&myCert, der, FOURK_BUF) != 0) {
    // subject 設定エラー
}
```

B.1.2.17 function wc_SetDatesBuffer

```
int wc_SetDatesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は、提供された der バッファの日付範囲から証明書の日付を設定します。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 日付を設定する証明書へのポインタ
- **der** 日付範囲を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** 日付範囲を取得する der 形式の証明書を含むバッファのサイズ

See: [wc_InitCert](#)

Return:

- 0 証明書の日付を正常に設定した場合に返されます
- MEMORY_E_XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます

- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);
// der を初期化
if(wc_SetDatesBuffer(&myCert, der, FOURK_BUF) != 0) {
    // subject 設定エラー
}
```

B.1.2.18 function wc_SetAuthKeyIdFromPublicKey

```
int wc_SetAuthKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsaKey,
    ecc_key * eckey
)
```

RSA または ECC 公開鍵から AKID を設定します。注：rsaKey または eckey のいずれか一方のみを設定し、両方は設定しないでください。

Parameters:

- **cert** SKID を設定する証明書へのポインタ。
- **rsaKey** 読み取り元の RsaKey 構造体へのポインタ。
- **eckey** 読み取り元の ecc_key へのポインタ。

See:

- [wc_SetSubjectKeyId](#)
- [wc_SetAuthKeyId](#)
- [wc_SetAuthKeyIdFromCert](#)

Return:

- 0 成功
- BAD_FUNC_ARG cert が null または rsaKey と eckey の両方が null の場合。
- MEMORY_E メモリの割り当てエラー。
- PUBLIC_KEY_E キーへの書き込みエラー。

Example

```
Cert myCert;
RsaKey keypub;

wc_InitRsaKey(&keypub, 0);
```

```

if (wc_SetAuthKeyIdFromPublicKey(&myCert, &keypub, NULL) != 0)
{
    // エラーを処理
}

```

B.1.2.19 function wc_SetAuthKeyIdFromCert

```

int wc_SetAuthKeyIdFromCert(
    Cert * cert,
    const byte * der,
    int derSz
)

```

DER エンコードされた証明書から AKID を設定します。

Parameters:

- **cert** 書き込み先の Cert 構造体。
- **der** DER エンコードされた証明書バッファ。
- **derSz** der のサイズ (バイト単位)。

See:

- [wc_SetAuthKeyIdFromPublicKey](#)
- [wc_SetAuthKeyId](#)

Return:

- 0 成功
- BAD_FUNC_ARG いずれかの引数が null または derSz が 0 未満の場合のエラー。
- MEMORY_E メモリの割り当てに問題がある場合のエラー。
- ASN_NO_SKID サブジェクトキー ID が見つかりません。

Example

```

Cert some_cert;
byte some_der[] = { // DER バッファを初期化 };
wc_InitCert(&some_cert);
if(wc_SetAuthKeyIdFromCert(&some_cert, some_der, sizeof(some_der) != 0)
{
    // エラーを処理
}

```

B.1.2.20 function wc_SetAuthKeyId

```

int wc_SetAuthKeyId(
    Cert * cert,
    const char * file
)

```

PEM 形式の証明書ファイルから AKID を設定します。

Parameters:

- **cert** AKID を設定したい Cert 構造体。
- **file** PEM 証明書ファイルを含むバッファ。

See:

- [wc_SetAuthKeyIdFromPublicKey](#)

- `wc_SetAuthKeyIdFromCert`

Return:

- 0 成功
- BAD_FUNC_ARG cert または file が null の場合のエラー。
- MEMORY_E メモリの割り当てに問題がある場合のエラー。

Example

```
char* file_name = "/path/to/file";
cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetAuthKeyId(&some_cert, file_name) != 0)
{
    // エラーを処理
}
```

B.1.2.21 function wc_SetSubjectKeyIdFromPublicKey

```
int wc_SetSubjectKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsakey,
    ecc_key * eckey
)
```

RSA または ECC 公開鍵から SKID を設定します。

Parameters:

- **cert** 使用する Cert 構造体へのポインタ。
- **rsakey** RsaKey 構造体へのポインタ
- **eckey** ecc_key 構造体へのポインタ

See: `wc_SetSubjectKeyId`

Return:

- 0 成功
- BAD_FUNC_ARG cert または rsakey と eckey が null の場合に返されます。
- MEMORY_E メモリの割り当てエラーがある場合に返されます。
- PUBLIC_KEY_E 公開鍵の取得エラーがある場合に返されます。

Example

```
Cert some_cert;
RsaKey some_key;
wc_InitCert(&some_cert);
wc_InitRsaKey(&some_key);

if(wc_SetSubjectKeyIdFromPublicKey(&some_cert,&some_key, NULL) != 0)
{
    // エラーを処理
}
```

B.1.2.22 function wc_SetSubjectKeyId

```
int wc_SetSubjectKeyId(
    Cert * cert,
    const char * file
```

)

PEM 形式の公開鍵ファイルから SKID を設定します。両方の引数が必要です。

Parameters:

- **cert** SKID を設定する Cert 構造体。
- **file** PEM エンコードされたファイルを含む。

See: [wc_SetSubjectKeyIdFromPublicKey](#)

Return:

- 0 成功
- BAD_FUNC_ARG cert または file が null の場合に返されます。
- MEMORY_E キー用のメモリ割り当てに問題がある場合に返されます。
- PUBLIC_KEY_E 公開鍵のデコードエラーがある場合に返されます。

Example

```
const char* file_name = "path/to/file";
Cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetSubjectKeyId(&some_cert, file_name) != 0)
{
    // エラーを処理
}
```

B.1.2.23 function wc_PemPubKeyToDer

```
int wc_PemPubKeyToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

ファイルから PEM キーを読み込み、DER エンコードされたバッファに変換します。

Parameters:

- **fileName** ロードするファイルの名前。
- **derBuf** DER エンコードされたキー用のバッファ。
- **derSz** DER バッファのサイズ。

See: [wc_PubKeyPemToDer](#)

Return:

- 0 成功
- <0 エラー
- SSL_BAD_FILE ファイルを開く際に問題があります。
- MEMORY_E ファイルバッファ用のメモリ割り当てエラーがあります。
- BUFFER_E derBuf が変換されたキーを保持するのに十分な大きさではありません。

Example

```
char* some_file = "filename";
unsigned char der[];

if(wc_PemPubKeyToDer(some_file, der, sizeof(der)) != 0)
{
    // エラーを処理
}
```

```

    // エラーを処理
}

```

B.1.2.24 function wc_PubKeyPemToDer

```

int wc_PubKeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz
)

```

PEM エンコードされた公開鍵を DER に変換します。バッファに書き込まれたバイト数、またはエラーの場合は負の値を返します。

Parameters:

- **pem** PEM エンコードされたキー
- **pemSz** pem のサイズ
- **buff** 出力用バッファへのポインタ。
- **buffSz** バッファのサイズ。

See: [wc_PemPubKeyToDer](#)

Return:

- 0 成功、書き込まれたバイト数。
- BAD_FUNC_ARG pem、buff、または buffSz が null の場合に返されます
- <0 関数内でエラーが発生しました。

Example

```

byte some_pem[] = { PEM キーで初期化 }
unsigned char out_buffer[1024]; // バッファが DER を格納するのに十分な大きさであること
    ⇨ を確認

if(wc_PubKeyPemToDer(some_pem, sizeof(some_pem), out_buffer,
    sizeof(out_buffer)) < 0)
{
    // エラーを処理
}

```

B.1.2.25 function wc_PemCertToDer

```

int wc_PemCertToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)

```

この関数は、pem 証明書を der 証明書に変換し、結果の証明書を提供された derBuf バッファに配置します。

Parameters:

- **fileName** der 証明書に変換する pem 証明書を含むファイルへのパス
- **derBuf** 変換された証明書を格納する char バッファへのポインタ
- **derSz** 変換された証明書を格納する char バッファのサイズ

See: なし

Return:

- Success 成功時に生成された derBuf のサイズを返します
- BUFFER_E derBuf のサイズが生成された証明書を保持するには小さすぎる場合に返されます
- MEMORY_E XMALLOC の呼び出しが失敗した場合に返されます

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 エラー
}
```

B.1.2.26 function wc_DerToPem

```
int wc_DerToPem(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outSz,
    int type
)
```

この関数は、der バッファに含まれる der 形式の入力証明書を、output バッファに含まれる pem 形式の出力証明書に変換します。これはインプレース変換ではなく、pem 形式の出力を格納するために別のバッファを使用する必要があることに注意してください。

Parameters:

- **der** 変換する証明書のバッファへのポインタ
- **derSz** 変換する証明書のサイズ
- **output** pem 形式の証明書を格納するバッファへのポインタ
- **outSz** pem 形式の証明書を格納するバッファのサイズ
- **type** 生成する証明書のタイプ。有効なタイプは：CERT_TYPE、PRIVATEKEY_TYPE、ECC_PRIVATEKEY_TYPE、および CERTREQ_TYPE。

See: [wc_PemCertToDer](#)

Return:

- Success 入力 der 証明書から正常に pem 証明書を作成すると、生成された pem 証明書のサイズを返します。
- BAD_FUNC_ARG der ファイルを解析して pem ファイルとして格納する際にエラーがある場合に返されます
- MEMORY_E XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_INPUT_E base64 エンコードエラーの場合に返されます
- BUFFER_E 出力バッファが pem 形式の証明書を格納するには小さすぎる場合に返される可能性があります

Example

```
byte* der;
// 証明書で der を初期化
byte* pemFormatted[FOURK_BUF];
```



```
word32 pemSz;
pemSz = wc_DerToPem(der, derSz, pemFormatted, FOURK_BUF, CERT_TYPE);
```

B.1.2.27 function wc_DerToPemEx

```
int wc_DerToPemEx(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outSz,
    byte * cipher_info,
    int type
)
```

この関数は、der 形式の入力証明書を変換します、der バッファに含まれる、pem 形式の出力証明書に変換し、output バッファに格納します。これはインプレース変換ではなく、pem 形式の出力を格納するために別のバッファを使用する必要があることに注意してください。暗号情報の設定を許可します。

Parameters:

- **der** 変換する証明書のバッファへのポインタ
- **derSz** 変換する証明書のサイズ
- **output** pem 形式の証明書を格納するバッファへのポインタ
- **outSz** pem 形式の証明書を格納するバッファのサイズ
- **cipher_info** 追加の暗号情報。
- **type** 生成する証明書のタイプ。有効なタイプは：CERT_TYPE、PRIVATEKEY_TYPE、ECC_PRIVATEKEY_TYPE、および CERTREQ_TYPE。

See: [wc_PemCertToDer](#)

Return:

- Success 入力 der 証明書から正常に pem 証明書を作成すると、生成された pem 証明書のサイズを返します。
- BAD_FUNC_ARG der ファイルを解析して pem ファイルとして格納する際にエラーがある場合に返されます
- MEMORY_E XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_INPUT_E base64 エンコードエラーの場合に返されます
- BUFFER_E 出力バッファが pem 形式の証明書を格納するには小さすぎる場合に返される可能性があります

Example

```
byte* der;
// 証明書で der を初期化
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
byte* cipher_info[] { 追加の暗号情報。 }
pemSz = wc_DerToPemEx(der, derSz, pemFormatted, FOURK_BUF, cipher_info,
    ↪ CERT_TYPE);
```

B.1.2.28 function wc_EccPrivateKeyDecode

```
int wc_EccPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
```

```

    word32 inSz
)

```

この関数は、入力バッファ input から ECC 秘密鍵を読み取り、秘密鍵を解析し、それを使用して ecc_key オブジェクトを生成し、key に格納します。

Parameters:

- **input** 入力秘密鍵を含むバッファへのポインタ
- **inOutIdx** バッファ内で開始するインデックスを含む word32 オブジェクトへのポインタ
- **key** デコードされた秘密鍵を格納する初期化された ecc オブジェクトへのポインタ
- **inSz** 秘密鍵を含む入力バッファのサイズ

See: wc_RSA_PrivateKeyDecode

Return:

- 0 秘密鍵のデコードに成功し、結果を ecc_key 構造体に格納した場合
- ASN_PARSE_E der ファイルを解析して pem ファイルとして格納する際にエラーがある場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 変換する証明書が指定された最大証明書サイズより大きい場合に返されます
- ASN_OBJECT_ID_E 証明書エンコーディングに無効なオブジェクト ID がある場合に返されます
- ECC_CURVE_OID_E 提供されたキーの ECC 曲線がサポートされていない場合に返されます
- ECC_BAD_ARG_E ECC キー形式にエラーがある場合に返されます
- NOT_COMPILED_IN 秘密鍵が圧縮されており、圧縮キーが提供されていない場合に返されます
- MP_MEM 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます
- MP_VAL 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます
- MP_RANGE 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます

Example

```

int ret, idx=0;
ecc_key key; // キーを格納

byte* tmp; // キーを読み取るための一時バッファ
tmp = (byte*) malloc(FOURK_BUF);

int inSz;
inSz = fread(tmp, 1, FOURK_BUF, privateKeyFile);
// tmp バッファにキーを読み取る

wc_ecc_init(&key); // キーを初期化
ret = wc_EccPrivateKeyDecode(tmp, &idx, &key, (word32)inSz);
if(ret < 0) {
    // ecc キーのデコードエラー
}

```

B.1.2.29 function wc_EccKeyToDer

```

int wc_EccKeyToDer(
    ecc_key * key,
    byte * output,
    word32 inLen
)

```

この関数は、秘密 ECC キーを der 形式に書き込みます。

Parameters:

- **key** 入力 ecc キーを含むバッファへのポインタ
- **output** der 形式のキーを格納するバッファへのポインタ
- **inLen** der 形式のキーを格納するバッファの長さ

See: [wc_RsaKeyToDer](#)

Return:

- Success ECC キーの der 形式への書き込みに成功すると、バッファに書き込まれた長さを返します
- BAD_FUNC_ARG key または output が null、または inLen がゼロの場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 変換された証明書が出力バッファに格納するには大きすぎる場合に返されます
- ASN_UNKNOWN_OID_E 使用されている ECC キーが不明なタイプの場合に返されます
- MP_MEM 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます
- MP_VAL 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます
- MP_RANGE 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます

Example

```
int derSz;
ecc_key key;
// キーを初期化して作成
byte der[FOURK_BUF];
// ここに der 形式のキーを格納

derSz = wc_EccKeyToDer(&key, der, FOURK_BUF);
if(derSz < 0) {
    // ecc キーを der バッファに変換する際のエラー
}
```

B.1.2.30 function wc_EccPublicKeyDecode

```
int wc_EccPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

入力バッファから ECC 公開鍵をデコードします。ECC キーを取得するために ASN シーケンスを解析します。

Parameters:

- **input** デコードする DER エンコードされたキーを含むバッファ。
- **inOutIdx** 入力バッファの読み取り開始位置のインデックス。出力時、インデックスは入力バッファの最後に解析された位置に設定されます。
- **key** 公開鍵を格納する ecc_key 構造体へのポインタ。
- **inSz** 入力バッファのサイズ。

See: [wc_ecc_import_x963](#)

Return:

- 0 成功
- BAD_FUNC_ARG いずれかの引数が null の場合に返されます。
- ASN_PARSE_E 解析エラーがある場合に返されます
- ASN_ECC_KEY_E キーのインポートエラーがある場合に返されます。考えられる理由については [wc_ecc_import_x963](#) を参照してください。

Example

```

int ret;
word32 idx = 0;
byte buff[] = { // キーで初期化 };
ecc_key pubKey;
wc_ecc_init(&pubKey);
if ( wc_EccPublicKeyDecode(buff, &idx, &pubKey, sizeof(buff)) != 0) {
    // キーのデコードエラー
}

```

B.1.2.31 function wc_EccPublicKeyToDer

```

int wc_EccPublicKeyToDer(
    ecc_key * key,
    byte * output,
    word32 inLen,
    int with_AlgCurve
)

```

この関数は、ECC 公開鍵を DER 形式に変換します。使用されたバッファのサイズを返します。DER 形式の ECC 公開鍵は出力バッファに格納されます。with_AlgCurve フラグは、アルゴリズムと曲線情報を持つヘッダーを含めます

Parameters:

- **key** ECC キーへのポインタ
- **output** 書き込み先の出力バッファへのポインタ。
- **inLen** バッファのサイズ。
- **with_AlgCurve** アルゴリズムと曲線情報を持つヘッダーを含めるタイミングのフラグ。

See:

- [wc_EccKeyToDer](#)
- [wc_EccPrivateKeyDecode](#)

Return:

- 0 成功、使用されたバッファのサイズ
- BAD_FUNC_ARG output または key が null の場合に返されます。
- LENGTH_ONLY_E ECC 公開鍵のサイズ取得エラー。
- BUFFER_E 出力バッファが小さすぎる場合に返されます。

Example

```

ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // der 用の適切なサイズ;
byte der[derSz];

if(wc_EccPublicKeyToDer(&key, der, derSz, 1) < 0)
{
    // ECC 公開鍵の der 変換エラー
}

```

B.1.2.32 function wc_EccPublicKeyToDer_ex

```
int wc_EccPublicKeyToDer_ex(
    ecc_key * key,
    byte * output,
    word32 inLen,
    int with_AlgCurve,
    int comp
)
```

この関数は、ECC 公開鍵を DER 形式に変換します。使用されたバッファのサイズを返します。DER 形式の ECC 公開鍵は出力バッファに格納されます。with_AlgCurve フラグは、アルゴリズムと曲線情報を持つヘッダーを含めます。comp パラメータは、公開鍵を圧縮形式でエクスポートするかどうかを決定します。

Parameters:

- **key** ECC キーへのポインタ
- **output** 書き込み先の出力バッファへのポインタ。
- **inLen** バッファのサイズ。
- **with_AlgCurve** アルゴリズムと曲線情報を持つヘッダーを含めるタイミングのフラグ。
- **comp** 1（ゼロ以外）の場合、ECC 公開鍵は圧縮形式で書き込まれます。0 の場合、非圧縮形式で書き込まれます。

See:

- [wc_EccKeyToDer](#)
- [wc_EccPublicKeyDecode](#)

Return:

- 0 成功、使用されたバッファのサイズ
- BAD_FUNC_ARG output または key が null の場合に返されます。
- LENGTH_ONLY_E ECC 公開鍵のサイズ取得エラー。
- BUFFER_E 出力バッファが小さすぎる場合に返されます。

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // der 用の適切なサイズ;
byte der[derSz];

// 圧縮された ECC キーを書き出す
if(wc_EccPublicKeyToDer_ex(&key, der, derSz, 1, 1) < 0)
{
    // ECC 公開鍵の der 変換エラー
}
```

B.1.2.33 function wc_Curve25519PrivateKeyDecode

```
int wc_Curve25519PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

)

この関数は、DER エンコードされたバッファから Curve25519 秘密鍵（のみ）をデコードします

Parameters:

- **input** DER エンコードされた秘密鍵を含むバッファへのポインタ
- **inOutIdx** 入力バッファの読み取り開始位置のインデックス。出力時、インデックスは入力バッファの最後に解析された位置に設定されます。
- **key** デコードされたキーを格納する curve25519_key 構造体へのポインタ
- **inSz** 入力 DER バッファのサイズ

See:

- [wc_Curve25519KeyDecode](#)
- [wc_Curve25519PublicKeyDecode](#)

Return:

- 0 成功
- BAD_FUNC_ARG input、inOutIdx または key が null の場合に返されます
- ASN_PARSE_E DER エンコードされたデータの解析エラーがある場合に返されます
- ECC_BAD_ARG_E キー長が CURVE25519_KEYSIZE でない場合、または DER キーが適切にフォーマットされているにもかかわらず他の問題を含む場合に返されます。
- BUFFER_E 入力バッファが有効な DER エンコードされたキーを含むには小さすぎる場合に返されます。

Example

```
byte der[] = { // DER エンコードされたキー };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);

if (wc_Curve25519PrivateKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // 秘密鍵のデコードエラー
}
```

B.1.2.34 function wc_Curve25519PublicKeyDecode

```
int wc_Curve25519PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

この関数は、DER エンコードされたバッファから Curve25519 公開鍵（のみ）をデコードします。

Parameters:

- **input** DER エンコードされた公開鍵を含むバッファへのポインタ
- **inOutIdx** 入力バッファの読み取り開始位置のインデックス。出力時、インデックスは入力バッファの最後に解析された位置に設定されます。
- **key** デコードされたキーを格納する curve25519_key 構造体へのポインタ
- **inSz** 入力 DER バッファのサイズ

See:

- [wc_Curve25519KeyDecode](#)
- [wc_Curve25519PrivateKeyDecode](#)

Return:

- 0 成功
- BAD_FUNC_ARG input、inOutIdx または key が null の場合に返されます
- ASN_PARSE_E DER エンコードされたデータの解析エラーがある場合に返されます
- ECC_BAD_ARG_E キー長が CURVE25519_KEYSIZE でない場合、または DER キーが適切にフォーマットされているにもかかわらず他の問題を含む場合に返されます。
- BUFFER_E 入力バッファが有効な DER エンコードされたキーを含むには小さすぎる場合に返されます。

Example

```
byte der[] = { // DER エンコードされたキー };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);
if (wc_Curve25519PublicKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // 公開鍵のデコードエラー
}
```

B.1.2.35 function wc_Curve25519KeyDecode

```
int wc_Curve25519KeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

この関数は、DER エンコードされたバッファから Curve25519 キーをデコードします。秘密鍵、公開鍵、または両方をデコードできます。

Parameters:

- **input** DER エンコードされたキーを含むバッファへのポインタ
- **inOutIdx** 入力バッファの読み取り開始位置のインデックス。出力時、インデックスは入力バッファの最後に解析された位置に設定されます。
- **key** デコードされたキーを格納する curve25519_key 構造体へのポインタ
- **inSz** 入力 DER バッファのサイズ

See:

- [wc_Curve25519PrivateKeyDecode](#)
- [wc_Curve25519PublicKeyDecode](#)

Return:

- 0 成功
- BAD_FUNC_ARG input、inOutIdx または key が null の場合に返されます
- ASN_PARSE_E DER エンコードされたデータの解析エラーがある場合に返されます
- ECC_BAD_ARG_E キー長が CURVE25519_KEYSIZE でない場合、または DER キーが適切にフォーマットされているにもかかわらず他の問題を含む場合に返されます。
- BUFFER_E 入力バッファが有効な DER エンコードされたキーを含むには小さすぎる場合に返されます。

Example

```
byte der[] = { // DER エンコードされたキー };
word32 idx = 0;
curve25519_key key;
```

```

wc_curve25519_init(&key);
if (wc_Curve25519KeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // キーのデコードエラー
}

```

B.1.2.36 function wc_Curve25519PrivateKeyToDer

```

int wc_Curve25519PrivateKeyToDer(
    curve25519_key * key,
    byte * output,
    word32 inLen
)

```

この関数は、Curve25519 秘密鍵を DER 形式にエンコードします。入力キー構造体に公開鍵が含まれている場合、それは無視されます。

Parameters:

- **key** エンコードする秘密鍵を含む curve25519_key 構造体へのポインタ
- **output** DER エンコーディングを保持するバッファ
- **inLen** 出力バッファのサイズ

See:

- [wc_Curve25519KeyToDer](#)
- [wc_Curve25519PublicKeyToDer](#)

Return:

- 0 成功、DER エンコーディングの長さ
- BAD_FUNC_ARG key または output が null の場合に返されます
- MEMORY_E 割り当て失敗がある場合に返されます
- BUFFER_E 出力バッファが小さすぎる場合に返されます

Example

```

curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // 出力 DER 用の適切なサイズ
byte der[derSz];
wc_Curve25519PrivateKeyToDer(&key, der, derSz);

```

B.1.2.37 function wc_Curve25519PublicKeyToDer

```

int wc_Curve25519PublicKeyToDer(
    curve25519_key * key,
    byte * output,
    word32 inLen,
    int withAlg
)

```

この関数は、Curve25519 公開鍵を DER 形式にエンコードします。入力キー構造体に秘密鍵が含まれている場合、それは無視されます。

Parameters:

- **key** エンコードする公開鍵を含む curve25519_key 構造体へのポインタ
- **output** DER エンコーディングを保持するバッファ

- **inLen** 出力バッファのサイズ
- **withAlg** DER エンコーディングにアルゴリズム識別子を含めるかどうか

See:

- [wc_Curve25519KeyToDer](#)
- [wc_Curve25519PrivateKeyToDer](#)

Return:

- 0 成功、DER エンコーディングの長さ
- BAD_FUNC_ARG key または output が null の場合に返されます
- MEMORY_E 割り当て失敗がある場合に返されます
- BUFFER_E 出力バッファが小さすぎる場合に返されます

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // 出力 DER 用の適切なサイズ
byte der[derSz];
wc_Curve25519PublicKeyToDer(&key, der, derSz, 1);
```

B.1.2.38 function wc_Curve25519KeyToDer

```
int wc_Curve25519KeyToDer(
    curve25519_key * key,
    byte * output,
    word32 inLen,
    int withAlg
)
```

この関数は、Curve25519 キーを DER 形式にエンコードします。秘密鍵、公開鍵、または両方をエンコードできます。

Parameters:

- **key** エンコードするキーを含む curve25519_key 構造体へのポインタ
- **output** DER エンコーディングを保持するバッファ
- **inLen** 出力バッファのサイズ
- **withAlg** DER エンコーディングにアルゴリズム識別子を含めるかどうか

See:

- [wc_Curve25519PrivateKeyToDer](#)
- [wc_Curve25519PublicKeyToDer](#)

Return:

- 0 成功、DER エンコーディングの長さ
- BAD_FUNC_ARG key または output が null の場合に返されます
- MEMORY_E 割り当て失敗がある場合に返されます
- BUFFER_E 出力バッファが小さすぎる場合に返されます

Example

```

curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // 出力 DER 用の適切なサイズ
byte der[derSz];
wc_Curve25519KeyToDer(&key, der, derSz, 1);

```

B.1.2.39 function wc_EncodeSignature

```

word32 wc_EncodeSignature(
    byte * out,
    const byte * digest,
    word32 digSz,
    int hashOID
)

```

この関数は、デジタル署名を出力バッファにエンコードし、作成されたエンコードされた署名のサイズを返します。

Parameters:

- **out** エンコードされた署名が書き込まれるバッファへのポインタ
- **digest** 署名のエンコードに使用するダイジェストへのポインタ
- **digSz** ダイジェストを含むバッファの長さ
- **hashOID** 署名の生成に使用されるハッシュタイプを識別する OID。ビルド構成に応じた有効なオプションは：SHAh、SHA256h、SHA384h、SHA512h、MD2h、MD5h、DESB、DES3b、CTC_MD5wRSA、CTC_SHAwRSA、CTC_SHA256wRSA、CTC_SHA384wRSA、CTC_SHA512wRSA、CTC_SHAwECDSA、CTC_SHA256wECDSA、CTC_SHA384wECDSA、および CTC_SHA512wECDSA。

See: なし

Return: Success エンコードされた署名を output に正常に書き込むと、バッファに書き込まれた長さを返します

```

int signSz;
byte encodedSig[MAX_ENCODED_SIG_SZ];
Sha256 sha256;
// ハッシュ化のために sha256 を初期化

byte* dig = (byte*)malloc(WC_SHA256_DIGEST_SIZE);
// ハッシュ化とハッシュ更新を実行して dig に SHA-256 ハッシュを格納
// (wc_InitSha256、wc_Sha256Update、wc_Sha256Final を参照)
signSz = wc_EncodeSignature(encodedSig, dig, WC_SHA256_DIGEST_SIZE, SHA256h);

```

B.1.2.40 function wc_GetCTC_HashOID

```

int wc_GetCTC_HashOID(
    int type
)

```

この関数は、ハッシュタイプに対応するハッシュ OID を返します。例えば、WC_SHA512 タイプが与えられた場合、この関数は SHA512 ハッシュに対応する識別子 SHA512h を返します。

Parameters:

- **type** OID を見つけるハッシュタイプ。ビルド構成に応じた有効なオプションには：WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512

See: なし

Return:

- Success 成功時に、その暗号化タイプで使用する適切なハッシュに対応する OID を返します。
- 0 認識されないハッシュタイプが引数として渡された場合に返されます。

Example

```
int hashOID;

hashOID = wc_GetCTC_HashOID(WC_SHA512);
if (hashOID == 0) {
    // WOLFSSL_SHA512 が定義されていません
}
```

B.1.2.41 function wc_SetCert_Free

```
void wc_SetCert_Free(
    Cert * cert
)
```

この関数は、証明書構造体のデコードされた証明書キャッシュによって使用されるメモリとリソースをクリーンアップします。WOLFSSL_CERT_GEN_CACHE が定義されている場合、デコードされた証明書構造体は証明書構造体にキャッシュされます。これにより、証明書設定関数への後続の呼び出しで、各呼び出しでデコードされた証明書を解析することを回避できます。

Parameters:

- **cert** 初期化されていない証明書情報構造体へのポインタ。

See:

- [wc_SetAuthKeyIdFromCert](#)
- [wc_SetIssuerBuffer](#)
- [wc_SetSubjectBuffer](#)
- [wc_SetSubjectRaw](#)
- [wc_SetIssuerRaw](#)
- [wc_SetAltNamesBuffer](#)
- [wc_SetDatesBuffer](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG 無効なポインタが引数として渡された場合に返されます。

Example

```
Cert cert; // 初期化された証明書構造体

wc_SetCert_Free(&cert);
```

B.1.2.42 function wc_GetPkcs8TraditionalOffset

```
int wc_GetPkcs8TraditionalOffset(
    byte * input,
    word32 * inOutIdx,
    word32 sz
)
```

この関数は、PKCS#8 暗号化されていないバッファ内の従来の秘密鍵の先頭を見つけます。

Parameters:

- **input** 暗号化されていない PKCS#8 秘密鍵を含むバッファ。
- **inOutIdx** 入力バッファへのインデックス。入力時、PKCS#8 バッファの先頭へのバイトオフセットである必要があります。出力時、入力バッファ内の従来の秘密鍵へのバイトオフセットになります。
- **sz** 入力バッファのバイト数。

See:

- [wc_CreatePKCS8Key](#)
- [wc_EncryptPKCS8Key](#)
- [wc_DecryptPKCS8Key](#)
- [wc_CreateEncryptedPKCS8Key](#)

Return:

- Length 成功時に従来の秘密鍵の長さ。
- Negative 失敗時に負の値。

Example

```
byte* pkcs8Buf; // PKCS#8 キーを含むバッファ。
word32 idx = 0;
word32 sz; // pkcs8Buf のサイズ。
...
ret = wc_GetPkcs8TraditionalOffset(pkcs8Buf, &idx, sz);
// pkcs8Buf + idx は従来の秘密鍵バイトの先頭になります。
```

B.1.2.43 function wc_CreatePKCS8Key

```
int wc_CreatePKCS8Key(
    byte * out,
    word32 * outSz,
    byte * key,
    word32 keySz,
    int algoID,
    const byte * curveOID,
    word32 oidSz
)
```

この関数は、DER 秘密鍵を受け取り、PKCS#8 形式に変換します。PKCS#12 縮小キーバッグの作成にも使用されます。RFC 5208 を参照してください。

Parameters:

- **out** 結果を配置するバッファ。NULL の場合、outSz に必要な出力バッファサイズが返されます。
- **outSz** out バッファのサイズ。
- **key** 従来の DER キーを持つバッファ。
- **keySz** key バッファのサイズ。
- **algoID** アルゴリズム ID (例: RSAk)。
- **curveOID** 使用される場合の ECC 曲線 OID。RSA キーの場合は NULL である必要があります。
- **oidSz** 曲線 OID のサイズ。curveOID が NULL の場合は 0 に設定されます。

See:

- [wc_GetPkcs8TraditionalOffset](#)
- [wc_EncryptPKCS8Key](#)
- [wc_DecryptPKCS8Key](#)
- [wc_CreateEncryptedPKCS8Key](#)

Return:

- The 成功時に out に配置された PKCS#8 キーのサイズ。
- LENGTH_ONLY_E out が NULL の場合、outSz に必要な出力バッファサイズが返されます。
- Other 失敗時に負の値。

Example

```

ecc_key eccKey;           // wolfSSL ECC キーオブジェクト。
byte* der;                // DER エンコードされた ECC キー。
word32 derSize;           // der のサイズ。
const byte* curveOid = NULL; // eccKey で使用される曲線の OID。
word32 curveOidSz = 0;    // 曲線 OID のサイズ。
byte* pkcs8;              // PKCS#8 キー用の出力バッファ。
word32 pkcs8Sz;           // 出力バッファのサイズ。

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); // pkcs8Sz に必要なサイズを取得。
...
ret = wc_CreatePKCS8Key(pkcs8, &pkcs8Sz, der,
    derSize, ECDSAk, curveOid, curveOidSz);

```

B.1.2.44 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
)

```

この関数は、暗号化されていない PKCS#8 DER キー（例：wc_CreatePKCS8Key によって作成されたもの）を受け取り、PKCS#8 暗号化形式に変換します。結果の暗号化されたキーは、wc_DecryptPKCS8Key を使用して復号できます。RFC 5208 を参照してください。

Parameters:

- **key** 従来の DER キーを持つバッファ。
- **keySz** key バッファのサイズ。
- **out** 結果を配置するバッファ。NULL の場合、outSz に必要な出力バッファサイズが返されます。
- **outSz** out バッファのサイズ。
- **password** パスワードベースの暗号化アルゴリズムに使用するパスワード。
- **passwordSz** パスワードの長さ（NULL ターミネータを含まない）。

- **vPKCS** 使用する PKCS バージョン。PKCS12 または PKCS5 の場合は 1。
- **pbeOid** 使用する PBE スキームの OID（例：PBES2 または RFC 2898 A.3 の PBES1 の OID の 1 つ）。
- **encAlgId** 使用する暗号化アルゴリズム ID（例：AES256CBCb）。
- **salt** 使用するソルトバッファ。NULL の場合、ランダムなソルトが使用されます。
- **saltSz** ソルトバッファの長さ。salt に NULL を渡す場合は 0。
- **itt** KDF に使用する反復回数。
- **rng** 初期化された WC_RNG オブジェクトへのポインタ。
- **heap** 動的割り当てに使用されるヒープへのポインタ。NULL でも可。

See:

- [wc_GetPkcs8TraditionalOffset](#)
- [wc_CreatePKCS8Key](#)
- [wc_DecryptPKCS8Key](#)
- [wc_CreateEncryptedPKCS8Key](#)

Return:

- The 成功時に out に配置された暗号化されたキーのサイズ。
- LENGTH_ONLY_E out が NULL の場合、outSz に必要な出力バッファサイズが返されます。
- Other 失敗時に負の値。

Example

```
byte* pkcs8;           // 暗号化されていない PKCS#8 キー。
word32 pkcs8Sz;        // pkcs8 のサイズ。
byte* pkcs8Enc;        // 暗号化された PKCS#8 キー。
word32 pkcs8EncSz;     // pkcs8Enc のサイズ。
const char* password;  // 暗号化に使用するパスワード。
int passwordSz;        // パスワードの長さ (NULL ターミネータを含まない)。
WC_RNG rng;

// 以下は、pkcs8Enc に pkcs8 の暗号化されたバージョンを生成します。暗号化は
// PKCS#5 のパスワードベース暗号化スキーム 2 (PBE2) と CBC モードの 256 ビット
// キーを持つ AES 暗号を使用します。PKCS#5 の詳細については RFC 8018 を参照してください。
ret = wc_EncryptPKCS8Key(pkcs8, pkcs8Sz, pkcs8Enc, &pkcs8EncSz, password,
    passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
    WC_PKCS12_ITT_DEFAULT, &rng, NULL);
```

B.1.2.45 function wc_DecryptPKCS8Key

```
int wc_DecryptPKCS8Key(
    byte * input,
    word32 sz,
    const char * password,
    int passwordSz
)
```

この関数は、暗号化された PKCS#8 DER キーを受け取り、PKCS#8 暗号化されていない DER に復号します。wc_EncryptPKCS8Key によって行われた暗号化を元に戻します。RFC5208 を参照してください。入力バッファは復号されたデータで上書きされます。

Parameters:

- **input** 入力時、暗号化された PKCS#8 キーを含むバッファ。出力が成功すると、復号されたキーを含みます。
- **sz** 入力バッファのサイズ。
- **password** キーの暗号化に使用されたパスワード。
- **passwordSz** パスワードの長さ (NULL ターミネータを含まない)。

See:

- `wc_GetPkcs8TraditionalOffset`
- `wc_CreatePKCS8Key`
- `wc_EncryptPKCS8Key`
- `wc_CreateEncryptedPKCS8Key`

Return:

- The 成功時に復号されたバッファの長さ。
- Negative 失敗時に負の値。

Example

```
byte* pkcs8Enc;           // wc_EncryptPKCS8Key で作成された暗号化された PKCS#8 キー。
word32 pkcs8EncSz;        // pkcs8Enc のサイズ。
const char* password;     // 復号に使用するパスワード。
int passwordSz;           // パスワードの長さ (NULL ターミネータを含まない)。

ret = wc_DecryptPKCS8Key(pkcs8Enc, pkcs8EncSz, password, passwordSz);
```

B.1.2.46 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
)
```

この関数は、従来の DER キーを受け取り、PKCS#8 形式に変換し、暗号化します。これを行うために `wc_CreatePKCS8Key` と `wc_EncryptPKCS8Key` を使用します。

Parameters:

- **key** 従来の DER キーを持つバッファ。
- **keySz** key バッファのサイズ。
- **out** 結果を配置するバッファ。NULL の場合、outSz に必要な出力バッファサイズが返されます。
- **outSz** out バッファのサイズ。
- **password** パスワードベースの暗号化アルゴリズムに使用するパスワード。
- **passwordSz** パスワードの長さ (NULL ターミネータを含まない)。
- **vPKCS** 使用する PKCS バージョン。PKCS12 または PKCS5 の場合は 1。
- **pbeOid** 使用する PBE スキームの OID (例: PBES2 または RFC 2898 A.3 の PBES1 の OID の 1 つ)。
- **encAlgId** 使用する暗号化アルゴリズム ID (例: AES256CBCb)。
- **salt** 使用するソルトバッファ。NULL の場合、ランダムなソルトが使用されます。
- **saltSz** ソルトバッファの長さ。salt に NULL を渡す場合は 0。
- **itt** KDF に使用する反復回数。
- **rng** 初期化された WC_RNG オブジェクトへのポインタ。

- **heap** 動的割り当てに使用されるヒープへのポインタ。NULL でも可。

See:

- `wc_GetPkcs8TraditionalOffset`
- `wc_CreatePKCS8Key`
- `wc_EncryptPKCS8Key`
- `wc_DecryptPKCS8Key`

Return:

- The 成功時に out に配置された暗号化されたキーのサイズ。
- `LENGTH_ONLY_E` out が NULL の場合、outSz に必要な出力バッファサイズが返されます。
- Other 失敗時に負の値。

Example

```
byte* key;           // 従来の秘密鍵 (DER 形式)。
word32 keySz;        // key のサイズ。
byte* pkcs8Enc;      // 暗号化された PKCS#8 キー。
word32 pkcs8EncSz;    // pkcs8Enc のサイズ。
const char* password; // 暗号化に使用するパスワード。
int passwordSz;       // パスワードの長さ (NULL ターミネータを含まない)。
WC_RNG rng;

// 以下は、pkcs8Enc に key の暗号化された PKCS#8 バージョンを生成します。
// 暗号化は PKCS#5 のパスワードベース暗号化スキーム 2 (PBE2) と CBC モードの
// 256 ビットキーを持つ AES 暗号を使用します。PKCS#5 の詳細については
// RFC 8018 を参照してください。
ret = wc_CreateEncryptedPKCS8Key(key, keySz, pkcs8Enc, &pkcs8EncSz,
    password, passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
    WC_PKCS12_ITT_DEFAULT, &rng, NULL);
```

B.1.2.47 function wc_InitDecodedCert

```
void wc_InitDecodedCert(
    struct DecodedCert * cert,
    const byte * source,
    word32 inSz,
    void * heap
)
```

この関数は、“cert” パラメータが指す DecodedCert を初期化します。長さ “inSz” の DER エンコードされた証明書への “source” ポインタを保存します。この証明書は、wc_ParseCert への後続の呼び出しによって解析できます。

Parameters:

- **cert** 割り当てられた DecodedCert オブジェクトへのポインタ。
- **source** DER エンコードされた証明書へのポインタ。
- **inSz** DER エンコードされた証明書の長さ (バイト単位)。
- **heap** 動的割り当てに使用されるヒープへのポインタ。NULL でも可。

See:

- `wc_ParseCert`
- `wc_FreeDecodedCert`

Example


```
DecodedCert decodedCert; // デコードされた証明書オブジェクト。
byte* certBuf;           // DER エンコードされた証明書バッファ。
word32 certBufSz;        // certBuf のサイズ (バイト単位)。

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
```

B.1.2.48 function wc_ParseCert

```
int wc_ParseCert(
    DecodedCert * cert,
    int type,
    int verify,
    void * cm
)
```

この関数は、DecodedCert オブジェクトに保存された DER エンコードされた証明書を解析し、そのオブジェクトのフィールドを設定します。DecodedCert は、wc_InitDecodedCert への事前の呼び出しで初期化されている必要があります。この関数は、CertificateManager オブジェクトへのオプションのポインタを取り、CA が CertificateManager で見つかった場合、DecodedCert の証明機関情報を設定するために使用されます。

Parameters:

- **cert** 初期化された DecodedCert オブジェクトへのポインタ。
- **type** 証明書のタイプ。asn_public.h の CertType 列挙型を参照してください。
- **verify** 設定されている場合、ユーザーが証明書の有効性を検証したいことを示すフラグ。
- **cm** CertificateManager へのオプションのポインタ。NULL でも可。

See:

- [wc_InitDecodedCert](#)
- [wc_FreeDecodedCert](#)

Return:

- 0 成功時。
- Other 失敗時に負の値。

Example

```
int ret;
DecodedCert decodedCert; // デコードされた証明書オブジェクト。
byte* certBuf;           // DER エンコードされた証明書バッファ。
word32 certBufSz;        // certBuf のサイズ (バイト単位)。

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
ret = wc_ParseCert(&decodedCert, CERT_TYPE, NO_VERIFY, NULL);
if (ret != 0) {
    fprintf(stderr, "wc_ParseCert failed.\n");
}
```

B.1.2.49 function wc_FreeDecodedCert

```
void wc_FreeDecodedCert(
    struct DecodedCert * cert
)
```

この関数は、wc_InitDecodedCert で以前に初期化された DecodedCert を解放します。

Parameters:

- **cert** 初期化された DecodedCert オブジェクトへのポインタ。

See:

- [wc_InitDecodedCert](#)
- [wc_ParseCert](#)

Example

```
int ret;
DecodedCert decodedCert; // デコードされた証明書オブジェクト。
byte* certBuf;           // DER エンコードされた証明書バッファ。
word32 certBufSz;        // certBuf のサイズ (バイト単位)。

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);
```

B.1.2.50 function wc_SetTimeCb

```
int wc_SetTimeCb(
    wc_time_cb f
)
```

この関数は、wolfSSL が現在時刻を取得する必要があるときに使用される時刻コールバックを登録します。コールバックのプロトタイプは、C 標準ライブラリの “time” 関数と同じである必要があります。

Parameters:

- **f** 時刻コールバックとして登録する関数。

See: [wc_Time](#)

Return: 0 成功時に返されます。

Example

```
int ret = 0;
// 時刻コールバックのプロトタイプ
time_t my_time_cb(time_t* t);
// 登録する
ret = wc_SetTimeCb(my_time_cb);
if (ret != 0) {
    // 時刻コールバックの設定失敗
}
time_t my_time_cb(time_t* t)
{
    // カスタム時刻関数
}
```

B.1.2.51 function wc_Time

```
time_t wc_Time(
    time_t * t
)
```

この関数は、現在時刻を取得します。デフォルトでは、プラットフォーム間で異なる XTIME マクロを使用します。ユーザーは、wc_SetTimeCb 関数を介して任意の関数を使用できます。

Parameters:

- **t** 現在時刻を設定するオプションの time_t ポインタ。

See: [wc_SetTimeCb](#)

Return: Time 成功時に返される現在時刻。

Example

```
time_t currentTime = 0;
currentTime = wc_Time(NULL);
wc_Time(&currentTime);
```

B.1.2.52 function wc_SetCustomExtension

```
int wc_SetCustomExtension(
    Cert * cert,
    int critical,
    const char * oid,
    const byte * der,
    word32 derSz
)
```

この関数は、X.509 証明書にカスタム拡張を挿入します。注：ポインタであるパラメータのいずれかが指すアドレスのコンテンツは、証明書が生成されて der 出力が得られるまで変更してはいけません。この関数はコンテンツを別のバッファにコピーしません。

Parameters:

- **cert** 初期化された DecodedCert オブジェクトへのポインタ。
- **critical** 0 の場合、拡張は重要としてマークされません。それ以外の場合は重要としてマークされます。
- **oid** ドット区切りの oid を文字列として。例：“1.2.840.10045.3.1.7”
- **der** 拡張のコンテンツの der エンコーディング。
- **derSz** der エンコーディングのサイズ（バイト単位）。

See:

- [wc_InitCert](#)
- [wc_SetUnknownExtCallback](#)

Return:

- 0 成功時に返されます。
- Other 失敗時に負の値。

Example

```
int ret = 0;
Cert newCert;
wc_InitCert(&newCert);

// subject、公開鍵、発行者、その他のものを設定するコードがここに入ります。

ret = wc_SetCustomExtension(&newCert, 1, "1.2.3.4.5",
    (const byte *)"This is a critical extension", 28);
if (ret < 0) {
    // 拡張の設定失敗。
}
```

```
ret = wc_SetCustomExtension(&newCert, 0, "1.2.3.4.6",
    (const byte *)"This is NOT a critical extension", 32)
if (ret < 0) {
    // 拡張の設定失敗。
}

// 証明書に署名してから書き出すコードがここに入ります。
```

B.1.2.53 function wc_SetUnknownExtCallback

```
int wc_SetUnknownExtCallback(
    DecodedCert * cert,
    wc_UnknownExtCallback cb
)
```

この関数は、wolfSSL が証明書の解析中に証明書内の不明な X.509 拡張に遭遇したときに使用されるコールバックを登録します。コールバックのプロトタイプは次のようにする必要があります：

Parameters:

- **cert** このコールバックに関連付けられる DecodedCert 構造体。
- **cb** 時刻コールバックとして登録する関数。

See:

- ParseCert
- [wc_SetCustomExtension](#)

Return:

- 0 成功時に返されます。
- Other 失敗時に負の値。

Example

```
int ret = 0;
// 不明な拡張コールバックのプロトタイプ
int myUnknownExtCallback(const word16* oid, word32 oidSz, int crit,
    const unsigned char* der, word32 derSz);

// 登録する
ret = wc_SetUnknownExtCallback(cert, myUnknownExtCallback);
if (ret != 0) {
    // コールバックの設定失敗
}

// oid: oid のドット区切り値である整数の配列。
// oidSz: oid 内の値の数。
// crit: 拡張が重要としてマークされたかどうか。
// der: 拡張のコンテンツの der エンコーディング。
// derSz: der エンコーディングのサイズ (バイト単位)。
int myCustomExtCallback(const word16* oid, word32 oidSz, int crit,
    const unsigned char* der, word32 derSz) {

    // 拡張を解析するロジックがここに入ります。

    // 注: 0 を返すことで、この拡張を受け入れ、wolfSSL に
    // それが許容可能であることを通知しています。許容できない拡張が
    // 見つかった場合は、エラーを返す必要があります。重要フラグが
```

```

// 設定された不明な拡張に遭遇した場合の標準的な動作は、
// ASN_CRIT_EXT_E を返すことです。簡潔にするため、この例では
// 常にすべての拡張を受け入れています。異なるロジックを
// 使用する必要があります。
return 0;
}

```

B.1.2.54 function wc_CheckCertSigPubKey

```

int wc_CheckCertSigPubKey(
    const byte * cert,
    word32 certSz,
    void * heap,
    const byte * pubKey,
    word32 pubKeySz,
    int pubKeyOID
)

```

この関数は、X.509 証明書の der 形式の署名を公開鍵に対して検証します。公開鍵は、der 形式の完全なサブジェクト公開鍵情報であることが期待されます。

Parameters:

- **cert** X.509 証明書の der エンコーディング。
- **certSz** cert のサイズ (バイト単位)。
- **heap** 動的割り当てに使用されるヒープへのポインタ。NULL でも可。
- **pubKey** 公開鍵の der エンコーディング。
- **pubKeySz** pubKey のサイズ (バイト単位)。
- **pubKeyOID** 公開鍵のアルゴリズムを識別する OID。(例: ECDSAk、DSAk または RSAk)

Return:

- 0 成功時に返されます。
- Other 失敗時に負の値。

B.1.2.55 function wc_Asn1PrintOptions_Init

```

int wc_Asn1PrintOptions_Init(
    Asn1PrintOptions * opts
)

```

この関数は、ASN.1 プリントオプションを初期化します。

Parameters:

- **opts** プリント用の ASN.1 オプション。

See:

- [wc_Asn1PrintOptions_Set](#)
- [wc_Asn1_PrintAll](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG asn1 が NULL の場合。

Example

```
Asn1PrintOptions opt;
```

```
// 使用前に ASN.1 プリントオプションを初期化します。  
wc_Asn1PrintOptions_Init(&opt);
```

B.1.2.56 function wc_Asn1PrintOptions_Set

```
int wc_Asn1PrintOptions_Set(  
    Asn1PrintOptions * opts,  
    enum Asn1PrintOpt opt,  
    word32 val  
)
```

この関数は、ASN.1 プリントオプションオブジェクトにプリントオプションを設定します。

Parameters:

- **opts** プリント用の ASN.1 オプション。
- **opt** 値を設定するオプション。
- **val** 設定する値。

See:

- [wc_Asn1PrintOptions_Init](#)
- [wc_Asn1_PrintAll](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG asn1 が NULL の場合。
- BAD_FUNC_ARG val が option の範囲外の場合。

Example

```
Asn1PrintOptions opt;  
  
// 使用前に ASN.1 プリントオプションを初期化します。  
wc_Asn1PrintOptions_Init(&opt);  
// タグ名をプリントする際のインデント数を 1 に設定します。  
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);
```

B.1.2.57 function wc_Asn1_Init

```
int wc_Asn1_Init(  
    Asn1 * asn1  
)
```

この関数は、ASN.1 解析オブジェクトを初期化します。

Parameters:

- **asn1** ASN.1 解析オブジェクト。

See:

- [wc_Asn1_SetFile](#)
- [wc_Asn1_PrintAll](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG asn1 が NULL の場合。

Example

```
Asn1 asn1;
```

```
// 使用前に ASN.1 解析オブジェクトを初期化します。  
wc_Asn1_Init(&asn1);
```

B.1.2.58 function wc_Asn1_SetFile

```
int wc_Asn1_SetFile(  
    Asn1 * asn1,  
    XFILE file  
)
```

この関数は、ASN.1 解析オブジェクトへのプリント時に使用するファイルを設定します。

Parameters:

- **asn1** ASN.1 解析オブジェクト。
- **file** プリント先のファイル。

See:

- [wc_Asn1_Init](#)
- [wc_Asn1_PrintAll](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG asn1 が NULL の場合。
- BAD_FUNC_ARG file が XBADFILE の場合。

Example

```
Asn1 asn1;  
  
// 使用前に ASN.1 解析オブジェクトを初期化します。  
wc_Asn1_Init(&asn1);  
// 標準出力を書き込み先のファイル記述子として設定します。  
wc_Asn1_SetFile(&asn1, stdout);
```

B.1.2.59 function wc_Asn1_PrintAll

```
int wc_Asn1_PrintAll(  
    Asn1 * asn1,  
    Asn1PrintOptions * opts,  
    unsigned char * data,  
    word32 len  
)
```

すべての ASN.1 項目をプリントします。

Parameters:

- **asn1** ASN.1 解析オブジェクト。
- **opts** ASN.1 プリントオプション。
- **data** プリントする BER/DER データを含むバッファ。
- **len** プリントするデータの長さ (バイト単位)。

See:

- [wc_Asn1_Init](#)
- [wc_Asn1_SetFile](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG asn1 または opts が NULL の場合。
- ASN_LEN_E ASN.1 項目の長さが長すぎる場合。
- ASN_DEPTH_E 終了オフセットが無効な場合。
- ASN_PARSE_E ASN.1 項目のすべてが解析されなかった場合。

```

Asn1PrintOptions opts;
Asn1 asn1;
unsigned char data[] = { DER/BER データで初期化 };
word32 len = sizeof(data);

// 使用前に ASN.1 プリントオプションを初期化します。
wc_Asn1PrintOptions_Init(&opt);
// タグ名をプリントする際のインデント数を 1 に設定します。
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);

// 使用前に ASN.1 解析オブジェクトを初期化します。
wc_Asn1_Init(&asn1);
// 標準出力を書き込み先のファイル記述子として設定します。
wc_Asn1_SetFile(&asn1, stdout);
// 指定されたプリントオプションでバッファ内のすべての ASN.1 項目をプリントします。
wc_Asn1_PrintAll(&asn1, &opts, data, len);

```

B.2 ベースエンコーディング**B.2.1 Functions**

	Name
int	Base64_Decode (const byte * in, word32 inLen, byte * out, word32 * outLen) この関数は、指定された Base64 エンコードされた入力 in を復号し、結果を出力バッファ out に格納します。また、変数 outLen に出力バッファに書き込まれたサイズを設定します。
int	Base64_Encode (const byte * in, word32 inLen, byte * out, word32 * outLen) この関数は、指定された入力 in をエンコードし、Base64 エンコードされた結果を出力バッファ out に格納します。エスケープされた%0A 行末の代わりに、従来の'' 行末でデータを書き込みます。正常に完了すると、この関数は outLen を出力バッファに書き込まれたバイト数に設定します。
int	Base64_EncodeEsc (const byte * in, word32 inLen, byte * out, word32 * outLen) この関数は、指定された入力 in をエンコードし、Base64 エンコードされた結果を出力バッファ out に格納します。'' 行末の代わりに%0A エスケープされた行末でデータを書き込みます。正常に完了すると、この関数は outLen を出力バッファに書き込まれたバイト数に設定します。

	Name
int	Base64_Encode_NoNL (const byte * in, word32 inLen, byte * out, word32 * outLen) この関数は、指定された入力 in をエンコードし、Base64 エンコードされた結果を出力バッファ out に格納します。改行なしでデータを書き込みます。正常に完了すると、この関数は outLen を出力バッファに書き込まれたバイト数に設定します。
int	Base16_Decode (const byte * in, word32 inLen, byte * out, word32 * outLen) この関数は、指定された Base16 エンコードされた入力 in を復号し、結果を出力バッファ out に格納します。また、変数 outLen に出力バッファに書き込まれたサイズを設定します。
int	Base16_Encode (const byte * in, word32 inLen, byte * out, word32 * outLen) 入力を base16 出力にエンコードします。

B.2.2 Functions Documentation

B.2.2.1 function Base64_Decode

```
int Base64_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この関数は、指定された Base64 エンコードされた入力 in を復号し、結果を出力バッファ out に格納します。また、変数 outLen に出力バッファに書き込まれたサイズを設定します。

Parameters:

- **in** 復号する入力バッファへのポインタ
- **inLen** 復号する入力バッファの長さ
- **out** 復号されたメッセージを格納する出力バッファへのポインタ
- **outLen** 出力バッファの長さへのポインタ。関数呼び出しの最後に書き込まれたバイト数で更新されます

See:

- [Base64_Encode](#)
- [Base16_Decode](#)

Return:

- 0 Base64 エンコードされた入力の復号に成功した場合に返されます
- BAD_FUNC_ARG 出力バッファが復号された入力を格納するには小さすぎる場合に返されます
- ASN_INPUT_E 入力バッファ内の文字が Base64 の範囲 ([A-Za-z0-9+/=]) 外にある場合、または Base64 エンコードされた入力に無効な行末がある場合に返されます

Example

```
byte encoded[] = { // 復号するテキストを初期化 };
byte decoded[sizeof(encoded)];
// 少なくとも (sizeof(encoded) * 3 + 3) / 4 のスペースが必要
```

```
int outLen = sizeof(decoded);

if( Base64_Decode(encoded, sizeof(encoded), decoded, &outLen) != 0 ) {
    // 入力バッファの復号エラー
}
```

B.2.2.2 function Base64_Encode

```
int Base64_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この関数は、指定された入力 in をエンコードし、Base64 エンコードされた結果を出力バッファ out に格納します。エスケープされた%0A 行末の代わりに、従来の' ' 行末でデータを書き込みます。正常に完了すると、この関数は outLen を出力バッファに書き込まれたバイト数に設定します。

Parameters:

- **in** エンコードする入力バッファへのポインタ
- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを格納する出力バッファへのポインタ
- **outLen** エンコードされたメッセージを格納する出力バッファの長さへのポインタ

See:

- [Base64_EncodeEsc](#)
- [Base64_Decode](#)

Return:

- 0 Base64 エンコードされた入力の復号に成功した場合に返されます
- BAD_FUNC_ARG 出力バッファがエンコードされた入力を格納するには小さすぎる場合に返されます
- BUFFER_E エンコード中に出力バッファのスペースが不足した場合に返されます

Example

```
byte plain[] = { // エンコードするテキストを初期化 };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_Encode(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // 入力バッファのエンコードエラー
}
```

B.2.2.3 function Base64_EncodeEsc

```
int Base64_EncodeEsc(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この関数は、指定された入力 in をエンコードし、Base64 エンコードされた結果を出力バッファ out に格納します。' ' 行末の代わりに%0A エスケープされた行末でデータを書き込みます。正常に完了すると、この関数は outLen を出力バッファに書き込まれたバイト数に設定します。

Parameters:

- **in** エンコードする入力バッファへのポインタ
- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを格納する出力バッファへのポインタ
- **outLen** エンコードされたメッセージを格納する出力バッファの長さへのポインタ

See:

- [Base64_Encode](#)
- [Base64_Decode](#)

Return:

- 0 Base64 エンコードされた入力の復号に成功した場合に返されます
- BAD_FUNC_ARG 出力バッファがエンコードされた入力を格納するには小さすぎる場合に返されます
- BUFFER_E エンコード中に出力バッファのスペースが不足した場合に返されます
- ASN_INPUT_E 入力メッセージの復号処理中にエラーが発生した場合に返されます

Example

```
byte plain[] = { // エンコードするテキストを初期化 };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_EncodeEsc(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // 入力バッファのエンコードエラー
}
```

B.2.2.4 function Base64_Encode_NoNI

```
int Base64_Encode_NoNI(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この関数は、指定された入力 *in* をエンコードし、Base64 エンコードされた結果を出力バッファ *out* に格納します。改行なしでデータを書き込みます。正常に完了すると、この関数は *outLen* を出力バッファに書き込まれたバイト数に設定します。

Parameters:

- **in** エンコードする入力バッファへのポインタ
- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを格納する出力バッファへのポインタ
- **outLen** エンコードされたメッセージを格納する出力バッファの長さへのポインタ

See:

- [Base64_Encode](#)
- [Base64_Decode](#)

Return:

- 0 Base64 エンコードされた入力の復号に成功した場合に返されます
- BAD_FUNC_ARG 出力バッファがエンコードされた入力を格納するには小さすぎる場合に返されます
- BUFFER_E エンコード中に出力バッファのスペースが不足した場合に返されます
- ASN_INPUT_E 入力メッセージの復号処理中にエラーが発生した場合に返されます

Example

```

byte plain[] = { // エンコードするテキストを初期化 };
byte encoded[MAX_BUFFER_SIZE];
int outLen = sizeof(encoded);
if( Base64_Encode_NoNl(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // 入力バッファのエンコードエラー
}

```

B.2.2.5 function Base16_Decode

```

int Base16_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)

```

この関数は、指定された Base16 エンコードされた入力 in を復号し、結果を出力バッファ out に格納します。また、変数 outLen に出力バッファに書き込まれたサイズを設定します。

Parameters:

- **in** 復号する入力バッファへのポインタ
- **inLen** 復号する入力バッファの長さ
- **out** 復号されたメッセージを格納する出力バッファへのポインタ
- **outLen** 出力バッファの長さへのポインタ。関数呼び出しの最後に書き込まれたバイト数で更新されます

See:

- [Base64_Encode](#)
- [Base64_Decode](#)
- [Base16_Encode](#)

Return:

- 0 Base16 エンコードされた入力の復号に成功した場合に返されます
- BAD_FUNC_ARG 出力バッファが復号された入力を格納するには小さすぎる場合、または入力長が 2 の倍数でない場合に返されます
- ASN_INPUT_E 入力バッファ内の文字が Base16 の範囲 ([0-9A-F]) 外にある場合に返されます

Example

```

byte encoded[] = { // 復号するテキストを初期化 };
byte decoded[sizeof(encoded)];
int outLen = sizeof(decoded);

if( Base16_Decode(encoded, sizeof(encoded), decoded, &outLen) != 0 ) {
    // 入力バッファの復号エラー
}

```

B.2.2.6 function Base16_Encode

```

int Base16_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)

```

入力を base16 出力にエンコードします。

Parameters:

- **in** エンコードする入力バッファへのポインタ。
- **inLen** 入力バッファの長さ。
- **out** 出力バッファへのポインタ。
- **outLen** 出力バッファの長さ。エンコードされた出力の長さに設定されます。

See:

- [Base64_Encode](#)
- [Base64_Decode](#)
- [Base16_Decode](#)

Return:

- 0 成功
- BAD_FUNC_ARG in、out、または outLen が null の場合、または outLen が inLen の 2 倍プラス 1 未満の場合に返されます。

Example

```
byte in[] = { // エンコードする何かの内容 };
byte out[NECESSARY_OUTPUT_SIZE];
word32 outSz = sizeof(out);

if(Base16_Encode(in, sizeof(in), out, &outSz) != 0)
{
    // エンコードエラーを処理
}
```

B.3 圧縮

B.3.1 Functions

	Name
int	wc_Compress (byte * out, word32 outSz, const byte * in, word32 inSz, word32 flags) この関数は、ハフマン符号化を使用して指定された入力データを圧縮し、出力を out に格納します。出力バッファは入力バッファよりも大きくする必要がありますことに注意してください。圧縮が不可能な特定の入力が存在する場合でも、ルックアップテーブルが必要になるためです。出力バッファには srcSz + 0.1% + 12 を割り当てることが推奨されます。
int	wc_DeCompress (byte * out, word32 outSz, const byte * in, word32 inSz) この関数は、ハフマン符号化を使用して指定された圧縮データを解凍し、出力を out に格納します。

B.3.2 Functions Documentation

B.3.2.1 function wc_Compress

```
int wc_Compress(
    byte * out,
```

```

    word32 outSz,
    const byte * in,
    word32 inSz,
    word32 flags
)

```

この関数は、ハフマン符号化を使用して指定された入力データを圧縮し、出力を out に格納します。出力バッファは入力バッファよりも大きくする必要がありますことに注意してください。圧縮が不可能な特定の入力が存在する場合でも、ルックアップテーブルが必要になるためです。出力バッファには $\text{srcSz} + 0.1\% + 12$ を割り当てることが推奨されます。

Parameters:

- **out** 圧縮されたデータを格納する出力バッファへのポインタ
- **outSz** 格納に使用できる出力バッファのサイズ
- **in** 圧縮するメッセージを含むバッファへのポインタ
- **inSz** 圧縮する入力メッセージのサイズ
- **flags** 圧縮の動作を制御するフラグ。通常の解凍には 0 を使用します

See: [wc_DeCompress](#)

Return:

- 入力データの圧縮に成功した場合、出力バッファに格納されたバイト数を返します
- COMPRESS_INIT_E 圧縮用のストリームの初期化中にエラーが発生した場合に返されます
- COMPRESS_E 圧縮中にエラーが発生した場合に返されます

Example

```

byte message[] = { // 圧縮するテキストを初期化 };
byte compressed[(sizeof(message) + sizeof(message) * .001 + 12)];
// 少なくとも srcSz + .1% + 12 を推奨

if( wc_Compress(compressed, sizeof(compressed), message, sizeof(message),
0) != 0){
    // データの圧縮エラー
}

```

B.3.2.2 function wc_DeCompress

```

int wc_DeCompress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz
)

```

この関数は、ハフマン符号化を使用して指定された圧縮データを解凍し、出力を out に格納します。

Parameters:

- **out** 解凍されたデータを格納する出力バッファへのポインタ
- **outSz** 格納に使用できる出力バッファのサイズ
- **in** 解凍するメッセージを含むバッファへのポインタ
- **inSz** 解凍する入力メッセージのサイズ

See: [wc_Compress](#)

Return:

- Success 入力データの解凍に成功した場合、出力バッファに格納されたバイト数を返します

- COMPRESS_INIT_E: 圧縮用のストリームの初期化中にエラーが発生した場合に返されます
- COMPRESS_E: 圧縮中にエラーが発生した場合に返されます

Example

```
byte compressed[] = { // 圧縮されたメッセージを初期化 };
byte decompressed[MAX_MESSAGE_SIZE];

if( wc_DeCompress(decompressed, sizeof(decompressed),
compressed, sizeof(compressed)) != 0 ) {
    // データの解凍エラー
}
```

B.4 エラー報告**B.4.1 Functions**

	Name
void	wc_ErrorString (int err, char * buff) この関数は、特定のエラーコードに対するエラー文字列を指定されたバッファに格納します。
const char *	wc_GetErrorString (int error) この関数は、特定のエラーコードに対するエラー文字列を返します。

B.4.2 Functions Documentation**B.4.2.1 function wc_ErrorString**

```
void wc_ErrorString(
    int err,
    char * buff
)
```

この関数は、特定のエラーコードに対するエラー文字列を指定されたバッファに格納します。

Parameters:

- **error** 文字列を取得するエラーコード
- **buffer** エラー文字列を格納するバッファ。バッファは少なくとも WOLFSSL_MAX_ERROR_SZ(80 バイト) の長さである必要があります

See: [wc_GetErrorString](#)

Return: none 戻り値なし。

Example

```
char errorMsg[WOLFSSL_MAX_ERROR_SZ];
int err = wc_some_function();

if( err != 0 ) { // エラーが発生
    wc_ErrorString(err, errorMsg);
}
```

```
B.4.2.2 function wc_GetErrorString
const char * wc_GetErrorString(
    int error
)
```

この関数は、特定のエラーコードに対するエラー文字列を返します。

Parameters:

- **error** 文字列を取得するエラーコード

See: [wc_ErrorString](#)

Return: string エラーコードに対するエラー文字列を文字列リテラルとして返します。

Example

```
char * errorMsg;
int err = wc_some_function();

if( err != 0 ) { // エラーが発生
    errorMsg = wc_GetErrorString(err);
}
```

B.5 IoT-Safe モジュール

[More...](#)

B.5.1 Functions

	Name
int	wolfSSL_CTX_iotsafe_enable (WOLFSSL_CTX * ctx) この関数は、指定されたコンテキストで IoT-Safe サポートを有効にします。
int	wolfSSL_iotsafe_on (WOLFSSL * ssl, byte privkey_id, byte ecdh_keypair_slot, byte peer_pubkey_slot, byte peer_cert_slot) この関数は、IoT-Safe TLS コールバックを指定された SSL セッションに接続します。
int	**wolfSSL_iotsafe_on_ex と同等ですが、IoT-SAFE スロットの ID を参照渡しでき、ID フィールドの長さをパラメータ "id_size" で指定できる点異なります。
void	wolfIoTSafe_SetCSIM_read_cb (wolfSSL_IOTSafe_CSIM_read_cb rf)AT+CSIM コマンド用の読み取りコールバックを関連付けます。この入力関数は通常、モデムと通信する UART チャネルの読み取りイベントに関連付けられます。関連付けられる読み取りコールバックはグローバルで、同時に IoT-safe サポートを使用するすべてのコンテキストに対して変更されます。

	Name
void	wolfIoTSafe_SetCSIM_write_cb (wolfSSL_IoTSafe_CSIM_write_cb wf) AT+CSIM コマンド用の書き込みコールバックを関連付けます。この出力関数は通常、モデムと通信する UART チャンネルの書き込みイベントに関連付けられます。関連付けられる書き込みコールバックはグローバルで、同時に IoT-safe サポートを使用するすべてのコンテキストに対して変更されます。
int	wolfIoTSafe_GetRandom (unsigned char * out, word32 sz) IoT-Safe 関数 GetRandom を使用して、指定されたサイズのランダムバッファを生成します。この関数は wolfCrypt RNG オブジェクトによって自動的に使用されます。
int	wolfIoTSafe_GetCert (byte id, unsigned char * output, unsigned long sz) IoT-Safe アプレット上のファイルに保存されている証明書をインポートし、メモリ内にローカルに保存します。1 バイトのファイル ID フィールドで動作します。
int	** wolfIoTSafe_GetCert_ex と同等ですが、2 バイト以上のファイル ID で呼び出すことができます。
int	wc_iotsafe_ecc_import_public (ecc_key * key, byte key_id) IoT-Safe アプレットに保存されている ECC 256 ビット公開鍵を ecc_key オブジェクトにインポートします。
int	wc_iotsafe_ecc_export_public (ecc_key * key, byte key_id) ECC 256 ビット公開鍵を ecc_key オブジェクトから IoT-Safe アプレット内の書き込み可能な公開鍵スロットにエクスポートします。
int	** wc_iotsafe_ecc_import_public_ex と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。
int	wc_iotsafe_ecc_export_private (ecc_key * key, byte key_id) ECC 256 ビット鍵を ecc_key オブジェクトから IoT-Safe アプレット内の書き込み可能な秘密鍵スロットにエクスポートします。
int	** wc_iotsafe_ecc_export_private_ex と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。
int	wc_iotsafe_ecc_sign_hash (byte * in, word32 inlen, byte * out, word32 * outlen, byte key_id) IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた秘密鍵を使用して、事前計算されたハッシュに署名します。
int	** wc_iotsafe_ecc_sign_hash_ex と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。

	Name
int	wc_iotsafe_ecc_verify_hash (byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte key_id)IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた公開鍵を使用して、事前計算されたハッシュに対する ECC 署名を検証します。結果は res に書き込まれます。1 は有効、0 は無効です。注意: 有効性をテストするために戻り値を使用しないでください。res のみを使用してください。
int	** wc_iotsafe_ecc_verify_hash_ex と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。
int	wc_iotsafe_ecc_gen_k (byte key_id)ECC 256 ビット鍵ペアを生成し、IoT-Safe アプレット内の(書き込み可能な) スロットに保存します。

B.5.2 Detailed Description

IoT-Safe(IoT-SIM Applet For Secure End-2-End Communication) は、SIM を堅牢で スケーラブルかつ標準化されたハードウェア Root of Trust として活用し、データ通信を保護する技術です。

IoT-Safe SSL セッションは、SIM をハードウェアセキュリティモジュールとして使用し、すべての暗号公開鍵操作をオフロードし、証明書と鍵へのアクセスを SIM に制限することで攻撃対象領域を削減します。

IoT-Safe サポートは、wolfSSL_CTX_iotsafe_enable() を使用して既存の WOLFSSL_CTX コンテキストで有効にできます。

コンテキスト内で作成されたセッションは、IoT-Safe 鍵とファイル使用のパラメータを設定し、**wolfSSL_iotsafe_on()**で公開鍵コールバックを有効にできます。

コンパイルされている場合、モジュールは wolfCrypt のエントロピーソースとして IoT-Safe 乱数生成器をサポートします。

B.5.3 Functions Documentation

B.5.3.1 function wolfSSL_CTX_iotsafe_enable

```
int wolfSSL_CTX_iotsafe_enable(
    WOLFSSL_CTX * ctx
)
```

この関数は、指定されたコンテキストで IoT-Safe サポートを有効にします。

Parameters:

- **ctx** IoT-safe サポートを有効にする必要がある WOLFSSL_CTX オブジェクトへのポインタ

See:

- **wolfSSL_iotsafe_on**
- **wolfIoTSafe_SetCSIM_read_cb**
- **wolfIoTSafe_SetCSIM_write_cb**

Return:

- 0 成功時
- WC_HW_E ハードウェアエラー時

Example

```
WOLFSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
if (!ctx)
    return NULL;
wolfSSL_CTX_iotsafe_enable(ctx);
```

B.5.3.2 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
)
```

この関数は、IoT-Safe TLS コールバックを指定された SSL セッションに接続します。

Parameters:

- **ssl** コールバックが有効になる WOLFSSL オブジェクトへのポインタ
- **privkey_id** ホストの秘密鍵を含む iot-safe アプレットスロットの ID
- **ecdh_keypair_slot** ECDH 鍵ペアを格納する iot-safe アプレットスロットの ID
- **peer_pubkey_slot** ECDH 用に他のエンドポイントの公開鍵を格納する iot-safe アプレットスロットの ID
- **peer_cert_slot** 検証用に他のエンドポイントの公開鍵を格納する iot-safe アプレットスロットの ID

See:

- [wolfSSL_iotsafe_on_ex](#)
- [wolfSSL_CTX_iotsafe_enable](#)

Return:

- 0 成功時
- NOT_COMPILED_IN HAVE_PK_CALLBACKS が無効な場合
- BAD_FUNC_ARG ssl ポインタが無効な場合

これは、スロットの ID が 1 バイト長の場合に SSL セッションを IoT-Safe アプレットに接続するために呼び出す必要があります。IoT-SAFE スロットの ID が 2 バイト以上の場合は、代わりに [wolfSSL_iotsafe_on_ex\(\)](#) を使用する必要があります。

Example

```
// IoT-Safe 用の鍵 ID を定義
#define PRIVKEY_ID 0x02
#define ECDH_KEYPAIR_ID 0x03
#define PEER_PUBKEY_ID 0x04
#define PEER_CERT_ID 0x05
// 新しい ssl セッションを作成
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// IoT-Safe を有効にして鍵スロットを関連付け
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);
}
```

B.5.3.3 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
)
```

この関数は、IoT-Safe TLS コールバックを指定された SSL セッションに接続します。これは `wolfSSL_iotsafe_on` と同等ですが、IoT-SAFE スロットの ID を参照渡しでき、ID フィールドの長さをパラメータ "id_size" で指定できる点が異なります。

Parameters:

- **ssl** コールバックが有効になる WOLFSSL オブジェクトへのポインタ
- **privkey_id** ホストの秘密鍵を含む iot-safe アプレットスロットの ID へのポインタ
- **ecdh_keypair_slot** ECDH 鍵ペアを格納する iot-safe アプレットスロットの ID へのポインタ
- **peer_pubkey_slot** ECDH 用に他のエンドポイントの公開鍵を格納する iot-safe アプレットスロットの ID へのポインタ
- **peer_cert_slot** 検証用に他のエンドポイントの公開鍵を格納する iot-safe アプレットスロットの ID へのポインタ
- **id_size** 各スロット ID のサイズ

See:

- `wolfSSL_iotsafe_on`
- `wolfSSL_CTX_iotsafe_enable`

Return:

- 0 成功時
- NOT_COMPILED_IN HAVE_PK_CALLBACKS が無効な場合
- BAD_FUNC_ARG ssl ポインタが無効な場合

Example

```
// IoT-Safe 用の鍵 ID を定義 (16 ビット、リトルエンディアン)
#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;
```

```
// 新しい ssl セッションを作成
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// IoT-Safe を有効にして鍵スロットを関連付け
```

```
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);
}
```

B.5.3.4 function wolfIoTSafe_SetCSIM_read_cb

```
void wolfIoTSafe_SetCSIM_read_cb(
    wolfSSL_IoTSafe_CSIM_read_cb rf
)
```

AT+CSIM コマンド用の読み取りコールバックを関連付けます。この入力関数は通常、モデムと通信する UART チャンネルの読み取りイベントに関連付けられます。関連付けられる読み取りコールバックはグローバルで、同時に IoT-safe サポートを使用するすべてのコンテキストに対して変更されます。

Parameters:

- **rf** UART 読み取りイベントに関連付けられる読み取りコールバック。コールバック関数は 2 つの引数 (buf, len) を受け取り、len まで読み取られた文字数を返します。改行文字に遭遇すると、コールバックはこれまでに受信した文字数 (改行文字を含む) を返す必要があります。

See: [wolfIoTSafe_SetCSIM_write_cb](#)

Example

```
// USART 読み取り関数、他の場所で定義
int usart_read(char *buf, int len);

wolfIoTSafe_SetCSIM_read_cb(usart_read);
```

B.5.3.5 function wolfIoTSafe_SetCSIM_write_cb

```
void wolfIoTSafe_SetCSIM_write_cb(
    wolfSSL_IoTSafe_CSIM_write_cb wf
)
```

AT+CSIM コマンド用の書き込みコールバックを関連付けます。この出力関数は通常、モデムと通信する UART チャンネルの書き込みイベントに関連付けられます。関連付けられる書き込みコールバックはグローバルで、同時に IoT-safe サポートを使用するすべてのコンテキストに対して変更されます。

Parameters:

- **rf** UART 書き込みイベントに関連付けられる書き込みコールバック。コールバック関数は 2 つの引数 (buf, len) を受け取り、len まで書き込まれた文字数を返します。

See: [wolfIoTSafe_SetCSIM_read_cb](#)

Example

```
// USART 書き込み関数、他の場所で定義
int usart_write(const char *buf, int len);
wolfIoTSafe_SetCSIM_write_cb(usart_write);
```

B.5.3.6 function wolfIoTSafe_GetRandom

```
int wolfIoTSafe_GetRandom(
    unsigned char * out,
    word32 sz
)
```

IoT-Safe 関数 `GetRandom` を使用して、指定されたサイズのランダムバッファを生成します。この関数は `wolfCrypt RNG` オブジェクトによって自動的に使用されます。

Parameters:

- **out** ランダムなバイトシーケンスが格納されるバッファ。
- **sz** 生成するランダムシーケンスのサイズ (バイト単位)

Return: 0 成功時

B.5.3.7 function `wolfIoTSafe_GetCert`

```
int wolfIoTSafe_GetCert(
    byte id,
    unsigned char * output,
    unsigned long sz
)
```

IoT-Safe アプレット上のファイルに保存されている証明書をインポートし、メモリ内にローカルに保存します。1 バイトのファイル ID フィールドで動作します。

Parameters:

- **id** 証明書が保存されている IoT-Safe アプレット内のファイル ID
- **output** 証明書がインポートされるバッファ
- **sz** バッファ `output` で利用可能な最大サイズ

Return:

- インポートされた証明書の長さ
- < 0 失敗の場合

Example

```
#define CRT_CLIENT_FILE_ID 0x03
unsigned char cert_buffer[2048];
// 証明書をバッファに取得
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);

// TLS クライアントコンテキストのアイデンティティとして証明書バッファを使用
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");
```

B.5.3.8 function `wolfIoTSafe_GetCert_ex`

```
int wolfIoTSafe_GetCert_ex(
    uint8_t * id,
    uint16_t id_sz,
    unsigned char * output,
    unsigned long sz
)
```

)

IoT-Safe アプレット上のファイルに保存されている証明書をインポートし、メモリ内にローカルに保存します。wolfIoTSafe_GetCertと同等ですが、2 バイト以上のファイル ID で呼び出すことができます。

Parameters:

- **id** 証明書が保存されている IoT-Safe アプレット内のファイル ID へのポインタ
- **id_sz** ファイル ID のサイズ
- **output** 証明書がインポートされるバッファ
- **sz** バッファ output で利用可能な最大サイズ

Return:

- インポートされた証明書の長さ
- < 0 失敗の場合

Example

B.5.3.9 function wc_iotsafe_ecc_import_public

```
int wc_iotsafe_ecc_import_public(  
    ecc_key * key,  
    byte key_id  
)
```

IoT-Safe アプレットに保存されている ECC 256 ビット公開鍵を ecc_key オブジェクトにインポートします。

Parameters:

- **key** IoT-Safe アプレットからインポートされた鍵を含む ecc_key オブジェクト
- **id** 公開鍵が保存されている IoT-Safe アプレット内の鍵 ID

See:

- [wc_iotsafe_ecc_export_public](#)
- [wc_iotsafe_ecc_export_private](#)

Return:

- 0 成功時
- < 0 失敗の場合

B.5.3.10 function wc_iotsafe_ecc_export_public

```
int wc_iotsafe_ecc_export_public(  
    ecc_key * key,  
    byte key_id  
)
```

ECC 256 ビット公開鍵を ecc_key オブジェクトから IoT-Safe アプレット内の書き込み可能な公開鍵スロットにエクスポートします。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 公開鍵が保存される IoT-Safe アプレット内の鍵 ID

See:

- [wc_iotsafe_ecc_import_public_ex](#)

- [wc_iotsafe_ecc_export_private](#)

Return:

- 0 成功時
- < 0 失敗の場合

B.5.3.11 function wc_iotsafe_ecc_import_public_ex

```
int wc_iotsafe_ecc_import_public_ex(  
    ecc_key * key,  
    byte * key_id,  
    word16 id_size  
)
```

ECC 256 ビット公開鍵を ecc_key オブジェクトから IoT-Safe アプレット内の書き込み可能な公開鍵スロットにエクスポートします。[wc_iotsafe_ecc_import_public](#)と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 公開鍵が保存される IoT-Safe アプレット内の鍵 ID へのポインタ
- **id_size** 鍵 ID のサイズ

See:

- [wc_iotsafe_ecc_import_public](#)
- [wc_iotsafe_ecc_export_private](#)

Return:

- 0 成功時
- < 0 失敗の場合

B.5.3.12 function wc_iotsafe_ecc_export_private

```
int wc_iotsafe_ecc_export_private(  
    ecc_key * key,  
    byte key_id  
)
```

ECC 256 ビット鍵を ecc_key オブジェクトから IoT-Safe アプレット内の書き込み可能な秘密鍵スロットにエクスポートします。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 秘密鍵が保存される IoT-Safe アプレット内の鍵 ID

See:

- [wc_iotsafe_ecc_export_private_ex](#)
- [wc_iotsafe_ecc_import_public](#)
- [wc_iotsafe_ecc_export_public](#)

Return:

- 0 成功時
- < 0 失敗の場合

B.5.3.13 function wc_iotsafe_ecc_export_private_ex

```
int wc_iotsafe_ecc_export_private_ex(  
    ecc_key * key,  
    byte * key_id,  
    word16 id_size  
)
```

ECC 256 ビット鍵を `ecc_key` オブジェクトから IoT-Safe アプレット内の書き込み可能な秘密鍵スロットにエクスポートします。`wc_iotsafe_ecc_export_private` と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。

Parameters:

- **key** エクスポートする鍵を含む `ecc_key` オブジェクト
- **id** 秘密鍵が保存される IoT-Safe アプレット内の鍵 ID へのポインタ
- **id_size** 鍵 ID のサイズ

See:

- `wc_iotsafe_ecc_export_private`
- `wc_iotsafe_ecc_import_public`
- `wc_iotsafe_ecc_export_public`

Return:

- 0 成功時
- < 0 失敗の場合

B.5.3.14 function wc_iotsafe_ecc_sign_hash

```
int wc_iotsafe_ecc_sign_hash(  
    byte * in,  
    word32 inlen,  
    byte * out,  
    word32 * outlen,  
    byte key_id  
)
```

IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた秘密鍵を使用して、事前計算されたハッシュに署名します。

Parameters:

- **in** 署名するメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージハッシュの長さ
- **out** 生成された署名を格納するバッファ
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功すると、out に書き込まれたバイト数を格納します
- **id** ペイロードに署名する秘密鍵を含むスロットの IoT-Safe アプレット内の鍵 ID

See:

- `wc_iotsafe_ecc_sign_hash_ex`
- `wc_iotsafe_ecc_verify_hash`
- `wc_iotsafe_ecc_gen_k`

Return:

- 0 成功時
- < 0 失敗の場合

B.5.3.15 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  
)
```

IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた秘密鍵を使用して、事前計算されたハッシュに署名します。[wc_iotsafe_ecc_sign_hash](#)と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。

Parameters:

- **in** 署名するメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージハッシュの長さ
- **out** 生成された署名を格納するバッファ
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功すると、out に書き込まれたバイト数を格納します
- **id** ペイロードに署名する秘密鍵を含むスロットの IoT-Safe アプレット内の鍵 ID へのポインタ
- **id_size** 鍵 ID のサイズ

See:

- [wc_iotsafe_ecc_sign_hash](#)
- [wc_iotsafe_ecc_verify_hash](#)
- [wc_iotsafe_ecc_gen_k](#)

Return:

- 0 成功時
- < 0 失敗の場合

B.5.3.16 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  
)
```

IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた公開鍵を使用して、事前計算されたハッシュに対する ECC 署名を検証します。結果は res に書き込まれます。1 は有効、0 は無効です。注意: 有効性をテストするために戻り値を使用しないでください。res のみを使用してください。

Parameters:

- **sig** 検証する署名を含むバッファ
- **hash** 署名されたハッシュ (メッセージダイジェスト)
- **hashlen** ハッシュの長さ (オクテット)
- **res** 署名の結果、1 == 有効、0 == 無効
- **key_id** IoT-Safe アプレット内に公開 ECC 鍵が保存されているスロットの ID

See:

- [wc_iotsafe_ecc_verify_hash_ex](#)
- [wc_iotsafe_ecc_sign_hash](#)
- [wc_iotsafe_ecc_gen_k](#)

Return:

- 0 成功時 (署名が有効でない場合でも)
- < 0 失敗の場合。

B.5.3.17 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  
)
```

IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた公開鍵を使用して、事前計算されたハッシュに対する ECC 署名を検証します。結果は `res` に書き込まれます。1 は有効、0 は無効です。注意: 有効性をテストするために戻り値を使用しないでください。res のみを使用してください。[wc_iotsafe_ecc_verify_hash](#) と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。

Parameters:

- **sig** 検証する署名を含むバッファ
- **hash** 署名されたハッシュ (メッセージダイジェスト)
- **hashlen** ハッシュの長さ (オクテット)
- **res** 署名の結果、1 == 有効、0 == 無効
- **key_id** IoT-Safe アプレット内に公開 ECC 鍵が保存されているスロットの ID
- **id_size** 鍵 ID のサイズ

See:

- [wc_iotsafe_ecc_verify_hash](#)
- [wc_iotsafe_ecc_sign_hash](#)
- [wc_iotsafe_ecc_gen_k](#)

Return:

- 0 成功時 (署名が有効でない場合でも)
- < 0 失敗の場合。

B.5.3.18 function wc_iotsafe_ecc_gen_k

```
int wc_iotsafe_ecc_gen_k(  
    byte key_id  
)
```

ECC 256 ビット鍵ペアを生成し、IoT-Safe アプレット内の (書き込み可能な) スロットに保存します。

Parameters:

- **key_id** IoT-Safe アプレット内に ECC 鍵ペアが保存されているスロットの ID。
- **key_id** IoT-Safe アプレット内に ECC 鍵ペアが保存されているスロットの ID。
- **id_size** 鍵 ID のサイズ

See:

- `wc_iotsafe_ecc_gen_k_ex`
- `wc_iotsafe_ecc_sign_hash`
- `wc_iotsafe_ecc_verify_hash`
- `wc_iotsafe_ecc_gen_k`
- `wc_iotsafe_ecc_sign_hash_ex`
- `wc_iotsafe_ecc_verify_hash_ex`

Return:

- 0 成功時
- < 0 失敗の場合。
- 0 成功時
- < 0 失敗の場合。

ECC 256 ビット鍵ペアを生成し、IoT-Safe アプレット内の (書き込み可能な) スロットに保存します。
`wc_iotsafe_ecc_gen_k`と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。

B.6 ログイン

B.6.1 Functions

	Name
int	<code>wolfSSL_SetLoggingCb</code> (<code>wolfSSL_Logging_cb log_function</code>) この関数は、wolfSSL ログメッセージを処理するために使用されるログインコールバックを登録します。デフォルトでは、システムがサポートしている場合、stderr への <code>fprintf()</code> が使用されますが、この関数を使用することでユーザーは任意の処理を行うことができます。

B.6.2 Functions Documentation

B.6.2.1 function `wolfSSL_SetLoggingCb`

```
int wolfSSL_SetLoggingCb(
    wolfSSL_Logging_cb log_function
)
```

この関数は、wolfSSL ログメッセージを処理するために使用されるログインコールバックを登録します。デフォルトでは、システムがサポートしている場合、stderr への `fprintf()` が使用されますが、この関数を使用することでユーザーは任意の処理を行うことができます。

Parameters:

- **`log_function`** ログインコールバックとして登録する関数。関数シグネチャは上記のプロトタイプに従う必要があります。

See:

- `wolfSSL_Debugging_ON`
- `wolfSSL_Debugging_OFF`

Return:

- Success 成功した場合、この関数は 0 を返します。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラーです。

Example

```

int ret = 0;
// ロギングコールバックのプロトタイプ
void MyLoggingCallback(const int logLevel, const char* const logMessage);
// カスタムロギングコールバックを wolfSSL に登録
ret = wolfSSL_SetLoggingCb(MyLoggingCallback);
if (ret != 0) {
    // ロギングコールバックの設定に失敗
}
void MyLoggingCallback(const int logLevel, const char* const logMessage)
{
    // カスタムロギング関数
}

```

B.7 整数演算 API

B.7.1 Functions

	Name
word32	CheckRunTimeFastMath (void) この関数は、整数の最大サイズに関するランタイム fastmath 設定をチェックします。ユーザーが wolfCrypt ライブラリを独立して使用している場合に重要です。数学が正しく動作するためには、各ライブラリの FP_SIZE が一致している必要があります。このチェックは CheckFastMathSettings() として定義されており、CheckRunTimeFastMath と FP_SIZE を単純に比較し、不一致の場合は 0 を、一致する場合は 1 を返します。
word32	CheckRunTimeSettings (void) この関数は、コンパイル時のクラス設定をチェックします。ユーザーが wolfCrypt ライブラリを独立して使用している場合に重要です。数学が正しく動作するためには、ライブラリ間で設定が一致している必要があります。このチェックは CheckCtcSettings() として定義されており、CheckRunTimeSettings と CTC_SETTINGS を単純に比較し、不一致の場合は 0 を、一致する場合は 1 を返します。

B.7.2 Functions Documentation

B.7.2.1 function CheckRunTimeFastMath

```

word32 CheckRunTimeFastMath(
    void
)

```

この関数は、整数の最大サイズに関するランタイム fastmath 設定をチェックします。ユーザーが wolfCrypt ライブラリを独立して使用している場合に重要です。数学が正しく動作するためには、各ライブラリの FP_SIZE が一致している必要があります。このチェックは CheckFastMathSettings() として定義されており、CheckRunTimeFastMath と FP_SIZE を単純に比較し、不一致の場合は 0 を、一致する場合は 1 を返します。

Parameters:

- **none** パラメータなし。

See: [CheckRunTimeSettings](#)

Return: FP_SIZE 数学ライブラリで使用可能な最大サイズに対応する FP_SIZE を返します。

Example

```
if (CheckFastMathSettings() != 1) {
    return err_sys("Build vs. runtime fastmath FP_MAX_BITS mismatch\n");
}
// これはプリプロセッサによって次のように変換されます:
// if ( (CheckRunTimeFastMath() == FP_SIZE) != 1) {
// そして fastmath 設定がコンパイル時の設定と一致することを確認します
```

B.7.2.2 function CheckRunTimeSettings

```
word32 CheckRunTimeSettings(
    void
)
```

この関数は、コンパイル時のクラス設定をチェックします。ユーザーが wolfCrypt ライブラリを独立して使用している場合に重要です。数学が正しく動作するためには、ライブラリ間で設定が一致している必要があります。このチェックは CheckCtcSettings() として定義されており、CheckRunTimeSettings と CTC_SETTINGS を単純に比較し、不一致の場合は 0 を、一致する場合は 1 を返します。

Parameters:

- **none** パラメータなし。

See: [CheckRunTimeFastMath](#)

Return: settings ランタイム CTC_SETTINGS(コンパイル時設定) を返します

Example

```
if (CheckCtcSettings() != 1) {
    return err_sys("Build vs. runtime math mismatch\n");
}
// これはプリプロセッサによって次のように変換されます:
// if ( (CheckCtcSettings() == CTC_SETTINGS) != 1) {
// そしてコンパイル時のクラス設定が現在の設定と一致するかどうかを比較します
```

B.8 乱数生成

B.8.1 Functions

	Name
int	wc_InitNetRandom (const char * configFile, wnr_hmac_key hmac_cb, int timeout) グローバル Whitewood netRandom コンテキストを初期化します
int	wc_FreeNetRandom (void) グローバル Whitewood netRandom コンテキストを解放します。

	Name
int	wc_InitRng (WC_RNG * rng) rng 用のシード (OS から) と鍵暗号を取得します。rng->drbg(決定論的乱数ビット生成器) が割り当てられます (wc_FreeRng で割り当て解除する必要があります)。これはブロッキング操作です。
int	wc_RNG_GenerateBlock (WC_RNG * rng, byte * b, word32 sz) 疑似乱数データの sz バイトを output にコピーします。必要に応じて rng を再シードします (ブロッキング)。
WC_RNG *	wc_rng_new (byte * nonce, word32 nonceSz, void * heap) 新しい WC_RNG 構造体を作成します。
int	wc_FreeRng (WC_RNG * rng) drbg を安全に解放するために、RNG が不要になったときに呼び出す必要があります。rng->drbg をゼロ化し XFREE します。
WC_RNG *	wc_rng_free (WC_RNG * rng) rng を安全に解放するために、RNG が不要になったときに呼び出す必要があります。
int	wc_RNG_HealthTest (int reseed, const byte * seedA, word32 seedASz, const byte * seedB, word32 seedBSz, byte * output, word32 outputSz) drbg の機能を作成してテストします。

B.8.2 Functions Documentation

B.8.2.1 function wc_InitNetRandom

```
int wc_InitNetRandom(
    const char * configFile,
    wnr_hmac_key hmac_cb,
    int timeout
)
```

グローバル Whitewood netRandom コンテキストを初期化します

Parameters:

- **configFile** 設定ファイルへのパス
- **hmac_cb** HMAC コールバックを作成するためのオプション。
- **timeout** タイムアウト期間。

See: [wc_FreeNetRandom](#)

Return:

- 0 成功
- BAD_FUNC_ARG configFile が null または timeout が負の値の場合。
- RNG_FAILURE_E rng の初期化に失敗しました。

Example

```
char* config = "path/to/config/example.conf";
int time = // 十分なタイムアウト値;

if (wc_InitNetRandom(config, NULL, time) != 0)
```

```
{
    // エラーが発生しました
}
```

B.8.2.2 function wc_FreeNetRandom

```
int wc_FreeNetRandom(
    void
)
```

グローバル Whitewood netRandom コンテキストを解放します。

Parameters:

- **none** 戻り値なし。

See: [wc_InitNetRandom](#)

Return:

- 0 成功
- BAD_MUTEX_E wnr_mutex のミューテックスロックエラー

Example

```
int ret = wc_FreeNetRandom();
if (ret != 0)
{
    // エラーを処理
}
```

B.8.2.3 function wc_InitRng

```
int wc_InitRng(
    WC_RNG * rng
)
```

rng 用のシード (OS から) と鍵暗号を取得します。rng->drbg(決定論的乱数ビット生成器) が割り当てられます (wc_FreeRng で割り当て解除する必要があります)。これはブロッキング操作です。

Parameters:

- **rng** シードと鍵暗号で使用するために初期化される乱数生成器

See:

- wc_InitRngCavium
- [wc_RNG_GenerateBlock](#)
- wc_RNG_GenerateByte
- [wc_FreeRng](#)
- [wc_RNG_HealthTest](#)

Return:

- 0 成功時。
- MEMORY_E XMALLOC が失敗しました
- WINCRYPT_E wc_GenerateSeed: コンテキストの取得に失敗しました
- CRYPTGEN_E wc_GenerateSeed: ランダムの取得に失敗しました
- BAD_FUNC_ARG wc_RNG_GenerateBlock 入力 が null または sz が MAX_REQUEST_LEN を超えています
- DRBG_CONT_FIPS_E wc_RNG_GenerateBlock: Hash_gen が DRBG_CONT_FAILURE を返しました

- `RNG_FAILURE_E wc_RNG_GenerateBlock`: デフォルトエラー。rng のステータスが元々 ok でないか、`DRBG_FAILED` に設定されています

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;
}

```

B.8.2.4 function wc_RNG_GenerateBlock

```

int wc_RNG_GenerateBlock(
    WC_RNG * rng,
    byte * b,
    word32 sz
)

```

疑似乱数データの `sz` バイトを `output` にコピーします。必要に応じて `rng` を再シードします (ブロッキング)。

Parameters:

- **rng** `wc_InitRng` で初期化された乱数生成器
- **output** ブロックがコピーされるバッファ
- **sz** 出力のサイズ (バイト単位)

See:

- `wc_InitRngCavium`, `wc_InitRng`
- `wc_RNG_GenerateByte`
- `wc_FreeRng`
- `wc_RNG_HealthTest`

Return:

- 0 成功時
- `BAD_FUNC_ARG` 入力が null または `sz` が `MAX_REQUEST_LEN` を超えています
- `DRBG_CONT_FIPS_E Hash_gen` が `DRBG_CONT_FAILURE` を返しました
- `RNG_FAILURE_E` デフォルトエラー。rng のステータスが元々 ok でないか、`DRBG_FAILED` に設定されています

Example

```

RNG  rng;
int  sz = 32;
byte block[sz];

int  ret = wc_InitRng(&rng);

```

```

if (ret != 0) {
    return -1; //rng の初期化失敗!
}

ret = wc_RNG_GenerateBlock(&rng, block, sz);
if (ret != 0) {
    return -1; //ブロック生成失敗!
}

```

B.8.2.5 function wc_rng_new

```

WC_RNG * wc_rng_new(
    byte * nonce,
    word32 nonceSz,
    void * heap
)

```

新しい WC_RNG 構造体を作成します。

Parameters:

- **heap** ヒープ識別子へのポインタ
- **nonce** nonce を含むバッファへのポインタ
- **nonceSz** nonce の長さ
- **rng** wc_InitRng で初期化された乱数生成器
- **b** ブロックがコピーされる 1 バイトのバッファ

See:

- [wc_InitRng](#)
- [wc_rng_free](#)
- [wc_FreeRng](#)
- [wc_RNG_HealthTest](#)
- [wc_InitRngCavium](#)
- [wc_InitRng](#)
- [wc_RNG_GenerateBlock](#)
- [wc_FreeRng](#)
- [wc_RNG_HealthTest](#)

Return:

- WC_RNG 成功時の構造体
- NULL エラー時
- 0 成功時
- BAD_FUNC_ARG 入力が null または sz が MAX_REQUEST_LEN を超えています
- DRBG_CONT_FIPS_E Hash_gen が DRBG_CONT_FAILURE を返しました
- RNG_FAILURE_E デフォルトエラー。rng のステータスが元々 ok でないか、DRBG_FAILED に設定されています

Example

```

RNG rng;
byte nonce[] = { nonce を初期化 };
word32 nonceSz = sizeof(nonce);

wc_rng_new(&nonce, nonceSz, &heap);

```

疑似乱数データの 1 バイトを b にコピーするために wc_RNG_GenerateBlock を呼び出します。必要に応じて rng を再シードします。

Example

```

RNG rng;
int sz = 32;
byte b[1];

int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //rng の初期化失敗!
}

ret = wc_RNG_GenerateByte(&rng, b);
if (ret != 0) {
    return -1; //ブロック生成失敗!
}

```

B.8.2.6 function wc_FreeRng

```

int wc_FreeRng(
    WC_RNG * rng
)

```

drbg を安全に解放するために、RNG が不要になったときに呼び出す必要があります。rng-drbg をゼロ化し XFREE します。

Parameters:

- **rng** wc_InitRng で初期化された乱数生成器

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte,
- wc_RNG_HealthTest

Return:

- 0 成功時
- BAD_FUNC_ARG rng または rng->drbg が null
- RNG_FAILURE_E drbg の割り当て解除に失敗しました

Example

```

RNG rng;
int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //rng の初期化失敗!
}

int ret = wc_FreeRng(&rng);
if (ret != 0) {
    return -1; //rng の解放失敗!
}

```

B.8.2.7 function wc_rng_free

```

WC_RNG * wc_rng_free(
    WC_RNG * rng
)

```

)

rng を安全に解放するために、RNG が不要になったときに呼び出す必要があります。

Parameters:

- **rng** wc_InitRng で初期化された乱数生成器

See:

- [wc_InitRng](#)
- [wc_rng_new](#)
- [wc_FreeRng](#)
- [wc_RNG_HealthTest](#)

Example

```
RNG rng;
byte nonce[] = { nonce を初期化 };
word32 nonceSz = sizeof(nonce);

rng = wc_rng_new(&nonce, nonceSz, &heap);

// rng を使用

wc_rng_free(&rng);
```

B.8.2.8 function wc_RNG_HealthTest

```
int wc_RNG_HealthTest(
    int reseed,
    const byte * seedA,
    word32 seedASz,
    const byte * seedB,
    word32 seedBSz,
    byte * output,
    word32 outputSz
)
```

drbg の機能を作成してテストします。

Parameters:

- **int reseed:** 設定されている場合、再シード機能をテストします
- **seedA** drbg をインスタンス化するシード
- **seedASz** seedA のサイズ (バイト単位)
- **seedB** reseed が設定されている場合、drbg は seedB で再シードされます
- **seedBSz** seedB のサイズ (バイト単位)
- **output** seedrandom が設定されている場合は seedB でシードされたランダムデータに初期化され、それ以外の場合は seedA でシードされます
- **outputSz** output の長さ (バイト単位)

See:

- [wc_InitRngCavium](#)
- [wc_InitRng](#)
- [wc_RNG_GenerateBlock](#)
- [wc_RNG_GenerateByte](#)
- [wc_FreeRng](#)

Return:

- 0 成功時
- BAD_FUNC_ARG seedA と output は null であってはいけません。reseed が設定されている場合、seedB は null であってはいけません
- -1 テスト失敗

Example

```

byte output[SHA256_DIGEST_SIZE * 4];
const byte test1EntropyB[] = ....; // reseed false のテスト入力
const byte test1Output[] = ....; // テストベクター: reseed false の期待出力
ret = wc_RNG_HealthTest(0, test1Entropy, sizeof(test1Entropy), NULL, 0,
                        output, sizeof(output));
if (ret != 0)
    return -1; //再シードなしのヘルステスト失敗

if (XMEMCMP(test1Output, output, sizeof(output)) != 0)
    return -1; //テストベクターとの比較失敗: 予期しない出力

const byte test2EntropyB[] = ....; // reseed のテスト入力
const byte test2Output[] = ....; // テストベクター reseed の期待出力
ret = wc_RNG_HealthTest(1, test2EntropyA, sizeof(test2EntropyA),
                        test2EntropyB, sizeof(test2EntropyB),
                        output, sizeof(output));

if (XMEMCMP(test2Output, output, sizeof(output)) != 0)
    return -1; //テストベクターとの比較失敗

```

B.9 署名 API**B.9.1 Functions**

	Name
int	wc_SignatureGetSize (enum wc_SignatureType sig_type, const void * key, word32 key_len) この関数は、結果の署名の最大サイズを返します。
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, const void * key, word32 key_len) この関数は、データをハッシュ化し、結果のハッシュとキーを使用して署名を検証することで、署名を検証します。
int	wc_SignatureGenerate (enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, byte * sig, word32 * sig_len, const void * key, word32 key_len, WC_RNG * rng) この関数は、キーを使用してデータから署名を生成します。最初にデータのハッシュを作成し、次にキーを使用してハッシュに署名します。

B.9.2 Functions Documentation

B.9.2.1 function wc_SignatureGetSize

```
int wc_SignatureGetSize(
    enum wc_SignatureType sig_type,
    const void * key,
    word32 key_len
)
```

この関数は、結果の署名の最大サイズを返します。

Parameters:

- **sig_type** WC_SIGNATURE_TYPE_ECC や WC_SIGNATURE_TYPE_RSA などの署名タイプの列挙値。
- **key** ecc_key や RsaKey などのキー構造体へのポインタ。
- **key_len** キー構造体のサイズ。

See:

- [wc_HashGetDigestSize](#)
- [wc_SignatureGenerate](#)
- [wc_SignatureVerify](#)

Return: sig_type がサポートされていない場合、SIG_TYPE_E を返します。sig_type が無効な場合、BAD_FUNC_ARG を返します。正の戻り値は、署名の最大サイズを示します。

Example

```
// 署名の長さを取得
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) {
    // 成功
}
```

B.9.2.2 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,
    const void * key,
    word32 key_len
)
```

この関数は、データをハッシュ化し、結果のハッシュとキーを使用して署名を検証することで、署名を検証します。

Parameters:

- **hash_type** "WC_HASH_TYPE_SHA256" などの "enum wc_HashType" からのハッシュタイプ。
- **sig_type** WC_SIGNATURE_TYPE_ECC や WC_SIGNATURE_TYPE_RSA などの署名タイプの列挙値。
- **data** ハッシュ化するデータを含むバッファへのポインタ。
- **data_len** データバッファの長さ。

- **sig** 署名を出力するバッファへのポインタ。
- **sig_len** 署名出力バッファの長さ。
- **key** ecc_key や RsaKey などのキー構造体へのポインタ。
- **key_len** キー構造体のサイズ。

See:

- [wc_SignatureGetSize](#)
- [wc_SignatureGenerate](#)

Return:

- 0 成功
- SIG_TYPE_E -231、署名タイプが有効化されていない/利用できない
- BAD_FUNC_ARG -173、不正な関数引数が提供された
- BUFFER_E -132、出力バッファが小さすぎるか、入力が大きすぎる。

Example

```
int ret;
ecc_key eccKey;

// 公開鍵をインポート
wc_ecc_init(&eccKey);
ret = wc_ecc_import_x963(eccPubKeyBuf, eccPubKeyLen, &eccKey);
// 公開鍵を使用して署名検証を実行
ret = wc_SignatureVerify(
    WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
    fileBuf, fileLen,
    sigBuf, sigLen,
    &eccKey, sizeof(eccKey));
printf(" 署名検証: %s\n", (ret == 0) ? "合格" : "不合格", ret);
wc_ecc_free(&eccKey);
```

B.9.2.3 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,
    const void * key,
    word32 key_len,
    WC_RNG * rng
)
```

この関数は、キーを使用してデータから署名を生成します。最初にデータのハッシュを作成し、次にキーを使用してハッシュに署名します。

Parameters:

- **hash_type** “WC_HASH_TYPE_SHA256” などの “enum wc_HashType” からのハッシュタイプ。
- **sig_type** WC_SIGNATURE_TYPE_ECC や WC_SIGNATURE_TYPE_RSA などの署名タイプの列挙値。
- **data** ハッシュ化するデータを含むバッファへのポインタ。
- **data len** データバッファの長さ。
- **sig** 署名を出力するバッファへのポインタ。

- **sig_len** 署名出力バッファの長さ。
- **key** ecc_key や RsaKey などのキー構造体へのポインタ。
- **key_len** キー構造体のサイズ。
- **rng** 初期化された RNG 構造体へのポインタ。

See:

- [wc_SignatureGetSize](#)
- [wc_SignatureVerify](#)

Return:

- 0 成功
- SIG_TYPE_E -231、署名タイプが有効化されていない/利用できない
- BAD_FUNC_ARG -173、不正な関数引数が提供された
- BUFFER_E -132、出力バッファが小さすぎるか、入力が大きすぎる。

Example

```
int ret;
WC_RNG rng;
ecc_key eccKey;

wc_InitRng(&rng);
wc_ecc_init(&eccKey);

// キーを生成
ret = wc_ecc_make_key(&rng, 32, &eccKey);

// 署名の長さを取得してバッファを割り当て
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
sigBuf = malloc(sigLen);

// 公開鍵を使用して署名検証を実行
ret = wc_SignatureGenerate(
    WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
    fileBuf, fileLen,
    sigBuf, &sigLen,
    &eccKey, sizeof(eccKey),
    &rng);
printf(" 署名生成: %s\n", (ret == 0) ? "合格" : "不合格", ret);

free(sigBuf);
wc_ecc_free(&eccKey);
wc_FreeRng(&rng);
```

B.10 wolfCrypt の初期化とクリーンアップ

B.10.1 Functions

	Name
int	wc_HashGetOID (enum wc_HashType hash_type) この関数は、提供された wc_HashType の OID を返します。

	Name
int	wc_HashGetDigestSize (enum wc_HashType hash_type) この関数は、hash_type のダイジェスト (出力) のサイズを返します。返されるサイズは、wc_Hash に提供される出力バッファが十分な大きさであることを確認するために使用されます。
int	wc_Hash (enum wc_HashType hash_type, const byte * data, word32 data_len, byte * hash, word32 hash_len) この関数は、提供されたデータバッファに対してハッシュを実行し、提供されたハッシュバッファに結果を返します。
int	wolfCrypt_Init (void)wolfCrypt で使用されるリソースを初期化するために使用されます。
int	wolfCrypt_Cleanup (void)wolfCrypt で使用されるリソースをクリーンアップするために使用されます。

B.10.2 Functions Documentation

B.10.2.1 function wc_HashGetOID

```
int wc_HashGetOID(
    enum wc_HashType hash_type
)
```

この関数は、提供された wc_HashType の OID を返します。

Parameters:

- **hash_type** “WC_HASH_TYPE_SHA256” などの “enum wc_HashType” からのハッシュタイプ。

See:

- [wc_HashGetDigestSize](#)
- [wc_Hash](#)

Return:

- OID 0 より大きい値を返します
- HASH_TYPE_E ハッシュタイプがサポートされていません。
- BAD_FUNC_ARG 提供された引数の 1 つが正しくありません。

Example

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int oid = wc_HashGetOID(hash_type);
if (oid > 0) {
    // 成功
}
```

B.10.2.2 function wc_HashGetDigestSize

```
int wc_HashGetDigestSize(
    enum wc_HashType hash_type
)
```

この関数は、hash_type のダイジェスト (出力) のサイズを返します。返されるサイズは、wc_Hash に提供される出力バッファが十分な大きさであることを確認するために使用されます。

Parameters:

- **hash_type** “WC_HASH_TYPE_SHA256” などの “enum wc_HashType” からのハッシュタイプ。

See: `wc_Hash`

Return:

- Success 正の戻り値は、ハッシュのダイジェストサイズを示します。
- Error hash_type がサポートされていない場合、HASH_TYPE_E を返します。
- Failure 無効な hash_type が使用された場合、BAD_FUNC_ARG を返します。

Example

```
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len <= 0) {
    WOLFSSL_MSG("Invalid hash type/len");
    return BAD_FUNC_ARG;
}
```

B.10.2.3 function wc_Hash

```
int wc_Hash(
    enum wc_HashType hash_type,
    const byte * data,
    word32 data_len,
    byte * hash,
    word32 hash_len
)
```

この関数は、提供されたデータバッファに対してハッシュを実行し、提供されたハッシュバッファに結果を返します。

Parameters:

- **hash_type** “WC_HASH_TYPE_SHA256” などの “enum wc_HashType” からのハッシュタイプ。
- **data** ハッシュ化するデータを含むバッファへのポインタ。
- **data_len** データバッファの長さ。
- **hash** 最終ハッシュを出力するために使用されるバッファへのポインタ。
- **hash_len** ハッシュバッファの長さ。

See: `wc_HashGetDigestSize`

Return: 0 成功、それ以外はエラー (BAD_FUNC_ARG や BUFFER_E など)。

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) {
        // 成功
    }
}
```

B.10.2.4 function wolfCrypt_Init

```
int wolfCrypt_Init(
    void
)
```

wolfCrypt で使用されるリソースを初期化するために使用されます。

Parameters:

- **none** パラメータなし。

See: [wolfCrypt_Cleanup](#)

Return:

- 0 成功時。
- <0 リソースの初期化に失敗した場合。

Example

```
...
if (wolfCrypt_Init() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Init call");
}
```

B.10.2.5 function wolfCrypt_Cleanup

```
int wolfCrypt_Cleanup(
    void
)
```

wolfCrypt で使用されるリソースをクリーンアップするために使用されます。

Parameters:

- **none** パラメータなし。

See: [wolfCrypt_Init](#)

Return:

- 0 成功時。
- <0 リソースのクリーンアップに失敗した場合。

Example

```
...
if (wolfCrypt_Cleanup() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Cleanup call");
}
```

B.11 アルゴリズム - 3DES

B.11.1 Functions

	Name
int	wc_Des_SetKey (Des * des, const byte * key, const byte * iv, int dir) この関数は、引数として与えられた Des 構造体の鍵と初期化ベクトル (iv) を設定します。また、暗号化と復号に必要なバッファがまだ初期化されていない場合、それらを初期化し、スペースを割り当てます。注意: iv が提供されない場合 (つまり iv == NULL)、初期化ベクトルはデフォルトで 0 の iv になります。

	Name
void	wc_Des_SetIV (Des * des, const byte * iv) この関数は、引数として与えられた Des 構造体の初期化ベクトル (iv) を設定します。NULL の iv が渡された場合、初期化ベクトルを 0 に設定します。
int	wc_Des_CbcEncrypt (Des * des, byte * out, const byte * in, word32 sz) この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。
int	wc_Des_CbcDecrypt (Des * des, byte * out, const byte * in, word32 sz) この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。
int	wc_Des_EcbEncrypt (Des * des, byte * out, const byte * in, word32 sz) この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。電子コードブック (ECB) モードの Des 暗号化を使用します。
int	wc_Des3_EcbEncrypt (Des3 * des, byte * out, const byte * in, word32 sz) この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。電子コードブック (ECB) モードの Des3 暗号化を使用します。警告: ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。
int	wc_Des3_SetKey (Des3 * des, const byte * key, const byte * iv, int dir) この関数は、引数として与えられた Des3 構造体の鍵と初期化ベクトル (iv) を設定します。また、暗号化と復号に必要なバッファがまだ初期化されていない場合、それらを初期化し、スペースを割り当てます。注意: iv が提供されない場合 (つまり iv == NULL)、初期化ベクトルはデフォルトで 0 の iv になります。
int	wc_Des3_SetIV (Des3 * des, const byte * iv) この関数は、引数として与えられた Des3 構造体の初期化ベクトル (iv) を設定します。NULL の iv が渡された場合、初期化ベクトルを 0 に設定します。
int	wc_Des3_CbcEncrypt (Des3 * des, byte * out, const byte * in, word32 sz) この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。
int	wc_Des3_CbcDecrypt (Des3 * des, byte * out, const byte * in, word32 sz) この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。

	Name
int	wc_Des_CbcDecryptWithKey (byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。この関数は wc_Des_CbcDecrypt の代替で、ユーザーが Des 構造体を直接インスタンス化せずにメッセージを復号できるようにします。
int	wc_Des_CbcEncryptWithKey (byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は、入力平文 in を暗号化し、結果の暗号文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。この関数は wc_Des_CbcEncrypt の代替で、ユーザーが Des 構造体を直接インスタンス化せずにメッセージを暗号化できるようにします。
int	wc_Des3_CbcEncryptWithKey (byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は、入力平文 in を暗号化し、結果の暗号文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル DES(3DES) 暗号化を使用します。この関数は wc_Des3_CbcEncrypt の代替で、ユーザーが Des3 構造体を直接インスタンス化せずにメッセージを暗号化できるようにします。
int	wc_Des3_CbcDecryptWithKey (byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。この関数は wc_Des3_CbcDecrypt の代替で、ユーザーが Des3 構造体を直接インスタンス化せずにメッセージを復号できるようにします。

B.11.2 Functions Documentation

B.11.2.1 function wc_Des_SetKey

```
int wc_Des_SetKey(
    Des * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

この関数は、引数として与えられた Des 構造体の鍵と初期化ベクトル (iv) を設定します。また、暗号化と復号に必要なバッファがまだ初期化されていない場合、それらを初期化し、スペースを割り当てます。注意: iv が提供されない場合 (つまり iv == NULL)、初期化ベクトルはデフォルトで 0 の iv になります。

Parameters:

- **des** 初期化する Des 構造体へのポインタ
- **key** Des 構造体を初期化するための 8 バイトの鍵を含むバッファへのポインタ

- **iv** Des 構造体を初期化するための 8 バイトの iv を含むバッファへのポインタ。これが提供されない場合、iv はデフォルトで 0 になります
- **dir** 暗号化の方向。有効なオプションは: DES_ENCRYPTION と DES_DECRYPTION です

See:

- [wc_Des_SetIV](#)
- [wc_Des3_SetKey](#)

Return: 0 Des 構造体の鍵と初期化ベクトルの設定に成功した場合

3

Example

```
Des enc; // 暗号化に使用される Des 構造体
int ret;
byte key[] = { // 8 バイトの鍵で初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };

ret = wc_Des_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // des 構造体の初期化エラー
}
```

B.11.2.2 function wc_Des_SetIV

```
void wc_Des_SetIV(
    Des * des,
    const byte * iv
)
```

この関数は、引数として与えられた Des 構造体の初期化ベクトル (iv) を設定します。NULL の iv が渡された場合、初期化ベクトルを 0 に設定します。

Parameters:

- **des** iv を設定する Des 構造体へのポインタ
- **iv** Des 構造体を初期化するための 8 バイトの iv を含むバッファへのポインタ。これが提供されない場合、iv はデフォルトで 0 になります

See: [wc_Des_SetKey](#)

Return: none 戻り値なし。

3

Example

```
Des enc; // 暗号化に使用される Des 構造体
// wc_Des_SetKey で enc を初期化
byte iv[] = { // 8 バイトの iv で初期化 };
wc_Des_SetIV(&enc, iv);
}
```

B.11.2.3 function wc_Des_CbcEncrypt

```
int wc_Des_CbcEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

)

この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。

Parameters:

- **des** 暗号化に使用する Des 構造体へのポインタ
- **out** 暗号化された暗号文を格納するバッファへのポインタ
- **in** 暗号化するメッセージを含む入力バッファへのポインタ
- **sz** 暗号化するメッセージの長さ

See:

- [wc_Des_SetKey](#)
- [wc_Des_CbcDecrypt](#)

Return: 0 指定された入力メッセージの暗号化に成功した場合に返されます

3

Example

```
Des enc; // 暗号化に使用される Des 構造体
// wc_Des_SetKey で enc を初期化、モード DES_ENCRYPTION を使用

byte plain[] = { // メッセージで初期化 };
byte cipher[sizeof(plain)];

if ( wc_Des_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0 ) {
    // メッセージの暗号化エラー
}
```

B.11.2.4 function wc_Des_CbcDecrypt

```
int wc_Des_CbcDecrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。

Parameters:

- **des** 復号に使用する Des 構造体へのポインタ
- **out** 復号された平文を格納するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号する暗号文の長さ

See:

- [wc_Des_SetKey](#)
- [wc_Des_CbcEncrypt](#)

Return: 0 指定された暗号文の復号に成功した場合に返されます

3

Example

```

Des dec; // 復号に使用される Des 構造体
// wc_Des_SetKey で dec を初期化、モード DES_DECRYPTION を使用

byte cipher[] = { // 暗号文で初期化 };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
    // メッセージの復号エラー
}

```

B.11.2.5 function wc_Des_EcbEncrypt

```

int wc_Des_EcbEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)

```

この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。電子コードブック (ECB) モードの Des 暗号化を使用します。

Parameters:

- **des** 暗号化に使用する Des 構造体へのポインタ
- **out** 暗号化されたメッセージを格納するバッファへのポインタ
- **in** 暗号化する平文を含む入力バッファへのポインタ
- **sz** 暗号化する平文の長さ

See: wc_Des_SetKe

Return: 0: 指定された平文の暗号化に成功した場合に返されます。

3

Example

```

Des enc; // 暗号化に使用される Des 構造体
// wc_Des_SetKey で enc を初期化、モード DES_ENCRYPTION を使用

byte plain[] = { // 暗号化するメッセージで初期化 };
byte cipher[sizeof(plain)];

if ( wc_Des_EcbEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
    // メッセージの暗号化エラー
}

```

B.11.2.6 function wc_Des3_EcbEncrypt

```

int wc_Des3_EcbEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)

```

この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。電子コードブック (ECB) モードの Des3 暗号化を使用します。警告: ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

Parameters:

- **des3** 暗号化に使用する Des3 構造体へのポインタ
- **out** 暗号化されたメッセージを格納するバッファへのポインタ
- **in** 暗号化する平文を含む入力バッファへのポインタ
- **sz** 暗号化する平文の長さ

See: [wc_Des3_SetKey](#)

Return: 0 指定された平文の暗号化に成功した場合に返されます

3

Example

```
Des3 enc; // 暗号化に使用される Des3 構造体
// wc_Des3_SetKey で enc を初期化、モード DES_ENCRYPTION を使用

byte plain[] = { // 暗号化するメッセージで初期化 };
byte cipher[sizeof(plain)];

if ( wc_Des3_EcbEncrypt(&enc, cipher, plain, sizeof(plain)) != 0 ) {
    // メッセージの暗号化エラー
}
```

B.11.2.7 function wc_Des3_SetKey

```
int wc_Des3_SetKey(
    Des3 * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

この関数は、引数として与えられた Des3 構造体の鍵と初期化ベクトル (iv) を設定します。また、暗号化と復号に必要なバッファがまだ初期化されていない場合、それらを初期化し、スペースを割り当てます。注意: iv が提供されない場合 (つまり iv == NULL)、初期化ベクトルはデフォルトで 0 の iv になります。

Parameters:

- **des3** 初期化する Des3 構造体へのポインタ
- **key** Des3 構造体を初期化するための 24 バイトの鍵を含むバッファへのポインタ
- **iv** Des3 構造体を初期化するための 8 バイトの iv を含むバッファへのポインタ。これが提供されない場合、iv はデフォルトで 0 になります
- **dir** 暗号化の方向。有効なオプションは: DES_ENCRYPTION と DES_DECRYPTION です

See:

- [wc_Des3_SetIV](#)
- [wc_Des3_CbcEncrypt](#)
- [wc_Des3_CbcDecrypt](#)

Return: 0 Des 構造体の鍵と初期化ベクトルの設定に成功した場合

3

Example

```
Des3 enc; // 暗号化に使用される Des3 構造体
int ret;
byte key[] = { // 24 バイトの鍵で初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };
```

```
ret = wc_Des3_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // des 構造体の初期化エラー
}
```

B.11.2.8 function wc_Des3_SetIV

```
int wc_Des3_SetIV(
    Des3 * des,
    const byte * iv
)
```

この関数は、引数として与えられた Des3 構造体の初期化ベクトル (iv) を設定します。NULL の iv が渡された場合、初期化ベクトルを 0 に設定します。

Parameters:

- **des** iv を設定する Des3 構造体へのポインタ
- **iv** Des3 構造体を初期化するための 8 バイトの iv を含むバッファへのポインタ。これが提供されない場合、iv はデフォルトで 0 になります

See: [wc_Des3_SetKey](#)

Return: none 戻り値なし。

3

Example

```
Des3 enc; // 暗号化に使用される Des3 構造体
// wc_Des3_SetKey で enc を初期化

byte iv[] = { // 8 バイトの iv で初期化 };

wc_Des3_SetIV(&enc, iv);
}
```

B.11.2.9 function wc_Des3_CbcEncrypt

```
int wc_Des3_CbcEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。

Parameters:

- **des** 暗号化に使用する Des3 構造体へのポインタ
- **out** 暗号化された暗号文を格納するバッファへのポインタ
- **in** 暗号化するメッセージを含む入力バッファへのポインタ
- **sz** 暗号化するメッセージの長さ

See:

- [wc_Des3_SetKey](#)
- [wc_Des3_CbcDecrypt](#)

Return: 0 指定された入力メッセージの暗号化に成功した場合に返されます

3

Example

```
Des3 enc; // 暗号化に使用される Des3 構造体
// wc_Des3_SetKey で enc を初期化、モード DES_ENCRYPTION を使用

byte plain[] = { // メッセージで初期化 };
byte cipher[sizeof(plain)];

if ( wc_Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
    // メッセージの暗号化エラー
}
```

B.11.2.10 function wc_Des3_CbcDecrypt

```
int wc_Des3_CbcDecrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。

Parameters:

- **des** 復号に使用する Des3 構造体へのポインタ
- **out** 復号された平文を格納するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号する暗号文の長さ

See:

- [wc_Des3_SetKey](#)
- [wc_Des3_CbcEncrypt](#)

Return: 0 指定された暗号文の復号に成功した場合に返されます

3

Example

```
Des3 dec; // 復号に使用される Des 構造体
// wc_Des3_SetKey で dec を初期化、モード DES_DECRYPTION を使用

byte cipher[] = { // 暗号文で初期化 };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
    // メッセージの復号エラー
}
```

B.11.2.11 function wc_Des_CbcDecryptWithKey

```
int wc_Des_CbcDecryptWithKey(
    byte * out,
    const byte * in,
```

```

    word32 sz,
    const byte * key,
    const byte * iv
)

```

この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。この関数は wc_Des_CbcDecrypt の代替で、ユーザーが Des 構造体を直接インスタンス化せずにメッセージを復号できるようにします。

Parameters:

- **out** 復号された平文を格納するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号する暗号文の長さ
- **key** 復号に使用する 8 バイトのキーを含むバッファへのポインタ
- **iv** 復号に使用する 8 バイトの iv を含むバッファへのポインタ。iv が提供されない場合、iv はデフォルトで 0 になります

See: [wc_Des_CbcDecrypt](#)

Return:

- 0 指定された暗号文の復号に成功した場合に返されます
- MEMORY_E Des 構造体用のスペース割り当て中にエラーが発生した場合に返されます

3

Example

```

int ret;
byte key[] = { // 8 バイトのキーで初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };

byte cipher[] = { // 暗号文で初期化 };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecryptWithKey(decoded, cipher, sizeof(cipher), key,
iv) != 0 ) {
    // メッセージの復号エラー
}

```

B.11.2.12 function wc_Des_CbcEncryptWithKey

```

int wc_Des_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)

```

この関数は、入力平文 in を暗号化し、結果の暗号文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。この関数は wc_Des_CbcEncrypt の代替で、ユーザーが Des 構造体を直接インスタンス化せずにメッセージを暗号化できるようにします。

Parameters:

- **out** 最終的に暗号化されたデータ
- **in** 暗号化されるデータ。Des ブロックサイズにパディングされている必要があります。
- **sz** 入力バッファのサイズ。

- **key** 暗号化に使用するキーへのポインタ。
- **iv** 初期化ベクトル

See:

- [wc_Des_CbcDecryptWithKey](#)
- [wc_Des_CbcEncrypt](#)

Return:

- 0 データの暗号化に成功した後に返されます。
- MEMORY_E Des 構造体用のメモリ割り当て中にエラーが発生した場合に返されます。
- <0 暗号化中の任意のエラーで返されます。

3

Example

```
byte key[] = { // 8 バイトのキーで初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };
byte in[] = { // 平文で初期化 };
byte out[sizeof(in)];
if ( wc_Des_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
    // メッセージの暗号化エラー
}
```

B.11.2.13 function wc_Des3_CbcEncryptWithKey

```
int wc_Des3_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

この関数は、入力平文 in を暗号化し、結果の暗号文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル DES(3DES) 暗号化を使用します。この関数は wc_Des3_CbcEncrypt の代替で、ユーザーが Des3 構造体を直接インスタンス化せずにメッセージを暗号化できるようにします。

Parameters:

- **out** 最終的に暗号化されたデータ
- **in** 暗号化されるデータ。Des ブロックサイズにパディングされている必要があります。
- **sz** 入力バッファのサイズ。
- **key** 暗号化に使用するキーへのポインタ。
- **iv** 初期化ベクトル

See:

- [wc_Des3_CbcDecryptWithKey](#)
- [wc_Des_CbcEncryptWithKey](#)
- [wc_Des_CbcDecryptWithKey](#)

Return:

- 0 データの暗号化に成功した後に返されます。
- MEMORY_E Des 構造体用のメモリ割り当て中にエラーが発生した場合に返されます。
- <0 暗号化中の任意のエラーで返されます。

3

Example

```

byte key[] = { // 8 バイトのキーで初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };

byte in[] = { // 平文で初期化 };
byte out[sizeof(in)];

if ( wc_Des3_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
    // メッセージの暗号化エラー
}

```

B.11.2.14 function wc_Des3_CbcDecryptWithKey

```

int wc_Des3_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)

```

この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。この関数は wc_Des3_CbcDecrypt の代替で、ユーザーが Des3 構造体を直接インスタンス化せずにメッセージを復号できるようにします。

Parameters:

- **out** 復号された平文を格納するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号する暗号文の長さ
- **key** 復号に使用する 24 バイトのキーを含むバッファへのポインタ
- **iv** 復号に使用する 8 バイトの iv を含むバッファへのポインタ。iv が提供されない場合、iv はデフォルトで 0 になります

See: [wc_Des3_CbcDecrypt](#)

Return:

- 0 指定された暗号文の復号に成功した場合に返されます
- MEMORY_E Des 構造体用のスペース割り当て中にエラーが発生した場合に返されます

3

Example

```

int ret;
byte key[] = { // 24 バイトのキーで初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };

byte cipher[] = { // 暗号文で初期化 };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecryptWithKey(decoded, cipher, sizeof(cipher),
key, iv) != 0) {
    // メッセージの復号エラー
}

```

B.12 アルゴリズム - AES

B.12.1 Functions

	Name
int	wc_AesSetKey (Aes * aes, const byte * key, word32 len, const byte * iv, int dir) この関数は、キーを設定し、初期化ベクトルを設定することで AES 構造体を初期化します。
int	wc_AesSetIV (Aes * aes, const byte * iv) この関数は、特定の AES オブジェクトの初期化ベクトルを設定します。この関数を呼び出す前に AES オブジェクトを初期化する必要があります。
int	wc_AesCbcEncrypt (Aes * aes, byte * out, const byte * in, word32 sz) 入力バッファ in から平文メッセージを暗号化し、AES を使用した暗号ブロック連鎖により結果の暗号文を出力バッファ out に格納します。この関数は、メッセージを暗号化する前に AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、入力メッセージが AES ブロック長に整列していることを前提としており、入力長がブロック長の倍数であることを期待します。ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合、これはオプションでチェックおよび強制されます。ブロック倍数の入力を保証するために、事前に PKCS#7 スタイルのパディングを追加する必要があります。これは、自動的にパディングを追加する OpenSSL の AES-CBC メソッドとは異なります。wolfSSL と対応する OpenSSL 関数を相互運用させるには、OpenSSL コマンドライン関数で -nopad オプションを指定して、wolfSSL の AesCbcEncrypt メソッドのように動作し、暗号化中に余分なパディングを追加しないようにする必要があります。

	Name
int	wc_AesCbcDecrypt(Aes * aes, byte * out, const byte * in, word32 sz) 入力バッファ in から暗号を復号し、AES を使用した暗号ブロック連鎖により結果の平文を出力バッファ out に格納します。この関数は、メッセージを復号する前に AesSetKey を呼び出して AES 構造体が初期化されている必要があります。この関数は、元のメッセージが AES ブロック長に整列していたことを前提としており、入力長がブロック長の倍数であることを期待します。ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合、これはオプションでチェックおよび強制されます。これは、PKCS#7 パディングを自動的に追加する OpenSSL の AES-CBC メソッドとは異なり、ブロック倍数の入力を必要としません。wolfSSL 関数と同等の OpenSSL 関数を相互運用させるには、OpenSSL コマンドライン関数で -nopad オプションを指定して、wolfSSL の AesCbcEncrypt メソッドのように動作し、復号中にエラーを発生させないようにする必要があります。
int	wc_AesCtrEncrypt(Aes * aes, byte * out, const byte * in, word32 sz) AES を使用した CTR モードで、入力バッファ in からメッセージを暗号化/復号し、結果の暗号文を出力バッファ out に格納します。この関数は、コンパイル時に WOLFSSL_AES_COUNTER が有効になっている場合のみ有効です。この関数を呼び出す前に、AesSetKey を介して AES 構造体を初期化する必要があります。この関数は復号と暗号化の両方に使用されることに注意してください。注：暗号化と復号に同じ API を使用することについて。ユーザは暗号化/復号用の Aes 構造体を区別する必要があります。
int	wc_AesEncryptDirect(Aes * aes, byte * out, const byte * in) この関数は、入力ブロック in の 1 ブロック暗号化を出力ブロック out に行います。提供された AES 構造体のキーを使用し、この関数を呼び出す前に wc_AesSetKey で初期化する必要があります。wc_AesSetKey は iv を NULL に設定して呼び出す必要があります。これは、構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

	Name
int	wc_AesDecryptDirect (Aes * aes, byte * out, const byte * in) この関数は、入力ブロック in の 1 ブロック復号を出力ブロック out に行います。提供された AES 構造体のキーを使用し、この関数を呼び出す前に wc_AesSetKey で初期化する必要があります。wc_AesSetKey は iv を NULL に設定して呼び出す必要があります。これは、構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。
int	wc_AesSetKeyDirect (Aes * aes, const byte * key, word32 len, const byte * iv, int dir) この関数は、AES を使用した CTR モードの AES キーを設定するために使用されます。指定されたキー、iv (初期化ベクトル)、および暗号化 dir (方向) で AES オブジェクトを初期化します。構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。現在、wc_AesSetKeyDirect は内部的に wc_AesSetKey を使用しています。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。
int	wc_AesGcmSetKey (Aes * aes, const byte * key, word32 len) この関数は、AES GCM (Galois/Counter Mode) のキーを設定するために使用されます。指定されたキーで AES オブジェクトを初期化します。コンパイル時に構成オプション HAVE_AESGCM が有効になっている場合のみ有効です。
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) この関数は、バッファ in に保持されている入力メッセージを暗号化し、結果の暗号文を出力バッファ out に格納します。暗号化の呼び出しごとに新しい iv (初期化ベクトル) が必要です。また、入力認証ベクトル authIn を認証タグ authTag にエンコードします。

	Name
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) この関数は、バッファ in に保持されている入力暗号文を復号し、結果のメッセージテキストを出力バッファ out に格納します。また、入力認証ベクトル authIn を、提供された認証タグ authTag と照合してチェックします。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。
int	wc_GmacSetKey (Gmac * gmac, const byte * key, word32 len) この関数は、Galois メッセージ認証に使用される GMAC オブジェクトのキーを初期化および設定します。
int	wc_GmacUpdate (Gmac * gmac, const byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, byte * authTag, word32 authTagSz) この関数は、authIn 入力の Gmac ハッシュを生成し、結果を authTag バッファに格納します。wc_GmacUpdate を実行した後、生成された authTag を既知の認証タグと比較して、メッセージの真正性を検証する必要があります。
int	wc_AesCcmSetKey (Aes * aes, const byte * key, word32 keySz) この関数は、CCM (Counter with CBC-MAC) を使用して AES オブジェクトのキーを設定します。AES 構造体へのポインタを受け取り、提供されたキーで初期化します。
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) この関数は、CCM (Counter with CBC-MAC) を使用して、入力メッセージ in を出力バッファ out に暗号化します。その後、authIn 入力から認証タグ authTag を計算して格納します。
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) この関数は、CCM (Counter with CBC-MAC) を使用して、入力暗号文 in を出力バッファ out に復号します。その後、authIn 入力から認証タグ authTag を計算します。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

	Name
int	wc_AesXtsInit (XtsAes * aes, void * heap, int devId) これは AES-XTS コンテキストを初期化するためのものです。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。
int	wc_AesXtsSetKeyNoInit (XtsAes * aes, const byte * key, word32 len, int dir) これは、最初に wc_AesXtsInit() を呼び出した後、キーを正しい暗号化または復号タイプに設定するのに役立ちます。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。
int	wc_AesXtsSetKey (XtsAes * aes, const byte * key, word32 len, int dir, void * heap, int devId) これは、キーを正しい暗号化または復号タイプに設定するのに役立ちます。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。
int	wc_AesXtsEncryptSector (XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)wc_AesXtsEncrypt と同じプロセスですが、バイト配列の代わりに word64 型を tweak 値として使用します。これは word64 をバイト配列に変換し、wc_AesXtsEncrypt を呼び出すだけです。
int	wc_AesXtsDecryptSector (XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)wc_AesXtsDecrypt と同じプロセスですが、バイト配列の代わりに word64 型を tweak 値として使用します。これは word64 をバイト配列に変換するだけです。
int	wc_AesXtsEncrypt (XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)XTS モードの AES。(XTS) Tweak と Cipher Text Stealing を使用した XEX 暗号化。
int	wc_AesXtsDecrypt (XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz) 暗号化と同じプロセスですが、Aes キーは AES_DECRYPTION タイプです。
int	wc_AesXtsFree (XtsAes * aes) これは、XtsAes 構造体によって使用されるリソースを解放するためのものです。
int	wc_AesInit (Aes * aes, void * heap, int devId)Aes 構造体を初期化します。使用するヒープヒントと非同期ハードウェアで使用する ID を設定します。使用が完了したら、ユーザが Aes 構造体に対して wc_AesFree を呼び出す必要があります。

	Name
void	wc_AesFree (Aes * aes) 該当する場合、Aes 構造体に関連付けられたリソースを解放します。内部的には時々 no-op になることもありますが、新しい環境で使用するためにアプリケーションコードが移植される場合（呼び出しが適用される場合）など、一般的なベストプラクティスとしてすべてのケースで呼び出すことをお勧めします。
int	wc_AesCfbEncrypt (Aes * aes, byte * out, const byte * in, word32 sz) CFB モードの AES。
int	wc_AesCfbDecrypt (Aes * aes, byte * out, const byte * in, word32 sz) CFB モードの AES。
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) この関数は、RFC 5297 で説明されている SIV（合成初期化ベクトル）暗号化を実行します。
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) この関数は、RFC 5297 で説明されている SIV（合成初期化ベクトル）復号を実行します。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。
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) この関数は、「EAX: A Conventional Authenticated-Encryption Mode」(https://eprint.iacr.org/2003/069) で説明されている AES EAX 暗号化と認証を実行します。これは、すべての暗号化と認証操作を 1 つの関数呼び出しで実行する「ワンショット」API です。

	Name
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) この関数は、「EAX: A Conventional Authenticated-Encryption Mode」(https://eprint.iacr.org/2003/069) で説明されている AES EAX 復号と認証を実行します。これは、すべての復号と認証操作を 1 つの関数呼び出しで実行する「ワンショット」API です。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。
WOLFSSL_API int	wc_AesEaxInit (AesEax * eax, const byte * key, word32 keySz, const byte * nonce, word32 nonceSz, const byte * authIn, word32 authInSz) この関数は、認証付き暗号化または復号で使用する AesEax オブジェクトを初期化します。この関数は、AES EAX 増分 API のいずれかと使用する前に、AesEax オブジェクトに対して呼び出す必要があります。ワンショット EAX API 関数を使用する場合は呼び出す必要はありません。この関数で初期化されたすべての AesEax インスタンスは、インスタンスの使用が完了したら wc_AesEaxFree() の呼び出しで解放する必要があります。
WOLFSSL_API int	**wc_AesEaxEncryptUpdate の呼び出しで事前に初期化されている必要があります。
WOLFSSL_API int	**wc_AesEaxDecryptUpdate の呼び出しで事前に初期化されている必要があります。
WOLFSSL_API int	**wc_AesEaxAuthDataUpdate の呼び出しで事前に初期化されている必要があります。
WOLFSSL_API int	**wc_AesEaxEncryptFinal の呼び出しで事前に初期化されている必要があります。AesEax コンテキスト構造体の使用が完了したら、wc_AesEaxFree を使用して必ず解放してください。
WOLFSSL_API int	**wc_AesEaxDecryptFinal の呼び出しで事前に初期化されている必要があります。AesEax コンテキスト構造体の使用が完了したら、wc_AesEaxFree を使用して必ず解放してください。
WOLFSSL_API int	wc_AesEaxFree (AesEax * eax) この関数は、AesEax ラッパー構造体内の Aes インスタンスによって使用されるリソース、特にキーを解放します。wc_AesEaxInit で初期化された後、すべての必要な EAX 操作が完了した時点で AesEax 構造体に対して呼び出す必要があります。

	Name
int	wc_AesCtsEncrypt (const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv) この関数は、CTS モードを使用して AES 暗号化を実行します。これは、すべての操作を 1 回の呼び出しで処理するワンショット API です。
int	wc_AesCtsDecrypt (const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv) この関数は、CTS モードを使用して AES 復号を実行します。これは、すべての操作を 1 回の呼び出しで処理するワンショット API です。
int	wc_AesCtsEncryptUpdate (Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz) この関数は、AES CTS 暗号化の更新ステップを実行します。平文のチャンクを処理し、中間データを保存します。
int	wc_AesCtsEncryptFinal (Aes * aes, byte * out, word32 * outSz) この関数は、AES CTS 暗号化操作を完了します。残りの平文を処理し、暗号化を完了します。
int	wc_AesCtsDecryptUpdate (Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz) この関数は、AES CTS 復号の更新ステップを実行します。暗号文のチャンクを処理し、中間データを保存します。
int	wc_AesCtsDecryptFinal (Aes * aes, byte * out, word32 * outSz) この関数は、AES CTS 復号操作を完了します。残りの暗号文を処理し、復号を完了します。
int	wc_AesCbcDecryptWithKey (byte * out, const byte * in, word32 inSz, const byte * key, word32 keySz, const byte * iv) 入力バッファ in から暗号を復号し、結果の平文を AES を使用した暗号ブロック連鎖を使用して出力バッファ out に格納します。この関数は、AES 構造体を初期化する必要はありません。代わりに、キーと iv(初期化ベクトル)を受け取り、これらを使用して AES オブジェクトを初期化し、暗号文を復号します。

B.12.2 Functions Documentation

B.12.2.1 function wc_AesSetKey

```
int wc_AesSetKey(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

この関数は、キーを設定し、初期化ベクトルを設定することで AES 構造体を初期化します。

Parameters:

- **aes** 変更する AES 構造体へのポインタ
- **key** 暗号化と復号のための 16、24、または 32 バイトの秘密鍵
- **len** 渡されたキーの長さ
- **iv** キーの初期化に使用される初期化ベクトルへのポインタ
- **dir** 暗号の方向。暗号化する場合は AES_ENCRYPTION を設定し、復号する場合は AES_DECRYPTION を設定します。一部のモード（CFB および CTR）の方向は常に AES_ENCRYPTION です。

See:

- [wc_AesSetKeyDirect](#)
- [wc_AesSetIV](#)

Return:

- 0 キーと初期化ベクトルの設定に成功した場合に返されます。
- BAD_FUNC_ARG キーの長さが無効な場合に返されます。

Example

```
Aes enc;
int ret = 0;
byte key[] = { 16、24、または32バイトのキー };
byte iv[] = { 16バイトの iv };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // aes key の初期化に失敗
}
if (ret = wc_AesSetKey(&enc, key, AES_BLOCK_SIZE, iv,
AES_ENCRYPTION) != 0) {
    // aes key の設定に失敗
}
```

B.12.2.2 function wc_AesSetIV

```
int wc_AesSetIV(
    Aes * aes,
    const byte * iv
)
```

この関数は、特定の AES オブジェクトの初期化ベクトルを設定します。この関数を呼び出す前に AES オブジェクトを初期化する必要があります。

Parameters:

- **aes** 初期化ベクトルを設定する AES 構造体へのポインタ
- **iv** AES 構造体を初期化するために使用される初期化ベクトル。値が NULL の場合、デフォルトの動作は iv を 0 に初期化します。

See:

- [wc_AesSetKeyDirect](#)
- [wc_AesSetKey](#)

Return:

- 0 初期化ベクトルの設定に成功した場合に返されます。
- BAD_FUNC_ARG AES ポインタが NULL の場合に返されます。

Example


```

Aes enc;
// enc key を設定
byte iv[] = { 16バイトの iv };
if (ret = wc_AesSetIV(&enc, iv) != 0) {
// aes iv の設定に失敗
}

```

B.12.2.3 function wc_AesCbcEncrypt

```

int wc_AesCbcEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)

```

入力バッファ in から平文メッセージを暗号化し、AES を使用した暗号ブロック連鎖により結果の暗号文を出力バッファ out に格納します。この関数は、メッセージを暗号化する前に AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、入力メッセージが AES ブロック長に整列していることを前提としており、入力長がブロック長の倍数であることを期待します。ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合、これはオプションでチェックおよび強制されます。ブロック倍数の入力を保証するために、事前に PKCS#7 スタイルのパディングを追加する必要があります。これは、自動的にパディングを追加する OpenSSL の AES-CBC メソッドとは異なります。wolfSSL と対応する OpenSSL 関数を相互運用させるには、OpenSSL コマンドライン関数で -nopad オプションを指定して、wolfSSL の AesCbcEncrypt メソッドのように動作し、暗号化中に余分なパディングを追加しないようにする必要があります。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 暗号化されたメッセージの暗号文を格納する出力バッファへのポインタ
- **in** 暗号化されるメッセージを含む入力バッファへのポインタ
- **sz** 入力メッセージのサイズ

See:

- [wc_AesInit](#)
- [wc_AesSetKey](#)
- [wc_AesSetIV](#)
- [wc_AesCbcDecrypt](#)

Return:

- 0 メッセージの暗号化に成功した場合に返されます。
- BAD_ALIGN_E ブロック整列エラーで返される場合があります。
- BAD_LENGTH_E ライブラリが WOLFSSL_AES_CBC_LENGTH_CHECKS でビルドされている場合、入力長が AES ブロック長の倍数でない場合に返されます。

Example

```

Aes enc;
int ret = 0;
// wc_AesInit と wc_AesSetKey を使用して enc を初期化、方向は
// AES_ENCRYPTION を使用
byte msg[AES_BLOCK_SIZE * n]; // 16 バイトの倍数
// msg にデータを入力
byte cipher[AES_BLOCK_SIZE * n]; // 16 バイトの倍数
if ((ret = wc_AesCbcEncrypt(&enc, cipher, message, sizeof(msg))) != 0) {

```



```
// ブロック整列エラー
}
```

B.12.2.4 function wc_AesCbcDecrypt

```
int wc_AesCbcDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

入力バッファ in から暗号を復号し、AES を使用した暗号ブロック連鎖により結果の平文を出力バッファ out に格納します。この関数は、メッセージを復号する前に AesSetKey を呼び出して AES 構造体が初期化されている必要があります。この関数は、元のメッセージが AES ブロック長に整列していたことを前提としており、入力長がブロック長の倍数であることを期待します。ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合、これはオプションでチェックおよび強制されます。これは、PKCS#7 パディングを自動的に追加する OpenSSL の AES-CBC メソッドとは異なり、ブロック倍数の入力を必要としません。wolfSSL 関数と同等の OpenSSL 関数を相互運用させるには、OpenSSL コマンドライン関数で -nopad オプションを指定して、wolfSSL の AesCbcEncrypt メソッドのように動作し、復号中にエラーを発生させないようにする必要があります。

Parameters:

- **aes** データを復号するために使用される AES オブジェクトへのポインタ。
- **out** 復号されたメッセージの平文を格納する出力バッファへのポインタ。サイズは AES_BLOCK_LENGTH の倍数である必要があります、必要に応じてパディングされます
- **in** 復号される暗号文を含む入力バッファへのポインタ。サイズは AES_BLOCK_LENGTH の倍数である必要があります、必要に応じてパディングされます
- **sz** 入力メッセージのサイズ。

See:

- [wc_AesInit](#)
- [wc_AesSetKey](#)
- [wc_AesCbcEncrypt](#)

Return:

- 0 メッセージの復号に成功した場合に返されます。
- BAD_ALIGN_E ブロック整列エラーで返される場合があります。
- BAD_LENGTH_E ライブラリが WOLFSSL_AES_CBC_LENGTH_CHECKS でビルドされている場合、入力長が AES ブロック長の倍数でない場合に返されます。

Example

```
Aes dec;
int ret = 0;
// wc_AesInit と wc_AesSetKey を使用して dec を初期化、方向は
// AES_DECRYPTION を使用
byte cipher[AES_BLOCK_SIZE * n]; // 16 バイトの倍数
// cipher に暗号文を入力
byte plain [AES_BLOCK_SIZE * n];
if ((ret = wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher))) != 0 ) {
// ブロック整列エラー
}
```

B.12.2.5 function wc_AesCtrEncrypt

```
int wc_AesCtrEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES を使用した CTR モードで、入力バッファ in からメッセージを暗号化/復号し、結果の暗号文を出力バッファ out に格納します。この関数は、コンパイル時に WOLFSSL_AES_COUNTER が有効になっている場合のみ有効です。この関数を呼び出す前に、AesSetKey を介して AES 構造体を初期化する必要があります。この関数は復号と暗号化の両方に使用されることに注意してください。注：暗号化と復号に同じ API を使用することについて。ユーザは暗号化/復号用の Aes 構造体を区別する必要があります。

Parameters:

- **aes** データを復号するために使用される AES オブジェクトへのポインタ
- **out** 暗号化されたメッセージの暗号文を格納する出力バッファへのポインタ サイズは AES_BLOCK_LENGTH の倍数である必要があり、必要に応じてパディングされます
- **in** 暗号化される平文を含む入力バッファへのポインタ サイズは AES_BLOCK_LENGTH の倍数である必要があり、必要に応じてパディングされます
- **sz** 入力平文のサイズ

See: [wc_AesSetKey](#)

Return: int wolfSSL エラーまたは成功ステータスに対応する整数値

Example

```
Aes enc;
Aes dec;
// wc_AesInit と wc_AesSetKeyDirect を使用して enc と dec を初期化、方向は
// AES_ENCRYPTION を使用。基盤となる API は暗号化のみを呼び出し、
// デフォルトで暗号に対して暗号化を呼び出すと暗号の復号が
// 行われるため

byte msg[AES_BLOCK_SIZE * n]; // n は正の整数で、msg を
16バイトの倍数にする
// plain にメッセージテキストを入力
byte cipher[AES_BLOCK_SIZE * n];
byte decrypted[AES_BLOCK_SIZE * n];
wc_AesCtrEncrypt(&enc, cipher, msg, sizeof(msg)); // plain を暗号化
wc_AesCtrEncrypt(&dec, decrypted, cipher, sizeof(cipher));
// 暗号文を復号
```

B.12.2.6 function wc_AesEncryptDirect

```
int wc_AesEncryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

この関数は、入力ブロック in の 1 ブロック暗号化を出力ブロック out に行います。提供された AES 構造体のキーを使用し、この関数を呼び出す前に wc_AesSetKey で初期化する必要があります。wc_AesSetKey は iv を NULL に設定して呼び出す必要があります。これは、構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 暗号化されたメッセージの暗号文を格納する出力バッファへのポインタ
- **in** 暗号化される平文を含む入力バッファへのポインタ

See:

- [wc_AesDecryptDirect](#)
- [wc_AesSetKeyDirect](#)

Return: int wolfSSL エラーまたは成功ステータスに対応する整数値

Example

```
Aes enc;
// wc_AesInit と wc_AesSetKey を使用して enc を初期化、方向は
// AES_ENCRYPTION を使用
byte msg [AES_BLOCK_SIZE]; // 16 バイト
// msg を暗号化する平文で初期化
byte cipher[AES_BLOCK_SIZE];
wc_AesEncryptDirect(&enc, cipher, msg);
```

B.12.2.7 function wc_AesDecryptDirect

```
int wc_AesDecryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

この関数は、入力ブロック in の 1 ブロック復号を出力ブロック out に行います。提供された AES 構造体のキーを使用し、この関数を呼び出す前に wc_AesSetKey で初期化する必要があります。wc_AesSetKey は iv を NULL に設定して呼び出す必要があります。これは、構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 復号された暗号文の平文を格納する出力バッファへのポインタ
- **in** 復号される暗号文を含む入力バッファへのポインタ

See:

- [wc_AesEncryptDirect](#)
- [wc_AesSetKeyDirect](#)

Return: int wolfSSL エラーまたは成功ステータスに対応する整数値

Example

```
Aes dec;
// wc_AesInit と wc_AesSetKey を使用して enc を初期化、方向は
// AES_DECRYPTION を使用
byte cipher [AES_BLOCK_SIZE]; // 16 バイト
// cipher を復号する暗号文で初期化
byte msg[AES_BLOCK_SIZE];
wc_AesDecryptDirect(&dec, msg, cipher);
```

B.12.2.8 function wc_AesSetKeyDirect

```
int wc_AesSetKeyDirect(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

この関数は、AES を使用した CTR モードの AES キーを設定するために使用されます。指定されたキー、iv（初期化ベクトル）、および暗号化 dir（方向）で AES オブジェクトを初期化します。構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。現在、wc_AesSetKeyDirect は内部的に wc_AesSetKey を使用しています。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **key** 暗号化と復号のための 16、24、または 32 バイトの秘密鍵
- **len** 渡されたキーの長さ
- **iv** キーの初期化に使用される初期化ベクトル
- **dir** 暗号の方向。暗号化する場合は AES_ENCRYPTION を設定し、復号する場合は AES_DECRYPTION を設定します。（wolfssl/wolfcrypt/aes.h の列挙型を参照）（注：Aes カウンタモード（ストリーム暗号）で wc_AesSetKeyDirect を使用する場合、暗号化と復号の両方に AES_ENCRYPTION のみを使用）

See:

- [wc_AesEncryptDirect](#)
- [wc_AesDecryptDirect](#)
- [wc_AesSetKey](#)

Return:

- 0 キーの設定に成功した場合に返されます。
- BAD_FUNC_ARG 指定されたキーの長さが無効な場合に返されます。

Example

```
Aes enc;
int ret = 0;
byte key[] = { 16、24、または32バイトのキー };
byte iv[] = { 16バイトの iv };

if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // aes key の初期化に失敗
}
if (ret = wc_AesSetKeyDirect(&enc, key, sizeof(key), iv,
AES_ENCRYPTION) != 0) {
    // aes key の設定に失敗
}
```

B.12.2.9 function wc_AesGcmSetKey

```
int wc_AesGcmSetKey(
    Aes * aes,
    const byte * key,
    word32 len
)
```

この関数は、AES GCM (Galois/Counter Mode) のキーを設定するために使用されます。指定されたキーで AES オブジェクトを初期化します。コンパイル時に構成オプション HAVE_AESGCM が有効になっている場合のみ有効です。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **key** 暗号化と復号のための 16、24、または 32 バイトの秘密鍵
- **len** 渡されたキーの長さ

See:

- [wc_AesGcmEncrypt](#)
- [wc_AesGcmDecrypt](#)

Return:

- 0 キーの設定に成功した場合に返されます。
- BAD_FUNC_ARG 指定されたキーの長さが無効な場合に返されます。

Example

```
Aes enc;
int ret = 0;
byte key[] = { 16、24、32バイトのキー };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // aes key の初期化に失敗
}
if (ret = wc_AesGcmSetKey(&enc, key, sizeof(key)) != 0) {
    // aes key の設定に失敗
}
```

B.12.2.10 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
)
```

この関数は、バッファ in に保持されている入力メッセージを暗号化し、結果の暗号文を出力バッファ out に格納します。暗号化の呼び出しごとに新しい iv (初期化ベクトル) が必要です。また、入力認証ベクトル authIn を認証タグ authTag にエンコードします。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 暗号文を格納する出力バッファへのポインタ サイズは in のサイズ (sz) と一致する必要があります
- **in** 暗号化するメッセージを保持する入力バッファへのポインタ サイズは AES_BLOCK_LENGTH の倍数である必要があります、必要に応じてパディングされます
- **sz** 暗号化する入力メッセージの長さ
- **iv** 初期化ベクトルを含むバッファへのポインタ

- **ivSz** 初期化ベクトルの長さ
- **authTag** 認証タグを格納するバッファへのポインタ
- **authTagSz** 希望する認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ
- **authInSz** 入力認証ベクトルの長さ

See:

- `wc_AesGcmSetKey`
- `wc_AesGcmDecrypt`

Return: 0 入力メッセージの暗号化に成功した場合に返されます

Example

```
Aes enc;
// wc_AesInit() と wc_AesGcmSetKey を呼び出して Aes 構造体を初期化

byte plain[AES_BLOCK_LENGTH * n]; // n は正の整数で
plain を16バイトの倍数にする
// plain を暗号化するメッセージで初期化
byte cipher[sizeof(plain)];
byte iv[] = // 16 バイトの iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // 認証ベクトル

wc_AesGcmEncrypt(&enc, cipher, plain, sizeof(cipher), iv, sizeof(iv),
    authTag, sizeof(authTag), authIn, sizeof(authIn));
```

B.12.2.11 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
)
```

この関数は、バッファ `in` に保持されている入力暗号文を復号し、結果のメッセージテキストを出力バッファ `out` に格納します。また、入力認証ベクトル `authIn` を、提供された認証タグ `authTag` と照合してチェックします。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** メッセージテキストを格納する出力バッファへのポインタ サイズは `in` のサイズ (`sz`) と一致する必要があります
- **in** 復号する暗号文を保持する入力バッファへのポインタ サイズは `AES_BLOCK_LENGTH` の倍数である必要があり、必要に応じてパディングされます
- **sz** 復号する暗号文の長さ
- **iv** 初期化ベクトルを含むバッファへのポインタ

- **ivSz** 初期化ベクトルの長さ
- **authTag** 認証タグを含むバッファへのポインタ
- **authTagSz** 希望する認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ
- **authInSz** 入力認証ベクトルの長さ

See:

- [wc_AesGcmSetKey](#)
- [wc_AesGcmEncrypt](#)

Return:

- 0 入力メッセージの復号と認証に成功した場合に返されます
- AES_GCM_AUTH_E 認証タグが提供された認証コードベクトル authTag と一致しない場合に返されます。

Example

```
Aes enc; // wc_AesGcmEncrypt に渡されたものと同じ構造体を使用可能
// まだ完了していない場合は、wc_AesInit と wc_AesGcmSetKey を呼び出して
// aes 構造体を初期化
```

```
byte cipher[AES_BLOCK_LENGTH * n]; // n は正の整数で
cipher を16バイトの倍数にする
// cipher を復号する暗号文で初期化
byte output[sizeof(cipher)];
byte iv[] = // 16 バイトの iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // 認証ベクトル
```

```
wc_AesGcmDecrypt(&enc, output, cipher, sizeof(cipher), iv, sizeof(iv),
    authTag, sizeof(authTag), authIn, sizeof(authIn));
```

B.12.2.12 function wc_GmacSetKey

```
int wc_GmacSetKey(
    Gmac * gmac,
    const byte * key,
    word32 len
)
```

この関数は、Galois メッセージ認証に使用される GMAC オブジェクトのキーを初期化および設定します。

Parameters:

- **gmac** 認証に使用される gmac オブジェクトへのポインタ
- **key** 認証のための 16、24、または 32 バイトの秘密鍵
- **len** キーの長さ

See:

- [wc_GmacUpdate](#)
- [wc_AesInit](#)

Return:

- 0 キーの設定に成功した場合に返されます
- BAD_FUNC_ARG キーの長さが無効な場合に返されます。

Example


```
Gmac gmac;
key[] = { 16、24、または32バイト長のキー };
wc_AesInit(gmac.aes, HEAP_HINT, INVALID_DEVID); // devId が更新されていることを確認
wc_GmacSetKey(&gmac, key, sizeof(key));
```

B.12.2.13 function wc_GmacUpdate

```
int wc_GmacUpdate(
    Gmac * gmac,
    const byte * iv,
    word32 ivSz,
    const byte * authIn,
    word32 authInSz,
    byte * authTag,
    word32 authTagSz
)
```

この関数は、authIn 入力の Gmac ハッシュを生成し、結果を authTag バッファに格納します。wc_GmacUpdate を実行した後、生成された authTag を既知の認証タグと比較して、メッセージの真正性を検証する必要があります。

Parameters:

- **gmac** 認証に使用される gmac オブジェクトへのポインタ
- **iv** ハッシュに使用される初期化ベクトル
- **ivSz** 使用される初期化ベクトルのサイズ
- **authIn** 検証する認証ベクトルを含むバッファへのポインタ
- **authInSz** 認証ベクトルのサイズ
- **authTag** Gmac ハッシュを格納する出力バッファへのポインタ
- **authTagSz** Gmac ハッシュを格納するために使用される出力バッファのサイズ

See:

- [wc_GmacSetKey](#)
- [wc_AesInit](#)

Return: 0 Gmac ハッシュの計算に成功した場合に返されます。

Example

```
Gmac gmac;
key[] = { 16、24、または32バイト長のキー };
iv[] = { 16バイト長の iv };

wc_AesInit(gmac.aes, HEAP_HINT, INVALID_DEVID); // devId が更新されていることを確認
wc_GmacSetKey(&gmac, key, sizeof(key));
authIn[] = { 16バイトの認証入力 };
tag[AES_BLOCK_SIZE]; // 認証コードを格納

wc_GmacUpdate(&gmac, iv, sizeof(iv), authIn, sizeof(authIn), tag,
sizeof(tag));
```

B.12.2.14 function wc_AesCcmSetKey

```
int wc_AesCcmSetKey(
    Aes * aes,
    const byte * key,
    word32 keySz
)
```


この関数は、CCM（Counter with CBC-MAC）を使用して AES オブジェクトのキーを設定します。AES 構造体へのポインタを受け取り、提供されたキーで初期化します。

Parameters:

- **aes** 提供されたキーを格納する aes 構造体
- **key** 暗号化と復号のための 16、24、または 32 バイトの秘密鍵
- **keySz** 提供されたキーのサイズ

See:

- [wc_AesCcmEncrypt](#)
- [wc_AesCcmDecrypt](#)

Return: none

Example

```
Aes enc;
key[] = { 16, 24, または32バイト長のキー };
```

```
wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID); // devId が更新されていることを確認
wc_AesCcmSetKey(&enc, key, sizeof(key));
```

B.12.2.15 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
)
```

この関数は、CCM（Counter with CBC-MAC）を使用して、入力メッセージ in を出力バッファ out に暗号化します。その後、authIn 入力から認証タグ authTag を計算して格納します。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 暗号文を格納する出力バッファへのポインタ
- **in** 暗号化するメッセージを保持する入力バッファへのポインタ
- **sz** 暗号化する入力メッセージの長さ
- **nonce** ナンス（1 回のみ使用される数値）を含むバッファへのポインタ
- **nonceSz** ナンスの長さ
- **authTag** 認証タグを格納するバッファへのポインタ
- **authTagSz** 希望する認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ
- **authInSz** 入力認証ベクトルの長さ

See:

- [wc_AesCcmSetKey](#)
- [wc_AesCcmDecrypt](#)

Return: none

Example

```
Aes enc;
// wc_AesInit と wc_AesCcmSetKey で enc を初期化

nonce[] = { ナンスを初期化 };
plain[] = { 平文メッセージ };
cipher[sizeof(plain)];

authIn[] = { 16バイトの認証入力 };
tag[AES_BLOCK_SIZE]; // 認証コードを格納

wc_AesCcmEncrypt(&enc, cipher, plain, sizeof(plain), nonce, sizeof(nonce),
    tag, sizeof(tag), authIn, sizeof(authIn));
```

B.12.2.16 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
)
```

この関数は、CCM（Counter with CBC-MAC）を使用して、入力暗号文 in を出力バッファ out に復号します。その後、authIn 入力から認証タグ authTag を計算します。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 暗号文を格納する出力バッファへのポインタ
- **in** 暗号化するメッセージを保持する入力バッファへのポインタ
- **sz** 復号する入力暗号文の長さ
- **nonce** ナンス（1 回のみ使用される数値）を含むバッファへのポインタ
- **nonceSz** ナンスの長さ
- **authTag** 認証タグを格納するバッファへのポインタ
- **authTagSz** 希望する認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ
- **authInSz** 入力認証ベクトルの長さ

See:

- [wc_AesCcmSetKey](#)
- [wc_AesCcmEncrypt](#)

Return:

- 0 入力メッセージの復号に成功した場合に返されます

- AES_CCM_AUTH_E 認証タグが提供された認証コードベクトル authTag と一致しない場合に返されます。

Example

```
Aes dec;
// wc_AesInit と wc_AesCcmSetKey で dec を初期化

nonce[] = { ナンスを初期化 };
cipher[] = { 暗号化されたメッセージ };
plain[sizeof(cipher)];

authIn[] = { 16バイトの認証入力 };
tag[AES_BLOCK_SIZE] = { 検証のために受信した認証タグ };

int return = wc_AesCcmDecrypt(&dec, plain, cipher, sizeof(cipher),
nonce, sizeof(nonce), tag, sizeof(tag), authIn, sizeof(authIn));
if(return != 0) {
// 復号エラー、無効な認証コード
}
```

B.12.2.17 function wc_AesXtsInit

```
int wc_AesXtsInit(
    XtsAes * aes,
    void * heap,
    int devId
)
```

これは AES-XTS コンテキストを初期化するためのものです。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。

Parameters:

- **aes** 暗号化/復号プロセスのための AES キー
- **heap** メモリに使用するヒープヒント。NULL でも可
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定

See:

- wc_AesXtsSetKey
- wc_AesXtsSetKeyNoInit
- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 成功

Example

```
XtsAes aes;

if(wc_AesXtsInit(&aes, NULL, INVALID_DEVID) != 0)
{
// エラーを処理
}
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION) != 0)
{
```

```

    // エラーを処理
}
wc_AesXtsFree(&aes);

```

B.12.2.18 function wc_AesXtsSetKeyNoInit

```

int wc_AesXtsSetKeyNoInit(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir
)

```

これは、最初に wc_AesXtsInit() を呼び出した後、キーを正しい暗号化または復号タイプに設定するのに役立ちます。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。

Parameters:

- **aes** 暗号化/復号プロセスのための AES キー
- **key** aes キー | tweak キーを保持するバッファ
- **len** キーバッファの長さ (バイト単位)。キーサイズの 2 倍である必要があります。つまり、16 バイトのキーの場合は 32 です。
- **dir** 方向、AES_ENCRYPTION または AES_DECRYPTION

See:

- [wc_AesXtsEncrypt](#)
- [wc_AesXtsDecrypt](#)
- [wc_AesXtsFree](#)

Return: 0 成功

Example

```

XtsAes aes;

if(wc_AesXtsInit(&aes, NULL, 0) != 0)
{
    // エラーを処理
}
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0)
    != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);

```

B.12.2.19 function wc_AesXtsSetKey

```

int wc_AesXtsSetKey(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir,
    void * heap,
    int devId
)

```

これは、キーを正しい暗号化または復号タイプに設定するのに役立ちます。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。

Parameters:

- **aes** 暗号化/復号プロセスのための AES キー
- **key** aes キー | tweak キーを保持するバッファ
- **len** キーバッファの長さ (バイト単位)。キーサイズの 2 倍である必要があります。つまり、16 バイトのキーの場合は 32 です。
- **dir** 方向、AES_ENCRYPTION または AES_DECRYPTION
- **heap** メモリに使用するヒープヒント。NULL でも可
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定

See:

- [wc_AesXtsInit](#)
- [wc_AesXtsSetKeyNoInit](#)
- [wc_AesXtsEncrypt](#)
- [wc_AesXtsDecrypt](#)
- [wc_AesXtsFree](#)

Return: 0 成功

Example

```
XtsAes aes;
```

```
if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, INVALID_DEVID)
    ↪ != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

B.12.2.20 function wc_AesXtsEncryptSector

```
int wc_AesXtsEncryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

wc_AesXtsEncrypt と同じプロセスですが、バイト配列の代わりに word64 型を tweak 値として使用します。これは word64 をバイト配列に変換し、wc_AesXtsEncrypt を呼び出すだけです。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 暗号文を保持する出力バッファ
- **in** 暗号化する入力平文バッファ
- **sz** out と in バッファの両方のサイズ
- **sector** tweak に使用する値

See:

- [wc_AesXtsEncrypt](#)
- [wc_AesXtsDecrypt](#)

- `wc_AesXtsInit`
- `wc_AesXtsSetKeyNoInit`
- `wc_AesXtsSetKey`
- `wc_AesXtsFree`

Return: 0 成功

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
word64 s = VALUE;

// AES_ENCRYPTION を dir としてキーを設定
if(wc_AesXtsEncryptSector(&aes, cipher, plain, SIZE, s) != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

B.12.2.21 function `wc_AesXtsDecryptSector`

```
int wc_AesXtsDecryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

`wc_AesXtsDecrypt` と同じプロセスですが、バイト配列の代わりに `word64` 型を `tweak` 値として使用します。これは `word64` をバイト配列に変換するだけです。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 平文を保持する出力バッファ
- **in** 復号する入力暗号文バッファ
- **sz** `out` と `in` バッファの両方のサイズ
- **sector** `tweak` に使用する値

See:

- `wc_AesXtsEncrypt`
- `wc_AesXtsDecrypt`
- `wc_AesXtsInit`
- `wc_AesXtsSetKeyNoInit`
- `wc_AesXtsSetKey`
- `wc_AesXtsFree`

Return: 0 成功

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
word64 s = VALUE;
```

// AES_DECRYPTION を dir として aes キーを設定し、tweak は AES_ENCRYPTION で設定

```
if(wc_AesXtsDecryptSector(&aes, plain, cipher, SIZE, s) != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

B.12.2.22 function wc_AesXtsEncrypt

```
int wc_AesXtsEncrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

XTS モードの AES。(XTS) Tweak と Cipher Text Stealing を使用した XEX 暗号化。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 暗号文を保持する出力バッファ
- **in** 暗号化する入力平文バッファ
- **sz** out と in バッファの両方のサイズ
- **i** tweak に使用する値
- **iSz** i バッファのサイズ、常に AES_BLOCK_SIZE である必要がありますが、この入力を持つことで、ユーザが関数を呼び出す方法についてのサニティチェックが追加されます。

See:

- [wc_AesXtsDecrypt](#)
- [wc_AesXtsInit](#)
- [wc_AesXtsSetKeyNoInit](#)
- [wc_AesXtsSetKey](#)
- [wc_AesXtsFree](#)

Return: 0 成功

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];

// AES_ENCRYPTION を dir としてキーを設定

if(wc_AesXtsEncrypt(&aes, cipher, plain, SIZE, i, sizeof(i)) != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

B.12.2.23 function wc_AesXtsDecrypt

```
int wc_AesXtsDecrypt(
```

```

    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)

```

暗号化と同じプロセスですが、Aes キーは AES_DECRYPTION タイプです。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 平文を保持する出力バッファ
- **in** 復号する入力暗号文バッファ
- **sz** out と in バッファの両方のサイズ
- **i** tweak に使用する値
- **iSz** i バッファのサイズ、常に AES_BLOCK_SIZE である必要がありますが、この入力を持つことで、ユーザが関数を呼び出す方法についてのサニティチェックが追加されます。

See:

- [wc_AesXtsEncrypt](#)
- [wc_AesXtsInit](#)
- [wc_AesXtsSetKeyNoInit](#)
- [wc_AesXtsSetKey](#)
- [wc_AesXtsFree](#)

Return: 0 成功

Example

```

XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];

```

```

// AES_DECRYPTION を dir としてキーを設定し、tweak は AES_ENCRYPTION で設定

```

```

if(wc_AesXtsDecrypt(&aes, plain, cipher, SIZE, i, sizeof(i)) != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);

```

B.12.2.24 function wc_AesXtsFree

```

int wc_AesXtsFree(
    XtsAes * aes
)

```

これは、XtsAes 構造体によって使用されるリソースを解放するためのものです

Parameters:

- **aes** 解放する AES キー

See:

- [wc_AesXtsEncrypt](#)

- `wc_AesXtsDecrypt`
- `wc_AesXtsInit`
- `wc_AesXtsSetKeyNoInit`
- `wc_AesXtsSetKey`

Return: 0 成功

Example

```
XtsAes aes;
```

```
if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0) != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

B.12.2.25 function `wc_AesInit`

```
int wc_AesInit(
    Aes * aes,
    void * heap,
    int devId
)
```

Aes 構造体を初期化します。使用するヒープヒントと非同期ハードウェアで使用する ID を設定します。使用が完了したら、ユーザが Aes 構造体に対して `wc_AesFree` を呼び出す必要があります。

Parameters:

- **aes** 初期化する aes 構造体
- **heap** 必要に応じて malloc / free に使用するヒープヒント
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は `INVALID_DEVID` (-2) に設定

See:

- `wc_AesSetKey`
- `wc_AesSetIV`
- `wc_AesFree`

Return: 0 成功

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; // 非同期を使用しない場合は INVALID_DEVID がデフォルト

// 使用する場合はここでヒープヒントを設定可能

wc_AesInit(&enc, hint, devId);
```

B.12.2.26 function `wc_AesFree`

```
void wc_AesFree(
    Aes * aes
)
```

該当する場合、Aes 構造体に関連付けられたリソースを解放します。内部的には時々 no-op になることもありますが、新しい環境で使用するためにアプリケーションコードが移植される場合（呼び出しが適用される場合）など、一般的なベストプラクティスとしてすべてのケースで呼び出すことをお勧めします。

Parameters:

- **aes** 解放する aes 構造体

See: `wc_AesInit`

Return: no return (void 関数)

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; // 非同期を使用しない場合は INVALID_DEVID がデフォルト

// 使用する場合はここでヒープヒントを設定可能

wc_AesInit(&enc, hint, devId);
// ... 興味深いことをいくつか行う ...
wc_AesFree(&enc);
```

B.12.2.27 function wc_AesCfbEncrypt

```
int wc_AesCfbEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

CFB モードの AES。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 暗号文を保持する出力バッファ（少なくとも入力バッファと同じ大きさである必要があります）
- **in** 暗号化する入力平文バッファ
- **sz** 入力バッファのサイズ

See:

- `wc_AesCfbDecrypt`
- `wc_AesSetKey`

Return: 0 成功、失敗時は負のエラー値

Example

```
Aes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];

// 暗号化と復号の両方に AES_ENCRYPTION を dir としてキーを設定

if(wc_AesCfbEncrypt(&aes, cipher, plain, SIZE) != 0)
{
    // エラーを処理
}
```

B.12.2.28 function wc_AesCfbDecrypt

```
int wc_AesCfbDecrypt(  
    Aes * aes,  
    byte * out,  
    const byte * in,  
    word32 sz  
)
```

CFB モードの AES。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 復号されたテキストを保持する出力バッファ（少なくとも入力バッファと同じ大きさである必要があります）
- **in** 復号する入力バッファ
- **sz** 入力バッファのサイズ

See:

- [wc_AesCfbEncrypt](#)
- [wc_AesSetKey](#)

Return: 0 成功、失敗時は負のエラー値

Example

```
Aes aes;  
unsigned char plain[SIZE];  
unsigned char cipher[SIZE];
```

```
// 暗号化と復号の両方に AES_ENCRYPTION を dir としてキーを設定
```

```
if(wc_AesCfbDecrypt(&aes, plain, cipher, SIZE) != 0)  
{  
    // エラーを処理  
}
```

B.12.2.29 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  
)
```

この関数は、RFC 5297 で説明されている SIV（合成初期化ベクトル）暗号化を実行します。

Parameters:

- **key** 使用するキーを含むバイトバッファ。
- **keySz** キーバッファの長さ（バイト単位）。

- **assoc** 追加の認証された関連データ (AD)。
- **assocSz** AD バッファの長さ (バイト単位)。
- **nonce** 1 回のみ使用される数値。アルゴリズムによって AD と同じ方法で使用されます。
- **nonceSz** nonce バッファの長さ (バイト単位)。
- **in** 暗号化する平文バッファ。
- **inSz** 平文バッファの長さ。
- **siv** S2V によって出力される SIV (RFC 5297 2.4 を参照)。
- **out** 暗号文を保持するバッファ。平文バッファと同じ長さである必要があります。

See: [wc_AesSivDecrypt](#)

Return:

- 0 暗号化に成功した場合。
- BAD_FUNC_ARG key、SIV、または出力バッファが NULL の場合に返されます。また、キーサイズが 32、48、または 64 バイトでない場合にも返されます。
- Other AES または CMAC 操作が失敗した場合に返されるその他の負のエラー値。

Example

```
byte key[] = { 32、48、または64バイトのキー };
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) {
    // 暗号化に失敗
}
```

B.12.2.30 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
)
```

この関数は、RFC 5297 で説明されている SIV (合成初期化ベクトル) 復号を実行します。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

Parameters:

- **key** 使用するキーを含むバイトバッファ。
- **keySz** キーバッファの長さ (バイト単位)。
- **assoc** 追加の認証された関連データ (AD)。
- **assocSz** AD バッファの長さ (バイト単位)。
- **nonce** 1 回のみ使用される数値。基盤となるアルゴリズムによって AD と同じ方法で使用されます。
- **nonceSz** nonce バッファの長さ (バイト単位)。

- **in** 復号する暗号文バッファ。
- **inSz** 暗号文バッファの長さ。
- **siv** 暗号文に付随する SIV (RFC 5297 2.4 を参照)。
- **out** 復号された平文を保持するバッファ。暗号文バッファと同じ長さである必要があります。

See: [wc_AesSivEncrypt](#)

Return:

- 0 復号に成功した場合。
- BAD_FUNC_ARG key、SIV、または出力バッファが NULL の場合に返されます。また、キーサイズが 32、48、または 64 バイトでない場合にも返されます。
- AES_SIV_AUTH_E S2V によって導出された SIV が入力 SIV と一致しない場合 (RFC 5297 2.7 を参照)。
- Other AES または CMAC 操作が失敗した場合に返されるその他の負のエラー値。

Example

```
byte key[] = { 32, 48, または64バイトのキー };
byte assoc[] = {0x01, 0x2, 0x3};
byte nonce[] = {0x04, 0x5, 0x6};
byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte siv[AES_BLOCK_SIZE] = { 暗号文に付属していた SIV };
byte plainText[sizeof(cipherText)];
if (wc_AesSivDecrypt(key, sizeof(key), assoc, sizeof(assoc), nonce,
    sizeof(nonce), cipherText, sizeof(cipherText), siv, plainText) != 0) {
    // 復号に失敗
}
```

B.12.2.31 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
)
```

この関数は、「EAX: A Conventional Authenticated-Encryption Mode」(<https://eprint.iacr.org/2003/069>) で説明されている AES EAX 暗号化と認証を実行します。これは、すべての暗号化と認証操作を 1 つの関数呼び出しで実行する「ワンショット」API です。

Parameters:

- **key** 使用するキーを含むバッファ
- **keySz** キーバッファの長さ (バイト単位)
- **out** 暗号文を保持するバッファ。平文バッファと同じ長さである必要があります
- **in** 暗号化する平文バッファ
- **inSz** 平文バッファの長さ
- **nonce** EAX 操作に使用する暗号ナンス
- **nonceSz** nonce バッファの長さ (バイト単位)
- **authTag** 認証タグを格納するバッファへのポインタ

- **authTagSz** 希望する認証タグの長さ
- **authIn** 認証する入力データを含むバッファへのポインタ
- **authInSz** 入力認証データの長さ

See: `wc_AesEaxDecryptAuth`

Return:

- 0 暗号化に成功した場合。
- BAD_FUNC_ARG 入力または出力バッファが NULL の場合に返されます。また、キーサイズが有効な AES キーサイズ (16、24、または 32 バイト) でない場合にも返されます
- other AES または CMAC 操作が失敗した場合に返されるその他の負のエラー値。

Example

```
byte key[] = { 32、48、または64バイトのキー };
byte nonce[] = {0x04, 0x5, 0x6};
byte plainText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte cipherText[sizeof(plainText)]; // 出力暗号文
byte authTag[length, up to AES_BLOCK_SIZE]; // 出力 authTag

if (wc_AesEaxEncrypt(key, sizeof(key),
                    cipherText, sizeof(plainText),
                    nonce, sizeof(nonce),
                    authTag, sizeof(authTag),
                    authIn, sizeof(authIn)) != 0) {
    // 暗号化に失敗
}
```

B.12.2.32 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
)
```

この関数は、「EAX: A Conventional Authenticated-Encryption Mode」(<https://eprint.iacr.org/2003/069>)で説明されている AES EAX 復号と認証を実行します。これは、すべての復号と認証操作を 1 つの関数呼び出しで実行する「ワンショット」API です。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

Parameters:

- **key** 使用するキーを含むバイトバッファ
- **keySz** キーバッファの長さ (バイト単位)
- **out** 平文を保持するバッファ。入力暗号文バッファと同じ長さである必要があります

- **in** 復号する暗号文バッファ
- **inSz** 暗号文バッファの長さ
- **nonce** EAX 操作に使用する暗号ナンス
- **nonceSz** nonce バッファの長さ (バイト単位)
- **authTag** データの真正性をチェックするために照合する認証タグを保持するバッファ
- **authTagSz** 入力認証タグの長さ
- **authIn** 認証する入力データを含むバッファへのポインタ
- **authInSz** 入力認証データの長さ

See: [wc_AesEaxEncryptAuth](#)

Return:

- 0 復号に成功した場合
- BAD_FUNC_ARG 入力または出力バッファが NULL の場合に返されます。また、キーサイズが有効な AES キーサイズ (16、24、または 32 バイト) でない場合にも返されます
- AES_EAX_AUTH_E 認証タグが提供された認証コードベクトル authTag と一致しない場合
- other AES または CMAC 操作が失敗した場合に返されるその他の負のエラー値。

Example

```
byte key[] = { 32、48、または64バイトのキー };
byte nonce[] = {0x04, 0x5, 0x6};
byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte plainText[sizeof(cipherText)]; // 出力平文
byte authTag[length, up to AES_BLOCK_SIZE]; // 出力 authTag

if (wc_AesEaxDecrypt(key, sizeof(key),
                    cipherText, plainText, sizeof(plainText),
                    nonce, sizeof(nonce),
                    authTag, sizeof(authTag),
                    authIn, sizeof(authIn)) != 0) {
    // 暗号化に失敗
}
```

B.12.2.33 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
)
```

この関数は、認証付き暗号化または復号で使用する AesEax オブジェクトを初期化します。この関数は、AES EAX 増分 API のいずれかを使用する前に、AesEax オブジェクトに対して呼び出す必要があります。ワンショット EAX API 関数を使用する場合は呼び出す必要はありません。この関数で初期化されたすべての AesEax インスタンスは、インスタンスの使用が完了したら wc_AesEaxFree() の呼び出しで解放する必要があります。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体

- **key** 暗号化と復号のための 16、24、または 32 バイトの秘密鍵
- **keySz** 提供されたキーの長さ (バイト単位)
- **nonce** EAX 操作に使用する暗号ナンス
- **nonceSz** nonce バッファの長さ (バイト単位)
- **authIn** (オプション) 認証ストリームに追加する入力データ 使用しない場合、この引数は NULL である必要があります
- **authInSz** 入力認証データのサイズ (バイト単位)

See:

- `wc_AesEaxEncryptUpdate`
- `wc_AesEaxDecryptUpdate`
- `wc_AesEaxAuthDataUpdate`
- `wc_AesEaxEncryptFinal`
- `wc_AesEaxDecryptFinal`
- `wc_AesEaxFree`

Return:

- 0 成功した場合
- error code 失敗した場合のエラーコード

Example

```

AesEax eax;
key[] = { 16、24、または32バイト長のキー };
nonce[] = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
plainText[] = {暗号化する平文データ};

cipherText[sizeof(plainText)]; // cipherText を保持するバッファ
authTag[length, up to AES_BLOCK_SIZE]; // 計算された認証データを保持するバッファ

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        authIn, authInSz)) != 0) {
    goto cleanup;
}

// さらに認証データを追加したい場合は、この時点で提供できます
// そうでない場合は、authIn パラメータに NULL を使用し、authIn サイズは 0 です
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);

```


B.12.2.34 function wc_AesEaxEncryptUpdate

```
WOLFSSL_API int wc_AesEaxEncryptUpdate(
    AesEax * eax,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
    word32 authInSz
)
```

この関数は、AES EAX を使用して入力データを暗号化し、オプションで認証ストリームにさらに入力データを追加します。eax は、[wc_AesEaxInit](#) の呼び出しで事前に初期化されている必要があります。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体
- **out** 暗号文を保持する出力バッファ
- **in** 暗号化する平文を保持する入力バッファ
- **inSz** 入力データバッファのサイズ (バイト単位)
- **authIn** (オプション) 認証ストリームに追加する入力データ 使用しない場合、この引数は NULL である必要があります
- **authInSz** 入力認証データのサイズ (バイト単位)

See:

- [wc_AesEaxInit](#)
- [wc_AesEaxDecryptUpdate](#)
- [wc_AesEaxAuthDataUpdate](#)
- [wc_AesEaxEncryptFinal](#)
- [wc_AesEaxDecryptFinal](#)
- [wc_AesEaxFree](#)

Return:

- 0 成功した場合
- error code 失敗した場合のエラーコード

Example

```
AesEax eax;
key[] = { 16、24、または32バイト長のキー };
nonce[] = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
plainText[] = {暗号化する平文データ};

cipherText[sizeof(plainText)]; // cipherText を保持するバッファ
authTag[length, up to AES_BLOCK_SIZE]; // 計算された認証データを保持するバッファ

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        authIn, authInSz)) != 0) {
    goto cleanup;
}

// さらに認証データを追加したい場合は、この時点で提供できます
```

```
// そうでない場合は、authIn パラメータに NULL を使用し、authInSz は 0 です
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);
```

B.12.2.35 function wc_AesEaxDecryptUpdate

```
WOLFSSL_API int wc_AesEaxDecryptUpdate(
    AesEax * eax,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
    word32 authInSz
)
```

この関数は、AES EAX を使用して入力データを復号し、オプションで認証ストリームにさらに入力データを追加します。eax は、[wc_AesEaxInit](#) の呼び出しで事前に初期化されている必要があります。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体
- **out** 復号された平文を保持する出力バッファ
- **in** 暗号文を保持する入力バッファ
- **inSz** 入力データバッファのサイズ (バイト単位)
- **authIn** (オプション) 認証ストリームに追加する入力データ 使用しない場合、この引数は NULL である必要があります
- **authInSz** 入力認証データのサイズ (バイト単位)

See:

- [wc_AesEaxInit](#)
- [wc_AesEaxEncryptUpdate](#)
- [wc_AesEaxAuthDataUpdate](#)
- [wc_AesEaxEncryptFinal](#)
- [wc_AesEaxDecryptFinal](#)
- [wc_AesEaxFree](#)

Return:

- 0 成功した場合
- error code 失敗した場合のエラーコード

Example

```
AesEax eax;
key[] = { 16、24、または32バイト長のキー };
nonce[] = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
cipherText[] = { 暗号化されたデータ};
```

```

plaintext[sizeof(cipherText)]; // 復号されたデータを保持するバッファ
// 認証タグは、暗号化 AEAD 操作によって別の場所で生成されます
authTag[length, up to AES_BLOCK_SIZE] = { 認証タグ };

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        authIn, authInSz)) != 0) {
    goto cleanup;
}

// さらに認証データを追加したい場合は、この時点で提供できます
// そうでない場合は、authIn パラメータに NULL を使用し、authInSz は 0 です
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);

```

B.12.2.36 function wc_AesEaxAuthDataUpdate

```

WOLFSSL_API int wc_AesEaxAuthDataUpdate(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)

```

この関数は、認証ストリームに入力データを追加します。eax は、`wc_AesEaxInit` の呼び出しで事前に初期化されている必要があります。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体
- **authIn** 認証ストリームに追加する入力データ
- **authInSz** 入力認証データのサイズ (バイト単位)

See:

- `wc_AesEaxInit`
- `wc_AesEaxEncryptUpdate`
- `wc_AesEaxDecryptUpdate`
- `wc_AesEaxEncryptFinal`
- `wc_AesEaxDecryptFinal`
- `wc_AesEaxFree`

Return:

- 0 成功した場合

- error code 失敗した場合のエラーコード

Example

```
AesEax eax;
key[] = { 16、24、または32バイト長のキー };
nonce[] = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
cipherText[] = {暗号化されたデータ};

plainText[sizeof(cipherText)]; // 復号されたデータを保持するバッファ
// 認証タグは、暗号化 AEAD 操作によって別の場所で生成されます
authTag[length, up to AES_BLOCK_SIZE] = { 認証タグ };

AesEax eax;

// ここでは追加する認証データなし
if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        NULL, 0)) != 0) {
    goto cleanup;
}

// ここでは追加する認証データなし、後で wc_AesEaxAuthDataUpdate で追加
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);
```

B.12.237 function wc_AesEaxEncryptFinal

```
WOLFSSL_API int wc_AesEaxEncryptFinal(
    AesEax * eax,
    byte * authTag,
    word32 authTagSz
)
```

この関数は、暗号化 AEAD 操作を完了し、現在の認証ストリームに対して認証タグを生成します。eax は、wc_AesEaxInit の呼び出しで事前に初期化されている必要があります。AesEax コンテキスト構造体の使用が完了したら、wc_AesEaxFree を使用して必ず解放してください。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体

- **authTag[out]** 計算された認証タグを保持するバッファ
- **authTagSz** authTag のサイズ (バイト単位)

See:

- `wc_AesEaxInit`
- `wc_AesEaxEncryptUpdate`
- `wc_AesEaxDecryptUpdate`
- `wc_AesEaxAuthDataUpdate`
- `wc_AesEaxDecryptFinal`
- `wc_AesEaxFree`

Return:

- 0 成功した場合
- error code 失敗した場合のエラーコード

Example

```
AesEax eax;
key[] = { 16、24、または32バイト長のキー };
nonce[] = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
plainText[] = { 暗号化する平文データ };
cipherText[sizeof(plainText)]; // cipherText を保持するバッファ
authTag[length, up to AES_BLOCK_SIZE]; // 計算された認証データを保持するバッファ
```

```
AesEax eax;
```

```
if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        authIn, authInSz)) != 0) {
    goto cleanup;
}

// さらに認証データを追加したい場合は、この時点で提供できます
// そうでない場合は、authIn パラメータに NULL を使用し、authInSz は 0 です
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);
```

B.12.2.38 function wc_AesEaxDecryptFinal

```
WOLFSSL_API int wc_AesEaxDecryptFinal(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)
```

この関数は、復号 AEAD 操作を完了し、認証タグの計算を完了して、ユーザが提供したタグに対してその有効性をチェックします。eax は、`wc_AesEaxInit` の呼び出しで事前に初期化されている必要があります。AesEax コンテキスト構造体の使用が完了したら、`wc_AesEaxFree` を使用して必ず解放してください。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体
- **authIn** 計算された認証タグと照合するための入力認証タグ
- **authInSz** authIn のサイズ (バイト単位)

See:

- `wc_AesEaxInit`
- `wc_AesEaxEncryptUpdate`
- `wc_AesEaxDecryptUpdate`
- `wc_AesEaxAuthDataUpdate`
- `wc_AesEaxEncryptFinal`
- `wc_AesEaxFree`

Return:

- 0 データが正常に認証された場合
- AES_EAX_AUTH_E 認証タグが提供された認証コードベクトル authIn と一致しない場合
- other error code 失敗した場合のその他のエラーコード

Example

```
AesEax eax;
key[] = { 16、24、または32バイト長のキー };
nonce[] = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
cipherText[] = {暗号化されたデータ};

plainText[sizeof(cipherText)]; // 復号されたデータを保持するバッファ
// 認証タグは、暗号化 AEAD 操作によって別の場所で生成されます
authTag[length, up to AES_BLOCK_SIZE] = { 認証タグ };

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        authIn, authInSz)) != 0) {
    goto cleanup;
}

// さらに認証データを追加したい場合は、この時点で提供できます
// そうでない場合は、authIn パラメータに NULL を使用し、authInSz は 0 です
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);
```

B.12.2.39 function wc_AesEaxFree

```
WOLFSSL_API int wc_AesEaxFree(
    AesEax * eax
)
```

この関数は、AesEax ラッパー構造体内の Aes インスタンスによって使用されるリソース、特にキーを解放します。wc_AesEaxInit で初期化された後、すべての必要な EAX 操作が完了した時点で AesEax 構造体に対して呼び出す必要があります。

Parameters:

- **eax** 解放する AES EAX インスタンス

See:

- [wc_AesEaxInit](#)
- [wc_AesEaxEncryptUpdate](#)
- [wc_AesEaxDecryptUpdate](#)
- [wc_AesEaxAuthDataUpdate](#)
- [wc_AesEaxEncryptFinal](#)
- [wc_AesEaxDecryptFinal](#)

Return: 0 成功

Example

```
AesEax eax;
```

```
if(wc_AesEaxInit(eax, key, keySz, nonce, nonceSz, authIn, authInSz) != 0) {
    // エラーを処理し、その後解放
    wc_AesEaxFree(&eax);
}
```

B.12.2.40 function wc_AesCtsEncrypt

```
int wc_AesCtsEncrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * iv
)
```

この関数は、CTS モードを使用して AES 暗号化を実行します。これは、すべての操作を 1 回の呼び出しで処理するワンショット API です。

Parameters:

- **key** 暗号化に使用される AES キーへのポインタ。
- **keySz** AES キーのサイズ (バイト単位) (16、24、または 32 バイト)。
- **out** 暗号化された暗号文を保持するバッファ。少なくとも入力と同じサイズである必要があります。
- **in** 暗号化する平文入力データへのポインタ。
- **inSz** 平文入力データのサイズ (バイト単位)。
- **iv** 暗号化に使用される初期化ベクトル (IV) へのポインタ。16 バイトである必要があります。
- **key** 暗号化に使用される AES キーへのポインタ。

- **keySz** AES キーのサイズ (バイト単位) (16、24、または 32 バイト)。
- **out** 暗号化された暗号文を保持するバッファ。少なくとも入力平文と同じサイズである必要があります。
- **in** 暗号化する平文入力データへのポインタ。
- **inSz** 平文入力データのサイズ (バイト単位)。
- **iv** 暗号化に使用される初期化ベクトル (IV) へのポインタ。16 バイトである必要があります。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) {
    // 暗号化エラーを処理
}
```

See:

- [wc_AesCtsDecrypt](#)
- [wc_AesCtsDecrypt](#)

Return:

- 0 暗号化に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。
- other negative error codes 暗号化失敗のためのその他の負のエラーコード。
- 0 暗号化に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。
- other negative error codes 暗号化失敗のためのその他の負のエラーコード。

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) {
    // 暗号化エラーを処理
}
```

B.12.2.41 function wc_AesCtsDecrypt

```
int wc_AesCtsDecrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * iv
)
```

この関数は、CTS モードを使用して AES 復号を実行します。これは、すべての操作を 1 回の呼び出しで処理するワンショット API です。

Parameters:

- **key** 復号に使用される AES キーへのポインタ。
- **keySz** AES キーのサイズ (バイト単位) (16、24、または 32 バイト)。
- **out** 復号された平文を保持するバッファ。少なくとも入力暗号文と同じサイズである必要があります。
- **in** 復号する暗号文入力データへのポインタ。
- **inSz** 暗号文入力データのサイズ (バイト単位)。
- **iv** 復号に使用される初期化ベクトル (IV) へのポインタ。16 バイトである必要があります。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) {
    // 復号エラーを処理
}
```

See: [wc_AesCtsEncrypt](#)

Return:

- 0 復号に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。
- other negative error codes 復号失敗のためのその他の負のエラーコード。

B.12.2.42 function wc_AesCtsEncryptUpdate

```
int wc_AesCtsEncryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)
```

この関数は、AES CTS 暗号化の更新ステップを実行します。平文のチャンクを処理し、中間データを保存します。

Parameters:

- **aes** 操作のコンテキストを保持する Aes 構造体へのポインタ。
- **out** 暗号化された暗号文を保持するバッファ。この更新ステップからの出力を保存するのに十分な大きさである必要があります。
- **outSz** out バッファに書き込まれた出力データのサイズ (バイト単位)。入力時には、out バッファに書き込むことができる最大バイト数を含める必要があります。
- **in** 暗号化する平文入力データへのポインタ。
- **inSz** 平文入力データのサイズ (バイト単位)。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) {
    // エラーを処理
}
wc_AesFree(&aes);

```

See: [wc_AesCtsDecryptUpdate](#)

Return:

- 0 処理に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。

B.12.2.43 function wc_AesCtsEncryptFinal

```

int wc_AesCtsEncryptFinal(
    Aes * aes,
    byte * out,
    word32 * outSz
)

```

この関数は、AES CTS 暗号化操作を完了します。残りの平文を処理し、暗号化を完了します。

Parameters:

- **aes** 操作のコンテキストを保持する Aes 構造体へのポインタ。
- **out** 最終的な暗号化された暗号文を保持するバッファ。この最終ステップから残りの暗号文を保存するのに十分な大きさである必要があります。
- **outSz** out バッファに書き込まれた出力データのサイズ (バイト単位)。入力時には、out バッファに書き込むことができる最大バイト数を含める必要があります。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);
// wc_AesCtsEncryptUpdate を使用して必要な更新ステップを実行
int ret = wc_AesCtsEncryptFinal(&aes, ciphertext, &outSz);
if (ret != 0) {
    // エラーを処理
}
wc_AesFree(&aes);

```

See: [wc_AesCtsDecryptFinal](#)

Return:

- 0 暗号化の完了に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。

B.12.2.44 function wc_AesCtsDecryptUpdate

```

int wc_AesCtsDecryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)

```

)

この関数は、AES CTS 復号の更新ステップを実行します。暗号文のチャンクを処理し、中間データを保存します。

Parameters:

- **aes** 操作のコンテキストを保持する Aes 構造体へのポインタ。
- **out** 復号された平文を保持するバッファ。この更新ステップからの出力を保存するのに十分な大きさである必要があります。
- **outSz** out バッファに書き込まれた出力データのサイズ (バイト単位)。入力時には、out バッファに書き込むことができる最大バイト数を含める必要があります。
- **in** 復号する暗号文入力データへのポインタ。
- **inSz** 暗号文入力データのサイズ (バイト単位)。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) {
    // エラーを処理
}
wc_AesFree(&aes);
```

See: [wc_AesCtsEncryptUpdate](#)

Return:

- 0 処理に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。

B.12.2.45 function wc_AesCtsDecryptFinal

```
int wc_AesCtsDecryptFinal(
    Aes * aes,
    byte * out,
    word32 * outSz
)
```

この関数は、AES CTS 復号操作を完了します。残りの暗号文を処理し、復号を完了します。

Parameters:

- **aes** 操作のコンテキストを保持する Aes 構造体へのポインタ。
- **out** 最終的な復号された平文を保持するバッファ。この最終ステップから残りの平文を保存するのに十分な大きさである必要があります。
- **outSz** out バッファに書き込まれた出力データのサイズ (バイト単位)。入力時には、out バッファに書き込むことができる最大バイト数を含める必要があります。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);
// wc_AesCtsDecryptUpdate を使用して必要な更新ステップを実行
int ret = wc_AesCtsDecryptFinal(&aes, plaintext, &outSz);
if (ret != 0) {
    // エラーを処理
}
wc_AesFree(&aes);

```

See: [wc_AesCtsEncryptFinal](#)

Return:

- 0 復号の完了に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。

B.12.2.46 function wc_AesCbcDecryptWithKey

```

int wc_AesCbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    const byte * iv
)

```

入力バッファ in から暗号を復号し、結果の平文を AES を使用した暗号ブロック連鎖を使用して出力バッファ out に格納します。この関数は、AES 構造体を初期化する必要はありません。代わりに、キーと iv(初期化ベクトル)を受け取り、これらを使用して AES オブジェクトを初期化し、暗号文を復号します。

Parameters:

- **out** 復号されたメッセージの平文を格納する出力バッファへのポインタ
- **in** 復号する暗号文を含む入力バッファへのポインタ
- **inSz** 入力メッセージのサイズ
- **key** 復号用の 16、24、または 32 バイトの秘密鍵
- **keySz** 復号に使用するキーのサイズ

See:

- [wc_AesSetKey](#)
- [wc_AesSetIV](#)
- [wc_AesCbcEncrypt](#)
- [wc_AesCbcDecrypt](#)

Return:

- 0 メッセージの復号に成功した場合
- BAD_ALIGN_E ブロックアライメントエラーが発生した場合に返されます
- BAD_FUNC_ARG AesSetIV 中にキー長が無効または AES オブジェクトが null の場合に返されます
- MEMORY_E WOLFSSL_SMALL_STACK が有効で、XMALLOC が AES オブジェクトのインスタンス化に失敗した場合に返されます。

Example

```

int ret = 0;
byte key[] = { 16、24、または32バイトのキー };

```

```
byte iv[] = { 16バイトの iv };
byte cipher[AES_BLOCK_SIZE * n]; //n は正の整数で、cipher を 16 バイトの倍数にする
// cipher を暗号文で埋める
byte plain [AES_BLOCK_SIZE * n];
if ((ret = wc_AesCbcDecryptWithKey(plain, cipher, AES_BLOCK_SIZE, key,
AES_BLOCK_SIZE, iv)) != 0 ) {
// 復号エラー
}
```

B.13 アルゴリズム - ARC4

B.13.1 Functions

	Name
int	wc_Arc4Process (Arc4 * arc4, byte * out, const byte * in, word32 length) この関数は、バッファ in から入力メッセージを暗号化し、暗号文を出力バッファ out に格納するか、またはバッファ in から暗号文を復号し、平文を出力バッファ out に格納します。ARC4 暗号化を使用します。この関数は暗号化と復号の両方に使用されます。このメソッドを呼び出す前に、wc_Arc4SetKey を使用して ARC4 構造体を初期化する必要があります。
int	wc_Arc4SetKey (Arc4 * arc4, const byte * key, word32 length) この関数は、ARC4 オブジェクトのキーを設定し、暗号として使用するために初期化します。wc_Arc4Process で暗号化に使用する前に呼び出す必要があります。

B.13.2 Functions Documentation

B.13.2.1 function wc_Arc4Process

```
int wc_Arc4Process(
    Arc4 * arc4,
    byte * out,
    const byte * in,
    word32 length
)
```

この関数は、バッファ in から入力メッセージを暗号化し、暗号文を出力バッファ out に格納するか、またはバッファ in から暗号文を復号し、平文を出力バッファ out に格納します。ARC4 暗号化を使用します。この関数は暗号化と復号の両方に使用されます。このメソッドを呼び出す前に、wc_Arc4SetKey を使用して ARC4 構造体を初期化する必要があります。

Parameters:

- **arc4** メッセージを処理するために使用される ARC4 構造体へのポインタ
- **out** 処理されたメッセージを格納する出力バッファへのポインタ
- **in** 処理するメッセージを含む入力バッファへのポインタ
- **length** 処理するメッセージの長さ

See: [wc_Arc4SetKey](#)

Return: none

Example

```
Arc4 enc;
byte key[] = { 暗号化に使用するキー };
wc_Arc4SetKey(&enc, key, sizeof(key));

byte plain[] = { エンコードする平文 };
byte cipher[sizeof(plain)];
byte decrypted[sizeof(plain)];
// plain を cipher に暗号化
wc_Arc4Process(&enc, cipher, plain, sizeof(plain));
// cipher を復号
wc_Arc4Process(&enc, decrypted, cipher, sizeof(cipher));
```

B.13.2.2 function wc_Arc4SetKey

```
int wc_Arc4SetKey(
    Arc4 * arc4,
    const byte * key,
    word32 length
)
```

この関数は、ARC4 オブジェクトのキーを設定し、暗号として使用するために初期化します。wc_Arc4Process で暗号化に使用する前に呼び出す必要があります。

Parameters:

- **arc4** 暗号化に使用される arc4 構造体へのポインタ
- **key** arc4 構造体を初期化するために使用するキー
- **length** arc4 構造体を初期化するために使用するキーの長さ

See: [wc_Arc4Process](#)

Return: none

Example

```
Arc4 enc;
byte key[] = { 暗号化に使用するキーで初期化 };
wc_Arc4SetKey(&enc, key, sizeof(key));
```

B.14 アルゴリズム - BLAKE2**B.14.1 Functions**

	Name
int	wc_InitBlake2b (Blake2b * b2b, word32 digestSz) この関数は、Blake2 ハッシュ関数で使用するための Blake2b 構造体を初期化します。
int	wc_Blake2bUpdate (Blake2b * b2b, const byte * data, word32 sz) この関数は、指定された入力データで Blake2b ハッシュを更新します。この関数は wc_InitBlake2b の後に呼び出され、最終ハッシュ (wc_Blake2bFinal) の準備ができるまで繰り返されます。

	Name
int	wc_Blake2bFinal (Blake2b * b2b, byte * final, word32 requestSz) この関数は、以前に提供された入力データの Blake2b ハッシュを計算します。出力ハッシュの長さは requestSz になります。requestSz==0 の場合は、b2b 構造体の digestSz が使用されます。この関数は、wc_InitBlake2b の後、および必要な各入力データに対して wc_Blake2bUpdate が処理された後に呼び出す必要があります。

B.14.2 Functions Documentation

B.14.2.1 function wc_InitBlake2b

```
int wc_InitBlake2b(  
    Blake2b * b2b,  
    word32 digestSz  
)
```

この関数は、Blake2 ハッシュ関数で使用するための Blake2b 構造体を初期化します。

Parameters:

- **b2b** 初期化する Blake2b 構造体へのポインタ
- **digestSz** 実装する blake 2 ダイジェストの長さ

See: [wc_Blake2bUpdate](#)

Return: 0 Blake2b 構造体の初期化とダイジェストサイズの設定に成功した場合に返されます。

Example

```
Blake2b b2b;  
// 64 バイトダイジェストで Blake2b 構造体を初期化  
wc_InitBlake2b(&b2b, 64);
```

B.14.2.2 function wc_Blake2bUpdate

```
int wc_Blake2bUpdate(  
    Blake2b * b2b,  
    const byte * data,  
    word32 sz  
)
```

この関数は、指定された入力データで Blake2b ハッシュを更新します。この関数は wc_InitBlake2b の後に呼び出され、最終ハッシュ (wc_Blake2bFinal) の準備ができるまで繰り返されます。

Parameters:

- **b2b** 更新する Blake2b 構造体へのポインタ
- **data** 追加するデータを含むバッファへのポインタ
- **sz** 追加する入力データの長さ

See:

- [wc_InitBlake2b](#)
- [wc_Blake2bFinal](#)

Return:

- 0 指定されたデータで Blake2b 構造体の更新に成功した場合に返されます
- -1 入力データの圧縮中に失敗した場合に返されます

Example

```
int ret;
Blake2b b2b;
// 64 バイトダイジェストで Blake2b 構造体を初期化
wc_InitBlake2b(&b2b, 64);

byte plain[] = { // 入力を初期化 };

ret = wc_Blake2bUpdate(&b2b, plain, sizeof(plain));
if( ret != 0 ) {
    // blake2b の更新エラー
}
```

B.14.2.3 function wc_Blake2bFinal

```
int wc_Blake2bFinal(
    Blake2b * b2b,
    byte * final,
    word32 requestSz
)
```

この関数は、以前に提供された入力データの Blake2b ハッシュを計算します。出力ハッシュの長さは requestSz になります。requestSz==0 の場合は、b2b 構造体の digestSz が使用されます。この関数は、wc_InitBlake2b の後、および必要な各入力データに対して wc_Blake2bUpdate が処理された後に呼び出す必要があります。

Parameters:

- **b2b** 更新する Blake2b 構造体へのポインタ
- **final** blake2b ハッシュを格納するバッファへのポインタ。requestSz の長さである必要があります
- **requestSz** 計算するダイジェストの長さ。これがゼロの場合、代わりに b2b->digestSz が使用されます

See:

- [wc_InitBlake2b](#)
- [wc_Blake2bUpdate](#)

Return:

- 0 Blake2b ハッシュの計算に成功した場合に返されます
- -1 Blake2b ハッシュの解析中に失敗した場合に返されます

Example

```
int ret;
Blake2b b2b;
byte hash[64];
// 64 バイトダイジェストで Blake2b 構造体を初期化
wc_InitBlake2b(&b2b, 64);
... // wc_Blake2bUpdate を呼び出してハッシュにデータを追加

ret = wc_Blake2bFinal(&b2b, hash, 64);
if( ret != 0 ) {
    // blake2b ハッシュの生成エラー
}
```


B.15 アルゴリズム - Camellia

B.15.1 Functions

	Name
int	wc_CamelliaSetKey (wc_Camellia * cam, const byte * key, word32 len, const byte * iv) この関数は、camellia オブジェクトのキーと初期化ベクトルを設定し、暗号として使用するために初期化します。
int	wc_CamelliaSetIV (wc_Camellia * cam, const byte * iv) この関数は、camellia オブジェクトの初期化ベクトルを設定します。
int	wc_CamelliaEncryptDirect (wc_Camellia * cam, byte * out, const byte * in) この関数は、提供された camellia オブジェクトを使用して 1 ブロックの暗号化を行います。バッファ in から最初の 16 バイトブロックを解析し、暗号化された結果をバッファ out に格納します。この関数を使用する前に、wc_CamelliaSetKey を使用して camellia オブジェクトを初期化する必要があります。
int	wc_CamelliaDecryptDirect (wc_Camellia * cam, byte * out, const byte * in) この関数は、提供された camellia オブジェクトを使用して 1 ブロックの復号を行います。バッファ in から最初の 16 バイトブロックを解析し、復号して、結果をバッファ out に格納します。この関数を使用する前に、wc_CamelliaSetKey を使用して camellia オブジェクトを初期化する必要があります。
int	wc_CamelliaCbcEncrypt (wc_Camellia * cam, byte * out, const byte * in, word32 sz) この関数は、バッファ in から平文を暗号化し、出力をバッファ out に格納します。暗号ブロック連鎖 (CBC) モードの Camellia を使用してこの暗号化を実行します。
int	wc_CamelliaCbcDecrypt (wc_Camellia * cam, byte * out, const byte * in, word32 sz) この関数は、バッファ in から暗号文を復号し、出力をバッファ out に格納します。暗号ブロック連鎖 (CBC) モードの Camellia を使用してこの復号を実行します。

B.15.2 Functions Documentation

B.15.2.1 function wc_CamelliaSetKey

```
int wc_CamelliaSetKey(  
    wc_Camellia * cam,  
    const byte * key,  
    word32 len,  
    const byte * iv  
)
```

この関数は、camellia オブジェクトのキーと初期化ベクトルを設定し、暗号として使用するために初期化します。

Parameters:

- **cam** キーと iv を設定する camellia 構造体へのポインタ
- **key** 暗号化と復号に使用する 16、24、または 32 バイトのキーを含むバッファへのポインタ
- **len** 渡されるキーの長さ
- **iv** この camellia 構造体で使用する 16 バイトの初期化ベクトルを含むバッファへのポインタ

See:

- [wc_CamelliaEncryptDirect](#)
- [wc_CamelliaDecryptDirect](#)
- [wc_CamelliaCbcEncrypt](#)
- [wc_CamelliaCbcDecrypt](#)

Return:

- 0 キーと初期化ベクトルの設定に成功した場合に返されます
- BAD_FUNC_ARG 入力引数の 1 つの処理中にエラーが発生した場合に返されます
- MEMORY_E_XMALLOC でメモリ割り当て中にエラーが発生した場合に返されます

Example

```
Camellia cam;
byte key[32];
// キーを初期化
byte iv[16];
// iv を初期化
if( wc_CamelliaSetKey(&cam, key, sizeof(key), iv) != 0) {
    // camellia 構造体の初期化エラー
}
```

B.15.2.2 function wc_CamelliaSetIV

```
int wc_CamelliaSetIV(
    wc_Camellia * cam,
    const byte * iv
)
```

この関数は、camellia オブジェクトの初期化ベクトルを設定します。

Parameters:

- **cam** iv を設定する camellia 構造体へのポインタ
- **iv** この camellia 構造体で使用する 16 バイトの初期化ベクトルを含むバッファへのポインタ

See: [wc_CamelliaSetKey](#)

Return:

- 0 キーと初期化ベクトルの設定に成功した場合に返されます
- BAD_FUNC_ARG 入力引数の 1 つの処理中にエラーが発生した場合に返されます

Example

```
Camellia cam;
byte iv[16];
// iv を初期化
if( wc_CamelliaSetIV(&cam, iv) != 0) {
```

```
// camellia 構造体の初期化エラー  
}
```

B.15.2.3 function wc_CamelliaEncryptDirect

```
int wc_CamelliaEncryptDirect(  
    wc_Camellia * cam,  
    byte * out,  
    const byte * in  
)
```

この関数は、提供された camellia オブジェクトを使用して 1 ブロックの暗号化を行います。バッファ in から最初の 16 バイトブロックを解析し、暗号化された結果をバッファ out に格納します。この関数を使用する前に、wc_CamelliaSetKey を使用して camellia オブジェクトを初期化する必要があります。

Parameters:

- **cam** 暗号化に使用する camellia 構造体へのポインタ
- **out** 暗号化されたブロックを格納するバッファへのポインタ
- **in** 暗号化する平文ブロックを含むバッファへのポインタ

See: [wc_CamelliaDecryptDirect](#)

Return: none 戻り値なし。

Example

```
Camellia cam;  
// キーと iv で cam 構造体を初期化  
byte plain[] = { // 暗号化するメッセージで初期化 };  
byte cipher[16];  
  
wc_CamelliaEncryptDirect(&cam, cipher, plain);
```

B.15.2.4 function wc_CamelliaDecryptDirect

```
int wc_CamelliaDecryptDirect(  
    wc_Camellia * cam,  
    byte * out,  
    const byte * in  
)
```

この関数は、提供された camellia オブジェクトを使用して 1 ブロックの復号を行います。バッファ in から最初の 16 バイトブロックを解析し、復号して、結果をバッファ out に格納します。この関数を使用する前に、wc_CamelliaSetKey を使用して camellia オブジェクトを初期化する必要があります。

Parameters:

- **cam** 暗号化に使用する camellia 構造体へのポインタ
- **out** 復号された平文ブロックを格納するバッファへのポインタ
- **in** 復号する暗号文ブロックを含むバッファへのポインタ

See: [wc_CamelliaEncryptDirect](#)

Return: none 戻り値なし。

Example

```
Camellia cam;  
// キーと iv で cam 構造体を初期化  
byte cipher[] = { // 復号する暗号化されたメッセージで初期化 };
```

```
byte decrypted[16];  
  
wc_CamelliaDecryptDirect(&cam, decrypted, cipher);
```

B.15.2.5 function wc_CamelliaCbcEncrypt

```
int wc_CamelliaCbcEncrypt(  
    wc_Camellia * cam,  
    byte * out,  
    const byte * in,  
    word32 sz  
)
```

この関数は、バッファ in から平文を暗号化し、出力をバッファ out に格納します。暗号ブロック連鎖 (CBC) モードの Camellia を使用してこの暗号化を実行します。

Parameters:

- **cam** 暗号化に使用する camellia 構造体へのポインタ
- **out** 暗号化された暗号文を格納するバッファへのポインタ
- **in** 暗号化する平文を含むバッファへのポインタ
- **sz** 暗号化するメッセージのサイズ

See: [wc_CamelliaCbcDecrypt](#)

Return: none 戻り値なし。

Example

```
Camellia cam;  
// キーと iv で cam 構造体を初期化  
byte plain[] = { // 復号する暗号化されたメッセージで初期化 };  
byte cipher[sizeof(plain)];  
  
wc_CamelliaCbcEncrypt(&cam, cipher, plain, sizeof(plain));
```

B.15.2.6 function wc_CamelliaCbcDecrypt

```
int wc_CamelliaCbcDecrypt(  
    wc_Camellia * cam,  
    byte * out,  
    const byte * in,  
    word32 sz  
)
```

この関数は、バッファ in から暗号文を復号し、出力をバッファ out に格納します。暗号ブロック連鎖 (CBC) モードの Camellia を使用してこの復号を実行します。

Parameters:

- **cam** 暗号化に使用する camellia 構造体へのポインタ
- **out** 復号されたメッセージを格納するバッファへのポインタ
- **in** 暗号化された暗号文を含むバッファへのポインタ
- **sz** 暗号化するメッセージのサイズ

See: [wc_CamelliaCbcEncrypt](#)

Return: none 戻り値なし。

Example

```
Camellia cam;
// キーと iv で cam 構造体を初期化
byte cipher[] = { // 復号する暗号化されたメッセージで初期化 };
byte decrypted[sizeof(cipher)];

wc_CamelliaCbcDecrypt(&cam, decrypted, cipher, sizeof(cipher));
```

B.16 アルゴリズム - ChaCha

B.16.1 Functions

	Name
int	wc_Chacha_SetIV (ChaCha * ctx, const byte * inIv, word32 counter) この関数は、ChaCha オブジェクトの初期化ベクトル (nonce) を設定し、暗号として使用するために初期化します。 wc_Chacha_SetKey を使用してキーが設定された後に呼び出す必要があります。暗号化のラウンドごとに異なる nonce を使用する必要があります。
int	wc_Chacha_Process (ChaCha * ctx, byte * cipher, const byte * plain, word32 msglen) この関数は、バッファ input からテキストを処理し、暗号化または復号して、結果をバッファ output に格納します。
int	wc_Chacha_SetKey (ChaCha * ctx, const byte * key, word32 keySz) この関数は、ChaCha オブジェクトのキーを設定し、暗号として使用するために初期化します。wc_Chacha_SetIV で nonce を設定する前、および wc_Chacha_Process で暗号化に使用する前に呼び出す必要があります。

B.16.2 Functions Documentation

B.16.2.1 function wc_Chacha_SetIV

```
int wc_Chacha_SetIV(
    ChaCha * ctx,
    const byte * inIv,
    word32 counter
)
```

この関数は、ChaCha オブジェクトの初期化ベクトル (nonce) を設定し、暗号として使用するために初期化します。wc_Chacha_SetKey を使用してキーが設定された後に呼び出す必要があります。暗号化のラウンドごとに異なる nonce を使用する必要があります。

Parameters:

- **ctx** iv を設定する ChaCha 構造体へのポインタ
- **inIv** ChaCha 構造体を初期化するための 12 バイトの初期化ベクトルを含むバッファへのポインタ
- **counter** ブロックカウンタが開始すべき値-通常はゼロ。

See:

- [wc_Chacha_SetKey](#)
- [wc_Chacha_Process](#)

Return:

- 0 初期化ベクトルの設定に成功した場合に返されます
- BAD_FUNC_ARG ctx 入力引数の処理中にエラーが発生した場合に返されます

Example

```
ChaCha enc;
// wc_Chacha_SetKey で enc を初期化
byte iv[12];
// iv を初期化
if( wc_Chacha_SetIV(&enc, iv, 0) != 0) {
    // ChaCha 構造体の初期化エラー
}
```

B.16.2.2 function wc_Chacha_Process

```
int wc_Chacha_Process(
    ChaCha * ctx,
    byte * cipher,
    const byte * plain,
    word32 msglen
)
```

この関数は、バッファ input からテキストを処理し、暗号化または復号して、結果をバッファ output に格納します。

Parameters:

- **ctx** iv を設定する ChaCha 構造体へのポインタ
- **output** 出力暗号文または復号された平文を格納するバッファへのポインタ
- **input** 暗号化する入力平文または復号する入力暗号文を含むバッファへのポインタ
- **msglen** 暗号化するメッセージまたは復号する暗号文の長さ

See:

- [wc_Chacha_SetKey](#)
- [wc_Chacha_Process](#)

Return:

- 0 入力の暗号化または復号に成功した場合に返されます
- BAD_FUNC_ARG ctx 入力引数の処理中にエラーが発生した場合に返されます

Example

```
ChaCha enc;
// wc_Chacha_SetKey と wc_Chacha_SetIV で enc を初期化

byte plain[] = { // 平文を初期化 };
byte cipher[sizeof(plain)];
if( wc_Chacha_Process(&enc, cipher, plain, sizeof(plain)) != 0) {
    // ChaCha 暗号の処理エラー
}
```

B.16.2.3 function wc_Chacha_SetKey

```
int wc_Chacha_SetKey(
    ChaCha * ctx,
    const byte * key,
    word32 keySz
)
```

)

この関数は、ChaCha オブジェクトのキーを設定し、暗号として使用するために初期化します。wc_Chacha_SetIV で nonce を設定する前、および wc_Chacha_Process で暗号化に使用する前に呼び出す必要があります。

Parameters:

- **ctx** キーを設定する ChaCha 構造体へのポインタ
- **key** ChaCha 構造体を初期化するための 16 バイトまたは 32 バイトのキーを含むバッファへのポインタ
- **keySz** 渡されるキーの長さ

See:

- [wc_Chacha_SetIV](#)
- [wc_Chacha_Process](#)

Return:

- 0 キーの設定に成功した場合に返されます
- BAD_FUNC_ARG ctx 入力引数の処理中にエラーが発生した場合、またはキーが 16 バイトまたは 32 バイトでない場合に返されます

Example

```
ChaCha enc;
byte key[] = { // キーを初期化 };

if( wc_Chacha_SetKey(&enc, key, sizeof(key)) != 0 ) {
    // ChaCha 構造体の初期化エラー
}
```

B.17 アルゴリズム - ChaCha20_Poly1305

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 outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]) この関数は、入力メッセージ inPlaintext を ChaCha20 ストリーム暗号を使用して暗号化し、出力バッファ outCiphertext に格納します。また、Poly_1305 認証 (暗号文に対して) を実行し、生成された認証タグを出力バッファ outAuthTag に格納します。

	Name
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) この関数は、入力暗号文 inCiphertext を ChaCha20 ストリーム暗号を使用して復号し、出力バッファ outPlaintext に格納します。また、Poly_1305 認証を実行し、指定された inAuthTag と inAAD(任意の長さの追加認証データ) で生成された認証を比較します。ゼロ以外のエラーコードが返された場合、出力データ outPlaintext は未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

B.17.2 Functions Documentation

B.17.2.1 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]
)
```

この関数は、入力メッセージ inPlaintext を ChaCha20 ストリーム暗号を使用して暗号化し、出力バッファ outCiphertext に格納します。また、Poly-1305 認証 (暗号文に対して) を実行し、生成された認証タグを出力バッファ outAuthTag に格納します。

Parameters:

- **inKey** 暗号化に使用する 32 バイトのキーを含むバッファへのポインタ
- **inIV** 暗号化に使用する 12 バイトの iv を含むバッファへのポインタ
- **inAAD** 任意の長さの追加認証データ (AAD) を含むバッファへのポインタ
- **inAADLen** 入力 AAD の長さ
- **inPlaintext** 暗号化する平文を含むバッファへのポインタ
- **inPlaintextLen** 暗号化する平文の長さ
- **outCiphertext** 暗号文を格納するバッファへのポインタ
- **outAuthTag** 認証タグを格納する 16 バイト幅のバッファへのポインタ

See:

- **wc_Chacha20Poly1305_Decrypt**
- wc_Chacha_*
- wc_Poly1305*

Return:

- 0 メッセージの暗号化に成功した場合に返されます
- BAD_FUNC_ARG 暗号化プロセス中にエラーが発生した場合に返されます

Example

```

byte key[] = { // 32 バイトのキーを初期化 };
byte iv[] = { // 12 バイトのキーを初期化 };
byte inAAD[] = { // AAD を初期化 };

byte plain[] = { // 暗号化するメッセージを初期化 };
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) {
    // 暗号化実行エラー
}

```

B.17.2.2 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
)

```

この関数は、入力暗号文 inCiphertext を ChaCha20 ストリーム暗号を使用して復号し、出力バッファ outPlaintext に格納します。また、Poly-1305 認証を実行し、指定された inAuthTag と inAAD(任意の長さの追加認証データ) で生成された認証を比較します。ゼロ以外のエラーコードが返された場合、出力データ outPlaintext は未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

Parameters:

- **inKey** 復号に使用する 32 バイトのキーを含むバッファへのポインタ
- **inIV** 復号に使用する 12 バイトの iv を含むバッファへのポインタ
- **inAAD** 任意の長さの追加認証データ (AAD) を含むバッファへのポインタ
- **inAADLen** 入力 AAD の長さ
- **inCiphertext** 復号する暗号文を含むバッファへのポインタ
- **outCiphertextLen** 復号する暗号文の長さ
- **inAuthTag** 認証用の 16 バイトのダイジェストを含むバッファへのポインタ
- **outPlaintext** 平文を格納するバッファへのポインタ

See:

- [wc_Chacha20Poly1305_Encrypt](#)
- [wc_Chacha_*](#)
- [wc_Poly1305*](#)

Return:

- 0 メッセージの復号と認証に成功した場合に返されます

- BAD_FUNC_ARG 関数の引数が期待される内容と一致しない場合に返されます
- MAC_CMP_FAILED_E 生成された認証タグが提供された inAuthTag と一致しない場合に返されます
- MEMORY_E 内部バッファの割り当てに失敗した場合に返されます
- CHACHA_POLY_OVERFLOW 入力が破損している場合に返される可能性があります

Example

```

byte key[]    = { // 32 バイトのキーを初期化 };
byte iv[]     = { // 12 バイトのキーを初期化 };
byte inAAD[]  = { // AAD を初期化 };

byte cipher[] = { // 受信した暗号文で初期化 };
byte authTag[16] = { // 受信した認証タグで初期化 };

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) {
    // 認証中のエラー
} else if( ret != 0) {
    // 関数引数のエラー
}

```

B.18 Callbacks コールバック - CryptoCb**B.19 アルゴリズム - Curve25519****B.19.1 Functions**

	Name
int	wc_curve25519_make_key (WC_RNG * rng, int keysize, curve25519_key * key) この関数は、与えられた乱数生成器 rng を使用して、与えられたサイズ (keysize) の Curve25519 鍵を生成し、与えられた curve25519_key 構造体に格納します。wc_curve25519_init() を通じて鍵構造体が初期化された後に呼び出す必要があります。
int	wc_curve25519_shared_secret (curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen) この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンのみをサポートします。
int	wc_curve25519_shared_secret_ex (curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen, int endian) この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンとリトルエンディアンの両方をサポートします。

	Name
int	wc_curve25519_init (curve25519_key * key) この関数は Curve25519 鍵を初期化します。構造体の鍵を生成する前に呼び出す必要があります。
void	wc_curve25519_free (curve25519_key * key) この関数は Curve25519 オブジェクトを解放します。
int	wc_curve25519_import_private (const byte * priv, word32 privSz, curve25519_key * key) この関数は curve25519 秘密鍵のみをインポートします。(ビッグエンディアン)
int	wc_curve25519_import_private_ex (const byte * priv, word32 privSz, curve25519_key * key, int endian) curve25519 秘密鍵のみのインポート。(ビッグエンディアンまたはリトルエンディアン)
int	wc_curve25519_import_private_raw (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key) この関数は、公開鍵-秘密鍵ペアを curve25519_key 構造体にインポートします。ビッグエンディアンのみ。
int	wc_curve25519_import_private_raw_ex (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key, int endian) この関数は、公開鍵-秘密鍵ペアを curve25519_key 構造体にインポートします。ビッグエンディアンとリトルエンディアンの両方をサポートします。
int	wc_curve25519_export_private_raw (curve25519_key * key, byte * out, word32 * outLen) この関数は、curve25519_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンのみ。
int	wc_curve25519_export_private_raw_ex (curve25519_key * key, byte * out, word32 * outLen, int endian) この関数は、curve25519_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンまたはリトルエンディアンを指定できます。
int	wc_curve25519_import_public (const byte * in, word32 inLen, curve25519_key * key) この関数は、与えられた in バッファから公開鍵をインポートし、curve25519_key 構造体に格納します。
int	wc_curve25519_import_public_ex (const byte * in, word32 inLen, curve25519_key * key, int endian) この関数は、与えられた in バッファから公開鍵をインポートし、curve25519_key 構造体に格納します。

	Name
int	wc_curve25519_check_public (const byte * pub, word32 pubSz, int endian) この関数は、エンディアン順序を考慮して、公開鍵バッファが有効な Curve25519 鍵値を保持しているかどうかをチェックします。
int	wc_curve25519_export_public (curve25519_key * key, byte * out, word32 * outLen) この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。
int	wc_curve25519_export_public_ex (curve25519_key * key, byte * out, word32 * outLen, int endian) この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンとリトルエンディアンの両方をサポートします。
int	wc_curve25519_export_key_raw (curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) Curve25519 鍵ペアをエクスポートします。ビッグエンディアンのみ。
int	wc_curve25519_export_key_raw_ex (curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian) curve25519 鍵ペアをエクスポートします。ビッグエンディアンまたはリトルエンディアン。
int	wc_curve25519_size (curve25519_key * key) この関数は、与えられた鍵構造体の鍵サイズを返します。

B.19.2 Functions Documentation

B.19.2.1 function wc_curve25519_make_key

```
int wc_curve25519_make_key(
    WC_RNG * rng,
    int keysize,
    curve25519_key * key
)
```

この関数は、与えられた乱数生成器 rng を使用して、与えられたサイズ (keysizes) の Curve25519 鍵を生成し、与えられた curve25519_key 構造体に格納します。wc_curve25519_init() を通じて鍵構造体が初期化された後に呼び出す必要があります。

Parameters:

- **rng** ecc 鍵を生成するために使用される RNG オブジェクトへのポインタ。
- **keysizes** 生成する鍵のサイズ。curve25519 では 32 バイトである必要があります。
- **key** 生成された鍵を格納する curve25519_key 構造体へのポインタ。

See: [wc_curve25519_init](#)

Return:

- 0 鍵の生成に成功し、与えられた curve25519_key 構造体に格納された場合に返されます。
- ECC_BAD_ARG_E 入力 keysize が curve25519 鍵の keysize に対応していない場合 (32 バイト) に返されます。

- RNG_FAILURE_E rng 内部ステータスが DRBG_OK でない場合、または rng で次のランダムブロックを生成する際にエラーがある場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

curve25519_key key;
wc_curve25519_init(&key); // 鍵を初期化
WC_RNG rng;
wc_InitRng(&rng); // 乱数生成器を初期化

ret = wc_curve25519_make_key(&rng, 32, &key);
if (ret != 0) {
    // Curve25519 鍵作成エラー
}
```

B.19.2.2 function wc_curve25519_shared_secret

```
int wc_curve25519_shared_secret(
    curve25519_key * private_key,
    curve25519_key * public_key,
    byte * out,
    word32 * outlen
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンのみをサポートします。

Parameters:

- **private_key** ユーザーの秘密鍵で初期化された curve25519_key 構造体へのポインタ。
- **public_key** 受信した公開鍵を含む curve25519_key 構造体へのポインタ。
- **out** 32 バイトの計算された秘密鍵を格納するバッファへのポインタ。
- **outlen** 出力バッファに書き込まれた長さを格納するポインタ。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_make_key](#)
- [wc_curve25519_shared_secret_ex](#)

Return:

- 0 共有秘密鍵の計算に成功した場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E 実装フィンガープリントを回避するために、公開鍵の最初のビットが設定されている場合に返されます。

Example

```
int ret;

byte sharedKey[32];
word32 keySz;
curve25519_key privKey, pubKey;
// 両方の鍵を初期化

ret = wc_curve25519_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
```

```
if (ret != 0) {
    // 共有鍵生成エラー
}
```

B.19.2.3 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
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **private_key** ユーザーの秘密鍵で初期化された curve25519_key 構造体へのポインタ。
- **public_key** 受信した公開鍵を含む curve25519_key 構造体へのポインタ。
- **out** 32 バイトの計算された秘密鍵を格納するバッファへのポインタ。
- **outlen** 出力バッファに書き込まれた長さを格納するポインタ。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_make_key](#)
- [wc_curve25519_shared_secret](#)

Return:

- 0 共有秘密鍵の計算に成功した場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E 実装フィンガープリントを回避するために、公開鍵の最初のビットが設定されている場合に返されます。

Example

```
int ret;

byte sharedKey[32];
word32 keySz;

curve25519_key privKey, pubKey;
// 両方の鍵を初期化

ret = wc_curve25519_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
    EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 共有鍵生成エラー
}
```

B.19.2.4 function wc_curve25519_init

```
int wc_curve25519_init(
    curve25519_key * key
)
```

この関数は Curve25519 鍵を初期化します。構造体の鍵を生成する前に呼び出す必要があります。

Parameters:

- **key** 初期化する curve25519_key 構造体へのポインタ。

See: [wc_curve25519_make_key](#)

Return:

- 0 curve25519_key 構造体の初期化に成功した場合に返されます。
- BAD_FUNC_ARG key が NULL の場合に返されます。

Example

```
curve25519_key key;  
wc_curve25519_init(&key); // 鍵を初期化  
// 鍵を作成し、暗号化に進む
```

B.19.2.5 function wc_curve25519_free

```
void wc_curve25519_free(  
    curve25519_key * key  
)
```

この関数は Curve25519 オブジェクトを解放します。

Parameters:

- **key** 解放する鍵オブジェクトへのポインタ。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_make_key](#)

Example

```
curve25519_key privKey;  
// 鍵を初期化し、共有秘密鍵の生成に使用  
wc_curve25519_free(&privKey);
```

B.19.2.6 function wc_curve25519_import_private

```
int wc_curve25519_import_private(  
    const byte * priv,  
    word32 privSz,  
    curve25519_key * key  
)
```

この関数は curve25519 秘密鍵のみをインポートします。(ビッグエンディアン)

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。

See:

- [wc_curve25519_import_private_ex](#)
- [wc_curve25519_size](#)

Return:

- 0 秘密鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG key または priv が null の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE25519_KEY_SIZE と等しくない場合に返されます。

Example

```
int ret;

byte priv[] = { // 秘密鍵の内容 };
curve25519_key key;
wc_curve25519_init(&key);

ret = wc_curve25519_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.19.2.7 function wc_curve25519_import_private_ex

```
int wc_curve25519_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve25519_key * key,
    int endian
)
```

curve25519 秘密鍵のみのインポート。(ビッグエンディアンまたはリトルエンディアン)

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- [wc_curve25519_import_private](#)
- [wc_curve25519_size](#)

Return:

- 0 秘密鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG key または priv が null の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE25519_KEY_SIZE と等しくない場合に返されます。

Example

```
int ret;

byte priv[] = { // 秘密鍵の内容 };
curve25519_key key;
wc_curve25519_init(&key);

ret = wc_curve25519_import_private_ex(priv, sizeof(priv), &key,
    EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}
```


B.19.2.8 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
)
```

この関数は、公開鍵-秘密鍵ペアを curve25519_key 構造体にインポートします。ビッグエンディアンのみ。

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **pub** インポートする公開鍵を含むバッファへのポインタ。
- **pubSz** インポートする公開鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_make_key](#)
- [wc_curve25519_import_public](#)
- [wc_curve25519_export_private_raw](#)

Return:

- 0 curve25519_key 構造体へのインポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが null の場合に返されます。
- ECC_BAD_ARG_E 入力鍵の鍵サイズが公開鍵または秘密鍵のサイズと一致しない場合に返されます。

Example

```
int ret;

byte priv[32];
byte pub[32];
// 公開鍵と秘密鍵で初期化
curve25519_key key;

wc_curve25519_init(&key);
// 鍵を初期化

ret = wc_curve25519_import_private_raw(&priv, sizeof(priv), pub,
                                       sizeof(pub), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.19.2.9 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
```

)

この関数は、公開鍵-秘密鍵ペアを `curve25519_key` 構造体にインポートします。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **pub** インポートする公開鍵を含むバッファへのポインタ。
- **pubSz** インポートする公開鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。
- **endian** 使用する形式を設定するための `EC25519_BIG_ENDIAN` または `EC25519_LITTLE_ENDIAN`。

See:

- `wc_curve25519_init`
- `wc_curve25519_make_key`
- `wc_curve25519_import_public`
- `wc_curve25519_export_private_raw`
- `wc_curve25519_import_private_raw`

Return:

- 0 `curve25519_key` 構造体へのインポートに成功した場合に返されます。
- `BAD_FUNC_ARG` 入力パラメータのいずれかが `null` の場合に返されます。
- `ECC_BAD_ARG_E` 入力鍵の鍵サイズが公開鍵または秘密鍵のサイズと一致しない場合に返されます。

Example

```
int ret;
byte priv[32];
byte pub[32];
// 公開鍵と秘密鍵で初期化
curve25519_key key;

wc_curve25519_init(&key);
// 鍵を初期化

ret = wc_curve25519_import_private_raw_ex(&priv, sizeof(priv), pub,
                                          sizeof(pub), &key, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.19.2.10 function `wc_curve25519_export_private_raw`

```
int wc_curve25519_export_private_raw(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、`curve25519_key` 構造体から秘密鍵をエクスポートし、与えられた `out` バッファに格納します。また、`outLen` をエクスポートされた鍵のサイズに設定します。ビッグエンディアンのみ。

Parameters:

- **key** 鍵をエクスポートする構造体へのポインタ。
- **out** エクスポートされた鍵を格納するバッファへのポインタ。

- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。

See:

- `wc_curve25519_init`
- `wc_curve25519_make_key`
- `wc_curve25519_import_private_raw`
- `wc_curve25519_export_private_raw_ex`

Return:

- 0 `curve25519_key` 構造体から秘密鍵のエクスポートに成功した場合に返されます。
- `BAD_FUNC_ARG` 入力パラメータのいずれかが `NULL` の場合に返されます。
- `ECC_BAD_ARG_E` `wc_curve25519_size()` が `key` と等しくない場合に返されます。

Example

```
int ret;
byte priv[32];
int privSz;

curve25519_key key;
// 鍵を初期化して作成

ret = wc_curve25519_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

B.19.2.11 function `wc_curve25519_export_private_raw_ex`

```
int wc_curve25519_export_private_raw_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

この関数は、`curve25519_key` 構造体から秘密鍵をエクスポートし、与えられた `out` バッファに格納します。また、`outLen` をエクスポートされた鍵のサイズに設定します。ビッグエンディアンまたはリトルエンディアンを指定できます。

Parameters:

- **key** 鍵をエクスポートする構造体へのポインタ。
- **out** エクスポートされた鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。
- **endian** 使用する形式を設定するための `EC25519_BIG_ENDIAN` または `EC25519_LITTLE_ENDIAN`。

See:

- `wc_curve25519_init`
- `wc_curve25519_make_key`
- `wc_curve25519_import_private_raw`
- `wc_curve25519_export_private_raw`
- `wc_curve25519_size`

Return:

- 0 curve25519_key 構造体から秘密鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E wc_curve25519_size() が key と等しくない場合に返されます。

Example

```
int ret;

byte priv[32];
int privSz;
curve25519_key key;
// 鍵を初期化して作成
ret = wc_curve25519_export_private_raw_ex(&key, priv, &privSz,
    EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

B.19.2.12 function wc_curve25519_import_public

```
int wc_curve25519_import_public(
    const byte * in,
    word32 inLen,
    curve25519_key * key
)
```

この関数は、与えられた in バッファから公開鍵をインポートし、curve25519_key 構造体に格納します。

Parameters:

- **in** インポートする公開鍵を含むバッファへのポインタ。
- **inLen** インポートする公開鍵の長さ。
- **key** 鍵を格納する curve25519_key 構造体へのポインタ。

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 curve25519_key 構造体への公開鍵のインポートに成功した場合に返されます。
- ECC_BAD_ARG_E inLen パラメータが鍵構造体の鍵サイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

byte pub[32];
// pub を公開鍵で初期化

curve25519_key key;
// 鍵を初期化

ret = wc_curve25519_import_public(pub, sizeof(pub), &key);
```

```

if (ret != 0) {
    // 鍵のインポートエラー
}

```

B.19.2.13 function wc_curve25519_import_public_ex

```

int wc_curve25519_import_public_ex(
    const byte * in,
    word32 inLen,
    curve25519_key * key,
    int endian
)

```

この関数は、与えられた in バッファから公開鍵をインポートし、curve25519_key 構造体に格納します。

Parameters:

- **in** インポートする公開鍵を含むバッファへのポインタ。
- **inLen** インポートする公開鍵の長さ。
- **key** 鍵を格納する curve25519_key 構造体へのポインタ。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

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 curve25519_key 構造体への公開鍵のインポートに成功した場合に返されます。
- ECC_BAD_ARG_E inLen パラメータが鍵構造体の鍵サイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```

int ret;

byte pub[32];
// pub を公開鍵で初期化
curve25519_key key;
// 鍵を初期化

ret = wc_curve25519_import_public_ex(pub, sizeof(pub), &key,
    EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}

```

B.19.2.14 function wc_curve25519_check_public

```

int wc_curve25519_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)

```

この関数は、エンディアン順序を考慮して、公開鍵バッファが有効な Curve25519 鍵値を保持しているかどうかをチェックします。

Parameters:

- **pub** チェックする公開鍵を含むバッファへのポインタ。
- **pubSz** チェックする公開鍵の長さ。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- `wc_curve25519_init`
- `wc_curve25519_import_public`
- `wc_curve25519_import_public_ex`
- `wc_curve25519_size`

Return:

- 0 公開鍵の値が有効な場合に返されます。
- ECC_BAD_ARG_E 公開鍵の値が有効でない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

byte pub[] = { // 公開鍵の内容 };

ret = wc_curve25519_check_public_ex(pub, sizeof(pub), EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.19.2.15 function wc_curve25519_export_public

```
int wc_curve25519_export_public(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。

Parameters:

- **key** 鍵をエクスポートする curve25519_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。

See:

- `wc_curve25519_init`
- `wc_curve25519_export_private_raw`
- `wc_curve25519_import_public`

Return:

- 0 curve25519_key 構造体から公開鍵のエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E outLen が CURVE25519_PUB_KEY_SIZE より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```

int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// 鍵を初期化して作成
ret = wc_curve25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}

```

B.19.2.16 function wc_curve25519_export_public_ex

```

int wc_curve25519_export_public_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)

```

この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **key** 鍵をエクスポートする curve25519_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_export_private_raw](#)
- [wc_curve25519_import_public](#)

Return:

- 0 curve25519_key 構造体から公開鍵のエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E outLen が CURVE25519_PUB_KEY_SIZE より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```

int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// 鍵を初期化して作成

ret = wc_curve25519_export_public_ex(&key, pub, &pubSz, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のエクスポートエラー
}

```

B.19.2.17 function wc_curve25519_export_key_raw

```
int wc_curve25519_export_key_raw(
    curve25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

Curve25519 鍵ペアをエクスポートします。ビッグエンディアンのみ。

Parameters:

- **key** 鍵ペアをエクスポートする curve448_key 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** 入力時は、priv バッファのバイト単位のサイズ。出力時は、priv バッファに書き込まれたバイト数を格納します。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** 入力時は、pub バッファのバイト単位のサイズ。出力時は、pub バッファに書き込まれたバイト数を格納します。

See:

- [wc_curve25519_export_key_raw_ex](#)
- [wc_curve25519_export_private_raw](#)

Return:

- 0 curve25519_key 構造体から鍵ペアのエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE25519_KEY_SIZE より小さい、または pubSz が CURVE25519_PUB_KEY_SIZE より小さい場合に返されます。

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
// 鍵を初期化して作成

ret = wc_curve25519_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

B.19.2.18 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
```


)

curve25519 鍵ペアをエクスポートします。ビッグエンディアンまたはリトルエンディアン。

Parameters:

- **key** 鍵ペアをエクスポートする curve448_key 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** 入力時は、priv バッファのバイト単位のサイズ。出力時は、priv バッファに書き込まれたバイト数を格納します。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** 入力時は、pub バッファのバイト単位のサイズ。出力時は、pub バッファに書き込まれたバイト数を格納します。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- [wc_curve25519_export_key_raw](#)
- [wc_curve25519_export_private_raw_ex](#)
- [wc_curve25519_export_public_ex](#)

Return:

- 0 curve25519_key 構造体から鍵ペアのエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE25519_KEY_SIZE より小さい、または pubSz が CURVE25519_PUB_KEY_SIZE より小さい場合に返されます。

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
// 鍵を初期化して作成

ret = wc_curve25519_export_key_raw_ex(&key, priv, &privSz, pub, &pubSz,
    EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

B.19.2.19 function wc_curve25519_size

```
int wc_curve25519_size(
    curve25519_key * key
)
```

この関数は、与えられた鍵構造体の鍵サイズを返します。

Parameters:

- **key** 鍵サイズを決定する curve25519_key 構造体へのポインタ。

See:

- [wc_curve25519_init](#)

- `wc_curve25519_make_key`

Return:

- Success 有効で初期化された `curve25519_key` 構造体を与えられた場合、鍵のサイズを返します。
- 0 `key` が NULL の場合に返されます。

Example

```
int keySz;  
  
curve25519_key key;  
// 鍵を初期化して作成  
  
keySz = wc_curve25519_size(&key);
```

B.20 アルゴリズム - Curve448

B.20.1 Functions

	Name
int	wc_curve448_make_key (WC_RNG * rng, int keysize, curve448_key * key) この関数は、与えられた乱数生成器 <code>rng</code> を使用して、与えられたサイズ (<code>keysize</code>) の Curve448 鍵を生成し、与えられた <code>curve448_key</code> 構造体に格納します。 <code>wc_curve448_init()</code> を通じて鍵構造体が初期化された後に呼び出す必要があります。
int	wc_curve448_shared_secret (curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen) この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ <code>out</code> に格納し、秘密鍵の変数を <code>outlen</code> に割り当てます。ビッグエンディアンのみをサポートします。
int	wc_curve448_shared_secret_ex (curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen, int endian) この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ <code>out</code> に格納し、秘密鍵の変数を <code>outlen</code> に割り当てます。ビッグエンディアンとリトルエンディアンの両方をサポートします。
int	wc_curve448_init (curve448_key * key) この関数は Curve448 鍵を初期化します。構造体の鍵を生成する前に呼び出す必要があります。
void	wc_curve448_free (curve448_key * key) この関数は Curve448 オブジェクトを解放します。
int	wc_curve448_import_private (const byte * priv, word32 privSz, curve448_key * key) この関数は <code>curve448</code> 秘密鍵のみをインポートします。(ビッグエンディアン)

	Name
int	wc_curve448_import_private_ex (const byte * priv, word32 privSz, curve448_key * key, int endian) curve448 秘密鍵のみのインポート。(ビッグエンディアンまたはリトルエンディアン)
int	wc_curve448_import_private_raw (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key) この関数は、公開鍵-秘密鍵ペアを curve448_key 構造体にインポートします。ビッグエンディアンのみ。
int	wc_curve448_import_private_raw_ex (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key, int endian) この関数は、公開鍵-秘密鍵ペアを curve448_key 構造体にインポートします。ビッグエンディアンとリトルエンディアンの両方をサポートします。
int	wc_curve448_export_private_raw (curve448_key * key, byte * out, word32 * outLen) この関数は、curve448_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンのみ。
int	wc_curve448_export_private_raw_ex (curve448_key * key, byte * out, word32 * outLen, int endian) この関数は、curve448_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンまたはリトルエンディアンを指定できます。
int	wc_curve448_import_public (const byte * in, word32 inLen, curve448_key * key) この関数は、与えられた in バッファから公開鍵をインポートし、curve448_key 構造体に格納します。
int	wc_curve448_import_public_ex (const byte * in, word32 inLen, curve448_key * key, int endian) この関数は、与えられた in バッファから公開鍵をインポートし、curve448_key 構造体に格納します。
int	wc_curve448_check_public (const byte * pub, word32 pubSz, int endian) この関数は、エンディアン順序を考慮して、公開鍵バッファが有効な Curve448 鍵値を保持しているかどうかをチェックします。
int	wc_curve448_export_public (curve448_key * key, byte * out, word32 * outLen) この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。

	Name
int	wc_curve448_export_public_ex (curve448_key * key, byte * out, word32 * outLen, int endian) この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンとリトルエンディアンの両方をサポートします。
int	wc_curve448_export_key_raw (curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、与えられた鍵構造体から鍵ペアをエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。
int	wc_curve448_export_key_raw_ex (curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian) curve448 鍵ペアをエクスポートします。ビッグエンディアンまたはリトルエンディアン。
int	wc_curve448_size (curve448_key * key) この関数は、与えられた鍵構造体の鍵サイズを返します。

B.20.2 Functions Documentation

B.20.2.1 function wc_curve448_make_key

```
int wc_curve448_make_key(
    WC_RNG * rng,
    int keysize,
    curve448_key * key
)
```

この関数は、与えられた乱数生成器 rng を使用して、与えられたサイズ (keysizes) の Curve448 鍵を生成し、与えられた curve448_key 構造体に格納します。wc_curve448_init() を通じて鍵構造体が初期化された後に呼び出す必要があります。

Parameters:

- **rng** ecc 鍵を生成するために使用される RNG オブジェクトへのポインタ。
- **keysizes** 生成する鍵のサイズ。curve448 では 56 バイトである必要があります。
- **key** 生成された鍵を格納する curve448_key 構造体へのポインタ。

See: [wc_curve448_init](#)

Return:

- 0 鍵の生成に成功し、与えられた curve448_key 構造体に格納された場合に返されます。
- ECC_BAD_ARG_E 入力 keysize が curve448 鍵の keysize に対応していない場合 (56 バイト) に返されます。
- RNG_FAILURE_E rng 内部ステータスが DRBG_OK でない場合、または rng で次のランダムブロックを生成する際にエラーがある場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

curve448_key key;
```

```

wc_curve448_init(&key); // 鍵を初期化
WC_RNG rng;
wc_InitRng(&rng); // 乱数生成器を初期化

ret = wc_curve448_make_key(&rng, 56, &key);
if (ret != 0) {
    // Curve448 鍵作成エラー
}

```

B.20.2.2 function wc_curve448_shared_secret

```

int wc_curve448_shared_secret(
    curve448_key * private_key,
    curve448_key * public_key,
    byte * out,
    word32 * outlen
)

```

この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンのみをサポートします。

Parameters:

- **private_key** ユーザーの秘密鍵で初期化された curve448_key 構造体へのポインタ。
- **public_key** 受信した公開鍵を含む curve448_key 構造体へのポインタ。
- **out** 56 バイトの計算された秘密鍵を格納するバッファへのポインタ。
- **outlen** 出力バッファに書き込まれた長さを格納するポインタ。

See:

- [wc_curve448_init](#)
- [wc_curve448_make_key](#)
- [wc_curve448_shared_secret_ex](#)

Return:

- 0 共有秘密鍵の計算に成功した場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

Example

```

int ret;

byte sharedKey[56];
word32 keySz;
curve448_key privKey, pubKey;
// 両方の鍵を初期化

ret = wc_curve448_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
if (ret != 0) {
    // 共有鍵生成エラー
}

```

B.20.2.3 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
)

```

この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ `out` に格納し、秘密鍵の変数を `outlen` に割り当てます。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **private_key** ユーザーの秘密鍵で初期化された `curve448_key` 構造体へのポインタ。
- **public_key** 受信した公開鍵を含む `curve448_key` 構造体へのポインタ。
- **out** 56 バイトの計算された秘密鍵を格納するバッファへのポインタ。
- **outlen** 出力バッファに書き込まれた長さを格納するポインタ。
- **endian** 使用する形式を設定するための `EC448_BIG_ENDIAN` または `EC448_LITTLE_ENDIAN`。

See:

- `wc_curve448_init`
- `wc_curve448_make_key`
- `wc_curve448_shared_secret`

Return:

- 0 共有秘密鍵の計算に成功した場合に返されます。
- `BAD_FUNC_ARG` 渡された入力パラメータのいずれかが `NULL` の場合に返されます。

Example

```

int ret;

byte sharedKey[56];
word32 keySz;

curve448_key privKey, pubKey;
// 両方の鍵を初期化

ret = wc_curve448_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
    EC448_BIG_ENDIAN);
if (ret != 0) {
    // 共有鍵生成エラー
}

```

B.20.2.4 function `wc_curve448_init`

```

int wc_curve448_init(
    curve448_key * key
)

```

この関数は Curve448 鍵を初期化します。構造体の鍵を生成する前に呼び出す必要があります。

Parameters:

- **key** 初期化する `curve448_key` 構造体へのポインタ。

See: `wc_curve448_make_key`

Return:

- 0 `curve448_key` 構造体の初期化に成功した場合に返されます。
- `BAD_FUNC_ARG` `key` が `NULL` の場合に返されます。

Example

```
curve448_key key;
wc_curve448_init(&key); // 鍵を初期化
// 鍵を作成し、暗号化に進む
```

B.20.2.5 function wc_curve448_free

```
void wc_curve448_free(
    curve448_key * key
)
```

この関数は Curve448 オブジェクトを解放します。

Parameters:

- **key** 解放する鍵オブジェクトへのポインタ。

See:

- [wc_curve448_init](#)
- [wc_curve448_make_key](#)

Example

```
curve448_key privKey;
// 鍵を初期化し、共有秘密鍵の生成に使用
wc_curve448_free(&privKey);
```

B.20.2.6 function wc_curve448_import_private

```
int wc_curve448_import_private(
    const byte * priv,
    word32 privSz,
    curve448_key * key
)
```

この関数は curve448 秘密鍵のみをインポートします。(ビッグエンディアン)

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。

See:

- [wc_curve448_import_private_ex](#)
- [wc_curve448_size](#)

Return:

- 0 秘密鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG key または priv が null の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE448_KEY_SIZE と等しくない場合に返されます。

Example

```
int ret;

byte priv[] = { // 秘密鍵の内容 };
curve448_key key;
wc_curve448_init(&key);
```

```
ret = wc_curve448_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.20.2.7 function wc_curve448_import_private_ex

```
int wc_curve448_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve448_key * key,
    int endian
)
```

curve448 秘密鍵のみのインポート。(ビッグエンディアンまたはリトルエンディアン)

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- [wc_curve448_import_private](#)
- [wc_curve448_size](#)

Return:

- 0 秘密鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG key または priv が null の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE448_KEY_SIZE と等しくない場合に返されます。

Example

```
int ret;

byte priv[] = { // 秘密鍵の内容 };
curve448_key key;
wc_curve448_init(&key);

ret = wc_curve448_import_private_ex(priv, sizeof(priv), &key,
    EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.20.2.8 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
)
```


この関数は、公開鍵-秘密鍵ペアを `curve448_key` 構造体にインポートします。ビッグエンディアンのみ。

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **pub** インポートする公開鍵を含むバッファへのポインタ。
- **pubSz** インポートする公開鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。

See:

- `wc_curve448_init`
- `wc_curve448_make_key`
- `wc_curve448_import_public`
- `wc_curve448_export_private_raw`

Return:

- 0 `curve448_key` 構造体へのインポートに成功した場合に返されます。
- `BAD_FUNC_ARG` 入力パラメータのいずれかが `null` の場合に返されます。
- `ECC_BAD_ARG_E` 入力鍵の鍵サイズが公開鍵または秘密鍵のサイズと一致しない場合に返されます。

Example

```
int ret;

byte priv[56];
byte pub[56];
// 公開鍵と秘密鍵で初期化
curve448_key key;

wc_curve448_init(&key);
// 鍵を初期化

ret = wc_curve448_import_private_raw(&priv, sizeof(priv), pub, sizeof(pub),
                                     &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.20.2.9 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
)
```

この関数は、公開鍵-秘密鍵ペアを `curve448_key` 構造体にインポートします。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **pub** インポートする公開鍵を含むバッファへのポインタ。

- **pubSz** インポートする公開鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- `wc_curve448_init`
- `wc_curve448_make_key`
- `wc_curve448_import_public`
- `wc_curve448_export_private_raw`
- `wc_curve448_import_private_raw`

Return:

- 0 `curve448_key` 構造体へのインポートに成功した場合に返されます。
- `BAD_FUNC_ARG` 入力パラメータのいずれかが `null` の場合に返されます。
- `ECC_BAD_ARG_E` 入力鍵の鍵サイズが公開鍵または秘密鍵のサイズと一致しない場合に返されます。

Example

```
int ret;

byte priv[56];
byte pub[56];
// 公開鍵と秘密鍵で初期化
curve448_key key;

wc_curve448_init(&key);
// 鍵を初期化

ret = wc_curve448_import_private_raw_ex(&priv, sizeof(priv), pub,
                                       sizeof(pub), &key, EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.20.2.10 function `wc_curve448_export_private_raw`

```
int wc_curve448_export_private_raw(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、`curve448_key` 構造体から秘密鍵をエクスポートし、与えられた `out` バッファに格納します。また、`outLen` をエクスポートされた鍵のサイズに設定します。ビッグエンディアンのみ。

Parameters:

- **key** 鍵をエクスポートする構造体へのポインタ。
- **out** エクスポートされた鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、`out` のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。

See:

- `wc_curve448_init`
- `wc_curve448_make_key`
- `wc_curve448_import_private_raw`
- `wc_curve448_export_private_raw_ex`

Return:

- 0 curve448_key 構造体から秘密鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E wc_curve448_size() が key と等しくない場合に返されます。

Example

```
int ret;
byte priv[56];
int privSz;

curve448_key key;
// 鍵を初期化して作成

ret = wc_curve448_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

B.20.2.11 function wc_curve448_export_private_raw_ex

```
int wc_curve448_export_private_raw_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

この関数は、curve448_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンまたはリトルエンディアンを指定できます。

Parameters:

- **key** 鍵をエクスポートする構造体へのポインタ。
- **out** エクスポートされた鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw
- wc_curve448_size

Return:

- 0 curve448_key 構造体から秘密鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E wc_curve448_size() が key と等しくない場合に返されます。

Example

```
int ret;

byte priv[56];
```

```

int privSz;
curve448_key key;
// 鍵を初期化して作成
ret = wc_curve448_export_private_raw_ex(&key, priv, &privSz,
    EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のエクスポートエラー
}

```

B.20.2.12 function wc_curve448_import_public

```

int wc_curve448_import_public(
    const byte * in,
    word32 inLen,
    curve448_key * key
)

```

この関数は、与えられた in バッファから公開鍵をインポートし、curve448_key 構造体に格納します。

Parameters:

- **in** インポートする公開鍵を含むバッファへのポインタ。
- **inLen** インポートする公開鍵の長さ。
- **key** 鍵を格納する curve448_key 構造体へのポインタ。

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 curve448_key 構造体への公開鍵のインポートに成功した場合に返されます。
- ECC_BAD_ARG_E inLen パラメータが鍵構造体の鍵サイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```

int ret;

byte pub[56];
// pub を公開鍵で初期化

curve448_key key;
// 鍵を初期化

ret = wc_curve448_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}

```

B.20.2.13 function wc_curve448_import_public_ex

```

int wc_curve448_import_public_ex(
    const byte * in,

```

```

    word32 inLen,
    curve448_key * key,
    int endian
)

```

この関数は、与えられた in バッファから公開鍵をインポートし、curve448_key 構造体に格納します。

Parameters:

- **in** インポートする公開鍵を含むバッファへのポインタ。
- **inLen** インポートする公開鍵の長さ。
- **key** 鍵を格納する curve448_key 構造体へのポインタ。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

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 curve448_key 構造体への公開鍵のインポートに成功した場合に返されます。
- ECC_BAD_ARG_E inLen パラメータが鍵構造体の鍵サイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```

int ret;

byte pub[56];
// pub を公開鍵で初期化
curve448_key key;
// 鍵を初期化

ret = wc_curve448_import_public_ex(pub, sizeof(pub), &key,
    EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}

```

B.20.2.14 function wc_curve448_check_public

```

int wc_curve448_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)

```

この関数は、エンディアン順序を考慮して、公開鍵バッファが有効な Curve448 鍵値を保持しているかどうかをチェックします。

Parameters:

- **pub** チェックする公開鍵を含むバッファへのポインタ。
- **pubSz** チェックする公開鍵の長さ。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- [wc_curve448_init](#)
- [wc_curve448_import_public](#)
- [wc_curve448_import_public_ex](#)
- [wc_curve448_size](#)

Return:

- 0 公開鍵の値が有効な場合に返されます。
- ECC_BAD_ARG_E 公開鍵の値が有効でない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

byte pub[] = { // 公開鍵の内容 };

ret = wc_curve448_check_public_ex(pub, sizeof(pub), EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.20.2.15 function wc_curve448_export_public

```
int wc_curve448_export_public(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。

Parameters:

- **key** 鍵をエクスポートする curve448_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。

See:

- [wc_curve448_init](#)
- [wc_curve448_export_private_raw](#)
- [wc_curve448_import_public](#)

Return:

- 0 curve448_key 構造体から公開鍵のエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E outLen が CURVE448_PUB_KEY_SIZE より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

byte pub[56];
int pubSz;
```

```

curve448_key key;
// 鍵を初期化して作成

ret = wc_curve448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}

```

B.20.2.16 function wc_curve448_export_public_ex

```

int wc_curve448_export_public_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)

```

この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **key** 鍵をエクスポートする curve448_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- [wc_curve448_init](#)
- [wc_curve448_export_private_raw](#)
- [wc_curve448_import_public](#)

Return:

- 0 curve448_key 構造体から公開鍵のエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E outLen が CURVE448_PUB_KEY_SIZE より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```

int ret;

byte pub[56];
int pubSz;

curve448_key key;
// 鍵を初期化して作成

ret = wc_curve448_export_public_ex(&key, pub, &pubSz, EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のエクスポートエラー
}

```

B.20.2.17 function wc_curve448_export_key_raw

```

int wc_curve448_export_key_raw(
    curve448_key * key,

```

```

    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)

```

この関数は、与えられた鍵構造体から鍵ペアをエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。

Parameters:

- **key** 鍵ペアをエクスポートする curve448_key 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** 入力時は、priv バッファのバイト単位のサイズ。出力時は、priv バッファに書き込まれたバイト数を格納します。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** 入力時は、pub バッファのバイト単位のサイズ。出力時は、pub バッファに書き込まれたバイト数を格納します。

See:

- [wc_curve448_export_key_raw_ex](#)
- [wc_curve448_export_private_raw](#)

Return:

- 0 curve448_key 構造体から鍵ペアのエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE448_KEY_SIZE より小さい、または pubSz が CURVE448_PUB_KEY_SIZE より小さい場合に返されます。

Example

```

int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

curve448_key key;
// 鍵を初期化して作成

ret = wc_curve448_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}

```

B.20.2.18 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
)

```


curve448 鍵ペアをエクスポートします。ビッグエンディアンまたはリトルエンディアン。

Parameters:

- **key** 鍵ペアをエクスポートする curve448_key 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** 入力時は、priv バッファのバイト単位のサイズ。出力時は、priv バッファに書き込まれたバイト数を格納します。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** 入力時は、pub バッファのバイト単位のサイズ。出力時は、pub バッファに書き込まれたバイト数を格納します。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- [wc_curve448_export_key_raw](#)
- [wc_curve448_export_private_raw_ex](#)
- [wc_curve448_export_public_ex](#)

Return:

- 0 成功。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE448_KEY_SIZE より小さい、または pubSz が CURVE448_PUB_KEY_SIZE より小さい場合に返されます。

この関数は、与えられた鍵構造体から鍵ペアをエクスポートし、結果を out バッファに格納します。ビッグエンディアンまたはリトルエンディアン。

Example

```
int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

curve448_key key;
// 鍵を初期化して作成

ret = wc_curve448_export_key_raw_ex(&key, priv, &privSz, pub, &pubSz,
    EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

B.20.2.19 function wc_curve448_size

```
int wc_curve448_size(
    curve448_key * key
)
```

この関数は、与えられた鍵構造体の鍵サイズを返します。

Parameters:

- **key** 鍵サイズを決定する curve448_key 構造体へのポインタ。

See:

- `wc_curve448_init`
- `wc_curve448_make_key`

Return:

- Success 有効で初期化された `curve448_key` 構造体を与えられた場合、鍵のサイズを返します。
- 0 `key` が NULL の場合に返されます。

Example

```
int keySz;

curve448_key key;
// 鍵を初期化して作成

keySz = wc_curve448_size(&key);
```

B.21 アルゴリズム - DSA**B.21.1 Functions**

	Name
int	wc_InitDsaKey (DsaKey * key) この関数は、デジタル署名アルゴリズム (DSA) による認証に使用するために、DsaKey オブジェクトを初期化します。
void	wc_FreeDsaKey (DsaKey * key) この関数は、使用後に DsaKey オブジェクトを解放します。
int	wc_DsaSign (const byte * digest, byte * out, DsaKey * key, WC_RNG * rng) この関数は、入力ダイジェストに署名し、結果を出力バッファ <code>out</code> に格納します。
int	wc_DsaVerify (const byte * digest, const byte * sig, DsaKey * key, int * answer) この関数は、秘密鍵を使用してダイジェストの署名を検証します。検証が正しく行われたかどうかを <code>answer</code> パラメータに格納します。1 は検証成功、0 は検証失敗に対応します。
int	wc_DsaPublicKeyDecode (const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz) この関数は、DSA 公開鍵を含む DER フォーマットの証明書バッファをデコードし、与えられた DsaKey 構造体に鍵を格納します。また、読み取られた入力の長さに応じて <code>inOutIdx</code> パラメータを設定します。
int	wc_DsaPrivateKeyDecode (const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz) この関数は、DSA 秘密鍵を含む DER フォーマットの証明書バッファをデコードし、与えられた DsaKey 構造体に鍵を格納します。また、読み取られた入力の長さに応じて <code>inOutIdx</code> パラメータを設定します。
int	wc_DsaKeyToDer (DsaKey * key, byte * output, word32 inLen) DsaKey 鍵を DER フォーマットに変換し、 <code>output(inLen)</code> に書き込み、書き込まれたバイト数を返します。

	Name
int	wc_MakeDsaKey (WC_RNG * rng, DsaKey * dsa) DSA 鍵を作成します。
int	wc_MakeDsaParameters (WC_RNG * rng, int modulus_size, DsaKey * dsa) FIPS 186_4 は、modulus_size 値として (1024, 160) (2048, 256) (3072, 256) を有効と定義しています

B.21.2 Functions Documentation

B.21.2.1 function wc_InitDsaKey

```
int wc_InitDsaKey(
    DsaKey * key
)
```

この関数は、デジタル署名アルゴリズム (DSA) による認証に使用するために、DsaKey オブジェクトを初期化します。

Parameters:

- **key** 初期化する DsaKey 構造体へのポインタ

See: [wc_FreeDsaKey](#)

Return:

- 0 成功時に返されます。
- BAD_FUNC_ARG NULL キーが渡された場合に返されます。

Example

```
DsaKey key;
int ret;
ret = wc_InitDsaKey(&key); // DSA 鍵を初期化
```

B.21.2.2 function wc_FreeDsaKey

```
void wc_FreeDsaKey(
    DsaKey * key
)
```

この関数は、使用後に DsaKey オブジェクトを解放します。

Parameters:

- **key** 解放する DsaKey 構造体へのポインタ

See: [wc_FreeDsaKey](#)

Return: none 戻り値なし。

Example

```
DsaKey key;
// 鍵を初期化、認証に使用
...
wc_FreeDsaKey(&key); // DSA 鍵を解放
```

B.21.2.3 function wc_DsaSign

```
int wc_DsaSign(
    const byte * digest,
    byte * out,
    DsaKey * key,
    WC_RNG * rng
)
```

この関数は、入力ダイジェストに署名し、結果を出力バッファ out に格納します。

Parameters:

- **digest** 署名するハッシュへのポインタ
- **out** 署名を格納するバッファへのポインタ
- **key** 署名を生成するために使用する初期化された DsaKey 構造体へのポインタ
- **rng** 署名生成で使用する初期化された RNG へのポインタ

See: [wc_DsaVerify](#)

Return:

- 0 入力ダイジェストへの署名に成功した場合に返されます
- MP_INIT_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_READ_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_MUL_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_TO_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_MEM DSA 署名の処理中にエラーが発生した場合に返される可能性があります。

Example

```
DsaKey key;
// DSA 鍵を初期化、秘密鍵をロード
int ret;
WC_RNG rng;
wc_InitRng(&rng);
byte hash[] = { // ハッシュダイジェストで初期化 };
byte signature[40]; // 署名は 40 バイト (320 ビット) になります

ret = wc_DsaSign(hash, signature, &key, &rng);
if (ret != 0) {
    // DSA 署名生成エラー
}
```

B.21.2.4 function wc_DsaVerify

```
int wc_DsaVerify(
    const byte * digest,
    const byte * sig,
    DsaKey * key,
    int * answer
)
```

この関数は、秘密鍵を使用してダイジェストの署名を検証します。検証が正しく行われたかどうかを `answer` パラメータに格納します。1 は検証成功、0 は検証失敗に対応します。

Parameters:

- **digest** 署名の対象を含むダイジェストへのポインタ
- **sig** 検証する署名を含むバッファへのポインタ
- **key** 署名を検証するために使用する初期化された `DsaKey` 構造体へのポインタ
- **answer** 検証が成功したかどうかを格納する整数へのポインタ

See: [wc_DsaSign](#)

Return:

- 0 検証リクエストの処理に成功した場合に返されます。注意: これは署名が検証されたことを意味するのではなく、関数が成功したことを意味します
- `MP_INIT_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_READ_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_CMP_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_INVMOD_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_EXPTMOD_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_MOD_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_MUL_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_ADD_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_MULMOD_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_TO_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_MEM` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。

Example

```
DsaKey key;
// DSA 鍵を初期化、公開鍵をロード

int ret;
int verified;
byte hash[] = { // ハッシュダイジェストで初期化 };
byte signature[] = { // 検証する署名で初期化 };
ret = wc_DsaVerify(hash, signature, &key, &verified);
if (ret != 0) {
    // 検証リクエストの処理エラー
} else if (answer == 0) {
    // 無効な署名
}
```

B.21.2.5 function `wc_DsaPublicKeyDecode`

```
int wc_DsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

この関数は、DSA 公開鍵を含む DER フォーマットの証明書バッファをデコードし、与えられた `DsaKey` 構造体に鍵を格納します。また、読み取られた入力長さに応じて `inOutIdx` パラメータを設定します。

Parameters:

- **input** DER フォーマットの DSA 公開鍵を含むバッファへのポインタ

- **inOutIdx** 読み取られた証明書の最終インデックスを格納する整数へのポインタ
- **key** 公開鍵を格納する DsaKey 構造体へのポインタ
- **inSz** 入力バッファのサイズ

See:

- [wc_InitDsaKey](#)
- [wc_DsaPrivateKeyDecode](#)

Return:

- 0 DsaKey オブジェクトの公開鍵の設定に成功した場合に返されます
- ASN_PARSE_E 証明書バッファの読み取り中にエンコーディングエラーがある場合に返されます
- ASN_DH_KEY_E DSA パラメータの 1 つが正しくフォーマットされていない場合に返されます

Example

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);
byte derBuff[] = { // DSA 公開鍵 };
ret = wc_DsaPublicKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // 公開鍵の読み取りエラー
}
```

B.21.2.6 function wc_DsaPrivateKeyDecode

```
int wc_DsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

この関数は、DSA 秘密鍵を含む DER フォーマットの証明書バッファをデコードし、与えられた DsaKey 構造体に鍵を格納します。また、読み取られた入力長さに応じて inOutIdx パラメータを設定します。

Parameters:

- **input** DER フォーマットの DSA 秘密鍵を含むバッファへのポインタ
- **inOutIdx** 読み取られた証明書の最終インデックスを格納する整数へのポインタ
- **key** 秘密鍵を格納する DsaKey 構造体へのポインタ
- **inSz** 入力バッファのサイズ

See:

- [wc_InitDsaKey](#)
- [wc_DsaPublicKeyDecode](#)

Return:

- 0 DsaKey オブジェクトの秘密鍵の設定に成功した場合に返されます
- ASN_PARSE_E 証明書バッファの読み取り中にエンコーディングエラーがある場合に返されます
- ASN_DH_KEY_E DSA パラメータの 1 つが正しくフォーマットされていない場合に返されます

Example

```
int ret, idx=0;

DsaKey key;
```

```

wc_InitDsaKey(&key);
byte derBuff[] = { // DSA 秘密鍵 };
ret = wc_DsaPrivateKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // 秘密鍵の読み取りエラー
}

```

B.21.2.7 function wc_DsaKeyToDer

```

int wc_DsaKeyToDer(
    DsaKey * key,
    byte * output,
    word32 inLen
)

```

DsaKey 鍵を DER フォーマットに変換し、output(inLen) に書き込み、書き込まれたバイト数を返します。

Parameters:

- **key** 変換する DsaKey 構造体へのポインタ。
- **output** 変換された鍵用の出力バッファへのポインタ。
- **inLen** 鍵入力の長さ。

See:

- [wc_InitDsaKey](#)
- [wc_FreeDsaKey](#)
- [wc_MakeDsaKey](#)

Return:

- outLen 成功、書き込まれたバイト数
- BAD_FUNC_ARG key または output が null または key->type が DSA_PRIVATE でない場合。
- MEMORY_E メモリ割り当てエラー。

Example

```

DsaKey key;
WC_RNG rng;
int derSz;
int bufferSize = // 十分なバッファサイズ;
byte der[bufferSize];

wc_InitDsaKey(&key);
wc_InitRng(&rng);
wc_MakeDsaKey(&rng, &key);
derSz = wc_DsaKeyToDer(&key, der, bufferSize);

```

B.21.2.8 function wc_MakeDsaKey

```

int wc_MakeDsaKey(
    WC_RNG * rng,
    DsaKey * dsa
)

```

DSA 鍵を作成します。

Parameters:

- **rng** WC_RNG 構造体へのポインタ。

- **dsa** DsaKey 構造体へのポインタ。

See:

- [wc_InitDsaKey](#)
- [wc_FreeDsaKey](#)
- [wc_DsaSign](#)

Return:

- MP_OKAY 成功
- BAD_FUNC_ARG rng または dsa が null の場合。
- MEMORY_E バッファ用のメモリを割り当てられませんでした。
- MP_INIT_E mp_int の初期化エラー

Example

```
WC_RNG rng;
DsaKey dsa;
wc_InitRng(&rng);
wc_InitDsa(&dsa);
if(wc_MakeDsaKey(&rng, &dsa) != 0)
{
    // 鍵作成エラー
}
```

B.21.2.9 function wc_MakeDsaParameters

```
int wc_MakeDsaParameters(
    WC_RNG * rng,
    int modulus_size,
    DsaKey * dsa
)
```

FIPS 186-4 は、modulus_size 値として (1024, 160) (2048, 256) (3072, 256) を有効と定義しています

Parameters:

- **rng** wolfCrypt rng へのポインタ。
- **modulus_size** 1024、2048、または 3072 が有効な値です。
- **dsa** DsaKey 構造体へのポインタ。

See:

- [wc_MakeDsaKey](#)
- [wc_DsaKeyToDer](#)
- [wc_InitDsaKey](#)

Return:

- 0 成功
- BAD_FUNC_ARG rng または dsa が null または modulus_size が無効な場合。
- MEMORY_E メモリ割り当て試行エラー。

Example

```
DsaKey key;
WC_RNG rng;
wc_InitDsaKey(&key);
wc_InitRng(&rng);
if(wc_MakeDsaParameters(&rng, 1024, &genKey) != 0)
{
```



```
// エラーを処理
}
```

B.22 アルゴリズム - Diffie-Hellman

B.22.1 Functions

	Name
int	wc_InitDhKey (DhKey * key) この関数は、Diffie-Hellman 鍵交換プロトコルで安全な秘密鍵をネゴシエートするために使用する Diffie-Hellman 鍵を初期化します。
int	wc_FreeDhKey (DhKey * key) この関数は、Diffie-Hellman 鍵交換プロトコルで安全な秘密鍵をネゴシエートするために使用された後、Diffie-Hellman 鍵を解放します。
int	wc_DhGenerateKeyPair (DhKey * key, WC_RNG * rng, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、Diffie-Hellman 公開パラメータに基づいて公開/秘密鍵ペアを生成し、秘密鍵を priv に、公開鍵を pub に格納します。初期化された Diffie-Hellman 鍵と初期化された rng 構造体を受け取ります。
int	wc_DhAgree (DhKey * key, byte * agree, word32 * agreeSz, const byte * priv, word32 privSz, const byte * otherPub, word32 pubSz) この関数は、ローカル秘密鍵と受信した公開鍵に基づいて合意された秘密鍵を生成します。交換の両側で完了すると、この関数は対称通信用の合意された秘密鍵を生成します。共有秘密鍵の生成に成功すると、書き込まれた秘密鍵のサイズが agreeSz に格納されます。
int	wc_DhKeyDecode (const byte * input, word32 * inOutIdx, DhKey * key, word32 inSz) この関数は、DER フォーマットの鍵を含む指定された入力バッファから Diffie-Hellman 鍵をデコードします。結果を DhKey 構造体に格納します。
int	wc_DhSetKey (DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz) この関数は、入力秘密鍵パラメータを使用して DhKey 構造体の鍵を設定します。wc_DhKeyDecode とは異なり、この関数は入力鍵が DER フォーマットでフォーマットされている必要はなく、代わりに単純に解析された入力パラメータ p(素数) と g(ベース)を受け入れます。
int	wc_DhParamsLoad (const byte * input, word32 inSz, byte * p, word32 * pInOutSz, byte * g, word32 * gInOutSz) この関数は、指定された入力バッファから DER フォーマットの Diffie-Hellman パラメータ p(素数) と g(ベース)をロードします。

	Name
const DhParams *	wc_Dh_ffdhe2048_Get (void) この関数は...を返し、HAVE_FFDHE_2048 が定義されている必要があります。
const DhParams *	wc_Dh_ffdhe3072_Get (void) この関数は...を返し、HAVE_FFDHE_3072 が定義されている必要があります。
const DhParams *	wc_Dh_ffdhe4096_Get (void) この関数は...を返し、HAVE_FFDHE_4096 が定義されている必要があります。
const DhParams *	wc_Dh_ffdhe6144_Get (void) この関数は...を返し、HAVE_FFDHE_6144 が定義されている必要があります。
const DhParams *	wc_Dh_ffdhe8192_Get (void) この関数は...を返し、HAVE_FFDHE_8192 が定義されている必要があります。
int	wc_DhCheckKeyPair (DhKey * key, const byte * pub, word32 pubSz, const byte * priv, word32 privSz) FFC の SP 800_56Ar3、セクション 5.6.2.1.4、メソッド (b) のプロセスに従って、DH 鍵のペアワイズ整合性をチェックします。
int	wc_DhCheckPrivKey (DhKey * key, const byte * priv, word32 pubSz) 無効な数値について DH 秘密鍵をチェックします
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.22.2 Functions Documentation

B.22.2.1 function wc_InitDhKey

```
int wc_InitDhKey(
    DhKey * key
)
```

この関数は、Diffie-Hellman 鍵交換プロトコルで安全な秘密鍵をネゴシエートするために使用する Diffie-

Hellman 鍵を初期化します。

Parameters:

- **key** 安全な鍵交換で使用するために初期化する DhKey 構造体へのポインタ

See:

- `wc_FreeDhKey`
- `wc_DhGenerateKeyPair`

Return: none 戻り値なし。

Example

```
DhKey key;  
wc_InitDhKey(&key); // DH 鍵を初期化
```

B.22.2.2 function wc_FreeDhKey

```
int wc_FreeDhKey(  
    DhKey * key  
)
```

この関数は、Diffie-Hellman 鍵交換プロトコルで安全な秘密鍵をネゴシエートするために使用された後、Diffie-Hellman 鍵を解放します。

Parameters:

- **key** 解放する DhKey 構造体へのポインタ

See: `wc_InitDhKey`

Return: none 戻り値なし。

Example

```
DhKey key;  
// 鍵を初期化、鍵交換を実行
```

```
wc_FreeDhKey(&key); // メモリリークを避けるために DH 鍵を解放
```

B.22.2.3 function wc_DhGenerateKeyPair

```
int wc_DhGenerateKeyPair(  
    DhKey * key,  
    WC_RNG * rng,  
    byte * priv,  
    word32 * privSz,  
    byte * pub,  
    word32 * pubSz  
)
```

この関数は、Diffie-Hellman 公開パラメータに基づいて公開/秘密鍵ペアを生成し、秘密鍵を `priv` に、公開鍵を `pub` に格納します。初期化された Diffie-Hellman 鍵と初期化された `rng` 構造体を受け取ります。

Parameters:

- **key** 鍵ペアを生成する DhKey 構造体へのポインタ
- **rng** 鍵を生成するために使用する初期化された乱数生成器 (`rng`) へのポインタ
- **priv** 秘密鍵を格納するバッファへのポインタ
- **privSz** `priv` に書き込まれた秘密鍵のサイズを格納します
- **pub** 公開鍵を格納するバッファへのポインタ

- **pubSz** pub に書き込まれた秘密鍵のサイズを格納します

See:

- [wc_InitDhKey](#)
- [wc_DhSetKey](#)
- [wc_DhKeyDecode](#)

Return:

- **BAD_FUNC_ARG** この関数への入力の 1 つを解析する際にエラーがある場合に返されます
- **RNG_FAILURE_E** rng を使用して乱数を生成する際にエラーがある場合に返されます
- **MP_INIT_E** 公開鍵を生成する際に数学ライブラリでエラーがある場合に返される可能性があります
- **MP_READ_E** 公開鍵を生成する際に数学ライブラリでエラーがある場合に返される可能性があります
- **MP_EXPTMOD_E** 公開鍵を生成する際に数学ライブラリでエラーがある場合に返される可能性があります
- **MP_TO_E** 公開鍵を生成する際に数学ライブラリでエラーがある場合に返される可能性があります

Example

```
DhKey key;
int ret;
byte priv[256];
byte pub[256];
word32 privSz, pubSz;

wc_InitDhKey(&key); // 鍵を初期化
// wc_DhSetKey または wc_DhKeyDecode を使用して DH パラメータを設定
WC_RNG rng;
wc_InitRng(&rng); // rng を初期化
ret = wc_DhGenerateKeyPair(&key, &rng, priv, &privSz, pub, &pubSz);
```

B.22.2.4 function wc_DhAgree

```
int wc_DhAgree(
    DhKey * key,
    byte * agree,
    word32 * agreeSz,
    const byte * priv,
    word32 privSz,
    const byte * otherPub,
    word32 pubSz
)
```

この関数は、ローカル秘密鍵と受信した公開鍵に基づいて合意された秘密鍵を生成します。交換の両側で完了すると、この関数は対称通信用の合意された秘密鍵を生成します。共有秘密鍵の生成に成功すると、書き込まれた秘密鍵のサイズが agreeSz に格納されます。

Parameters:

- **key** 共有鍵を計算するために使用する DhKey 構造体へのポインタ
- **agree** 秘密鍵を格納するバッファへのポインタ
- **agreeSz** 生成成功後に秘密鍵のサイズを保持します
- **priv** ローカル秘密鍵を含むバッファへのポインタ
- **privSz** ローカル秘密鍵のサイズ
- **otherPub** 受信した公開鍵を含むバッファへのポインタ
- **pubSz** 受信した公開鍵のサイズ

See: [wc_DhGenerateKeyPair](#)

Return:

- 0 合意された秘密鍵の生成に成功した場合に返されます
- MP_INIT_E 共有秘密鍵の生成中にエラーがある場合に返される可能性があります
- MP_READ_E 共有秘密鍵の生成中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 共有秘密鍵の生成中にエラーがある場合に返される可能性があります
- MP_TO_E 共有秘密鍵の生成中にエラーがある場合に返される可能性があります

Example

```
DhKey key;
int ret;
byte priv[256];
byte agree[256];
word32 agreeSz;

// 鍵を初期化、鍵素数とベースを設定
// wc_DhGenerateKeyPair -- 秘密鍵を priv に格納
byte pub[] = { // 受信した公開鍵で初期化 };
ret = wc_DhAgree(&key, agree, &agreeSz, priv, sizeof(priv), pub,
sizeof(pub));
if ( ret != 0 ) {
    // 共有鍵生成エラー
}
```

B.22.2.5 function wc_DhKeyDecode

```
int wc_DhKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DhKey * key,
    word32 inSz
)
```

この関数は、DER フォーマットの鍵を含む指定された入力バッファから Diffie-Hellman 鍵をデコードします。結果を DhKey 構造体に格納します。

Parameters:

- **input** DER フォーマットの Diffie-Hellman 鍵を含むバッファへのポインタ
- **inOutIdx** 鍵をデコードする際に解析されたインデックスを格納する整数へのポインタ
- **key** 入力鍵で初期化する DhKey 構造体へのポインタ
- **inSz** 入力バッファの長さ。読み取り可能な最大長を示します

See: [wc_DhSetKey](#)

Return:

- 0 入力鍵のデコードに成功した場合に返されます
- ASN_PARSE_E 入力のシーケンスを解析する際にエラーがある場合に返されます
- ASN_DH_KEY_E 解析された入力から秘密鍵パラメータを読み取る際にエラーがある場合に返されます

Example

```
DhKey key;
word32 idx = 0;

byte keyBuff[1024];
// DER フォーマットの鍵で初期化
```

```

wc_DhKeyInit(&key);
ret = wc_DhKeyDecode(keyBuff, &idx, &key, sizeof(keyBuff));

if ( ret != 0 ) {
    // 鍵のデコードエラー
}

```

B.22.2.6 function wc_DhSetKey

```

int wc_DhSetKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz
)

```

この関数は、入力秘密鍵パラメータを使用して DhKey 構造体の鍵を設定します。wc_DhKeyDecode とは異なり、この関数は入力鍵が DER フォーマットでフォーマットされている必要はなく、代わりに単純に解析された入力パラメータ p(素数) と g(ベース) を受け入れます。

Parameters:

- **key** 鍵を設定する DhKey 構造体へのポインタ
- **p** 鍵で使用する素数を含むバッファへのポインタ
- **pSz** 入力素数の長さ
- **g** 鍵で使用するベースを含むバッファへのポインタ
- **gSz** 入力ベースの長さ

See: [wc_DhKeyDecode](#)

Return:

- 0 鍵の設定に成功した場合に返されます
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価される場合に返されます
- MP_INIT_E 格納用の鍵パラメータを初期化する際にエラーがある場合に返されます
- ASN_DH_KEY_E DH 鍵パラメータ p と g を読み取る際にエラーがある場合に返されます

Example

```

DhKey key;

byte p[] = { // 素数で初期化 };
byte g[] = { // ベースで初期化 };
wc_DhKeyInit(&key);
ret = wc_DhSetKey(key, p, sizeof(p), g, sizeof(g));

if ( ret != 0 ) {
    // 鍵設定エラー
}

```

B.22.2.7 function wc_DhParamsLoad

```

int wc_DhParamsLoad(
    const byte * input,
    word32 inSz,
    byte * p,
    word32 * pInOutSz,
    byte * g,

```

```

    word32 * gInOutSz
)

```

この関数は、指定された入力バッファから DER フォーマットの Diffie-Hellman パラメータ p(素数) と g(ベース) をロードします。

Parameters:

- **input** 解析する DER フォーマットの Diffie-Hellman 証明書を含むバッファへのポインタ
- **inSz** 入力バッファのサイズ
- **p** 解析された素数を格納するバッファへのポインタ
- **pInOutSz** p バッファで利用可能なサイズを含む word32 オブジェクトへのポインタ。関数呼び出しの完了後、バッファに書き込まれたバイト数で上書きされます
- **g** 解析されたベースを格納するバッファへのポインタ
- **gInOutSz** g バッファで利用可能なサイズを含む word32 オブジェクトへのポインタ。関数呼び出しの完了後、バッファに書き込まれたバイト数で上書きされます

See:

- [wc_DhSetKey](#)
- [wc_DhKeyDecode](#)

Return:

- 0 DH パラメータの抽出に成功した場合に返されます
- ASN_PARSE_E DER フォーマットの DH 証明書を解析する際にエラーが発生した場合に返されます
- BUFFER_E p または g に解析されたパラメータを格納する十分なスペースがない場合に返されます

Example

```

byte dhCert[] = { DER フォーマットの証明書で初期化 };
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 ) {
    // 入力の解析エラー
}

```

B.22.2.8 function wc_Dh_ffdhe2048_Get

```

const DhParams * wc_Dh_ffdhe2048_Get(
    void
)

```

この関数は...を返し、HAVE_FFDHE_2048 が定義されている必要があります。

See:

- [wc_Dh_ffdhe3072_Get](#)
- [wc_Dh_ffdhe4096_Get](#)
- [wc_Dh_ffdhe6144_Get](#)
- [wc_Dh_ffdhe8192_Get](#)

B.22.2.9 function wc_Dh_ffdhe3072_Get

```

const DhParams * wc_Dh_ffdhe3072_Get(
    void
)

```

この関数は...を返し、HAVE_FFDHE_3072 が定義されている必要があります。

See:

- [wc_Dh_ffdhe2048_Get](#)
- [wc_Dh_ffdhe4096_Get](#)
- [wc_Dh_ffdhe6144_Get](#)
- [wc_Dh_ffdhe8192_Get](#)

B.22.2.10 function wc_Dh_ffdhe4096_Get

```
const DhParams * wc_Dh_ffdhe4096_Get(  
    void  
)
```

この関数は...を返し、HAVE_FFDHE_4096 が定義されている必要があります。

See:

- [wc_Dh_ffdhe2048_Get](#)
- [wc_Dh_ffdhe3072_Get](#)
- [wc_Dh_ffdhe6144_Get](#)
- [wc_Dh_ffdhe8192_Get](#)

B.22.2.11 function wc_Dh_ffdhe6144_Get

```
const DhParams * wc_Dh_ffdhe6144_Get(  
    void  
)
```

この関数は...を返し、HAVE_FFDHE_6144 が定義されている必要があります。

See:

- [wc_Dh_ffdhe2048_Get](#)
- [wc_Dh_ffdhe3072_Get](#)
- [wc_Dh_ffdhe4096_Get](#)
- [wc_Dh_ffdhe8192_Get](#)

B.22.2.12 function wc_Dh_ffdhe8192_Get

```
const DhParams * wc_Dh_ffdhe8192_Get(  
    void  
)
```

この関数は...を返し、HAVE_FFDHE_8192 が定義されている必要があります。

See:

- [wc_Dh_ffdhe2048_Get](#)
- [wc_Dh_ffdhe3072_Get](#)
- [wc_Dh_ffdhe4096_Get](#)
- [wc_Dh_ffdhe6144_Get](#)

B.22.2.13 function wc_DhCheckKeyPair

```
int wc_DhCheckKeyPair(  
    DhKey * key,  
    const byte * pub,  
    word32 pubSz,  
    const byte * priv,
```



```
    word32 privSz  
)
```

FFC の SP 800-56Ar3、セクション 5.6.2.1.4、メソッド (b) のプロセスに従って、DH 鍵のペアワイズ整合性をチェックします。

B.22.2.14 function wc_DhCheckPrivKey

```
int wc_DhCheckPrivKey(  
    DhKey * key,  
    const byte * priv,  
    word32 pubSz  
)
```

無効な数値について DH 秘密鍵をチェックします

B.22.2.15 function wc_DhCheckPrivKey_ex

```
int wc_DhCheckPrivKey_ex(  
    DhKey * key,  
    const byte * priv,  
    word32 pubSz,  
    const byte * prime,  
    word32 primeSz  
)
```

B.22.2.16 function wc_DhCheckPubKey

```
int wc_DhCheckPubKey(  
    DhKey * key,  
    const byte * pub,  
    word32 pubSz  
)
```

B.22.2.17 function wc_DhCheckPubKey_ex

```
int wc_DhCheckPubKey_ex(  
    DhKey * key,  
    const byte * pub,  
    word32 pubSz,  
    const byte * prime,  
    word32 primeSz  
)
```

B.22.2.18 function wc_DhExportParamsRaw

```
int wc_DhExportParamsRaw(  
    DhKey * dh,  
    byte * p,  
    word32 * pSz,  
    byte * q,  
    word32 * qSz,  
    byte * g,  
    word32 * gSz  
)
```

B.22.2.19 function wc_DhGenerateParams

```
int wc_DhGenerateParams(
    WC_RNG * rng,
    int modSz,
    DhKey * dh
)
```

B.22.2.20 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
)
```

B.22.2.21 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
)
```

B.23 アルゴリズム - ECC**B.23.1 Functions**

	Name
int	wc_ecc_make_key (WC_RNG * rng, int keysize, ecc_key * key) この関数は新しい ecc_key を生成し、key に格納します。
int	wc_ecc_make_key_ex (WC_RNG * rng, int keysize, ecc_key * key, int curve_id) この関数は新しい ecc_key を生成し、key に格納します。
int	wc_ecc_make_pub (ecc_key * key, ecc_point * pubOut)wc_ecc_make_pub は、既存の秘密成分を持つ ecc_key から公開成分を計算します。pubOut が提供されている場合、計算された公開鍵はそこに格納されます。そうでない場合は、提供された ecc_key の公開成分スロットに格納されます。

	Name
int	wc_ecc_make_pub_ex (ecc_key * key, ecc_point * pubOut, WC_RNG * rng)wc_ecc_make_pub_ex は、既存の秘密成分を持つ ecc_key から公開成分を計算します。pubOut が提供されている場合、計算された公開鍵はそこに格納されます。そうでない場合は、提供された ecc_key の公開成分スロットに格納されます。提供された rng が non-NULL の場合、計算で使用する秘密鍵値をブラインドするために使用されます。
int	wc_ecc_check_key (ecc_key * key)ecc キーの有効性に関する健全性チェックを実行します。
void	wc_ecc_key_free (ecc_key * key) この関数は、使用後に ecc_key キーを解放します。
int	wc_ecc_shared_secret (ecc_key * private_key, ecc_key * public_key, byte * out, word32 * outlen) この関数は、ローカル秘密鍵と受信した公開鍵を使用して新しい共有秘密鍵を生成します。この共有秘密鍵をバッファ out に格納し、outlen を更新して出力バッファに書き込まれたバイト数を保持します。
int	wc_ecc_shared_secret_ex (ecc_key * private_key, ecc_point * point, byte * out, word32 * outlen) 秘密鍵と公開ポイント間で ECC 共有秘密を作成します。
int	wc_ecc_sign_hash (const byte * in, word32 inlen, byte * out, word32 * outlen, WC_RNG * rng, ecc_key * key) この関数は、ecc_key オブジェクトを使用してメッセージダイジェストに署名し、真正性を保証します。
int	wc_ecc_sign_hash_ex (const byte * in, word32 inlen, WC_RNG * rng, ecc_key * key, mp_int * r, mp_int * s) メッセージダイジェストに署名します。
int	wc_ecc_verify_hash (const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * stat, ecc_key * key) この関数は、真正性を確保するためにハッシュの ECC 署名を検証します。stat を通じて答えを返し、1 は有効な署名に対応し、0 は無効な署名に対応します。
int	wc_ecc_verify_hash_ex (mp_int * r, mp_int * s, const byte * hash, word32 hashlen, int * stat, ecc_key * key)ECC 署名を検証します。結果は stat に書き込まれます。1 は有効、0 は無効です。注：有効性のテストに戻り値を使用しないでください。stat のみを使用してください。
int	wc_ecc_init (ecc_key * key) この関数は、メッセージ検証または鍵交換で将来使用するために ecc_key オブジェクトを初期化します。
int	wc_ecc_init_ex (ecc_key * key, void * heap, int devId) この関数は、メッセージ検証または鍵交換で将来使用するために ecc_key オブジェクトを初期化します。

	Name
<code>ecc_key *</code>	wc_ecc_key_new (void * heap) この関数は、ユーザー定義のヒープを使用し、キー構造体のスペースを割り当てます。
<code>int</code>	wc_ecc_free (ecc_key * key) この関数は、使用後に ecc_key オブジェクトを解放します。
<code>void</code>	wc_ecc_fp_free (void) この関数は、固定小数点キャッシュを解放します。これは計算時間を高速化するために ecc で使用できます。この機能を使用するには、FP_ECC (固定小数点 ecc) を定義する必要があります。スレッド化されたアプリケーションは、スレッドを終了する前にこの関数を呼び出す必要があります。
<code>int</code>	wc_ecc_is_valid_idx (int n) ECC idx が有効かどうかをチェックします。
<code>ecc_point *</code>	wc_ecc_new_point (void) 新しい ECC ポイントを割り当てます。
<code>void</code>	wc_ecc_del_point (ecc_point * p) メモリから ECC ポイントを解放します。
<code>int</code>	wc_ecc_copy_point (const ecc_point * p, ecc_point * r) あるポイントの値を別のポイントにコピーします。
<code>int</code>	wc_ecc_cmp_point (ecc_point * a, ecc_point * b) ポイントの値を別のポイントと比較します。
<code>int</code>	wc_ecc_point_is_at_infinity (ecc_point * p) ポイントが無限遠点にあるかどうかをチェックします。ポイントが無限遠点にある場合は 1 を返し、そうでない場合は 0 を返し、エラーの場合は <0 を返します
<code>int</code>	wc_ecc_mulmod (const mp_int * k, ecc_point * G, ecc_point * R, mp_int * a, mp_int * modulus, int map) ECC 固定点乗算を実行します。
<code>int</code>	wc_ecc_export_x963 (ecc_key * key, byte * out, word32 * outLen) この関数は、ecc_key 構造体から ECC キーをエクスポートし、結果を out に格納します。キーは ANSI X9.63 形式で格納されます。出力バッファに書き込まれたバイト数を outLen に格納します。
<code>int</code>	wc_ecc_export_x963_ex (ecc_key * key, byte * out, word32 * outLen, int compressed) この関数は、ecc_key 構造体から公開鍵をエクスポートし、結果を out に格納します。キーは ANSI X9.63 形式で格納されます。出力バッファに書き込まれたバイト数を outLen に格納します。この関数は、compressed パラメータを通じて証明書を圧縮する追加オプションを提供します。このパラメータが true の場合、キーは ANSI X9.63 圧縮形式で格納されます。

	Name
int	wc_ecc_import_x963 (const byte * in, word32 inLen, ecc_key * key) この関数は、ANSI X9.63 形式で格納されたキーを含むバッファから公開 ECC キーをインポートします。この関数は、コンパイル時に HAVE_COMP_KEY オプションを通じて圧縮キーが有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。
int	wc_ecc_import_private_key (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ecc_key * key) この関数は、生の秘密鍵を含むバッファと ANSI X9.63 形式の公開鍵を含む 2 番目のバッファから公開/秘密 ECC キーペアをインポートします。この関数は、コンパイル時に HAVE_COMP_KEY オプションを通じて圧縮キーが有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。
int	wc_ecc_rs_to_sig (const char * r, const char * s, byte * out, word32 * outlen) この関数は、ECC 署名の R と S 部分を DER エンコードされた ECDSA 署名に変換します。この関数は、出力バッファ out に書き込まれた長さも outlen に格納します。
int	wc_ecc_import_raw (ecc_key * key, const char * qx, const char * qy, const char * d, const char * curveName) この関数は、ECC 署名の生の成分で ecc_key 構造体を埋めます。
int	wc_ecc_export_private_only (ecc_key * key, byte * out, word32 * outLen) この関数は、ecc_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ecc_export_point_der (const int curve_idx, ecc_point * point, byte * out, word32 * outLen) ポイントを der 形式にエクスポートします。
int	wc_ecc_import_point_der (const byte * in, word32 inLen, const int curve_idx, ecc_point * point) der 形式からポイント进行インポートします。
int	wc_ecc_size (ecc_key * key) この関数は、ecc_key 構造体のキーサイズをオクテット単位で返します。
int	wc_ecc_sig_size_calc (int sz) この関数は、ECC 署名の最悪の場合のサイズを返します。これは (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ で与えられます。実際の署名サイズは、wc_ecc_sign_hash で計算できます。
int	wc_ecc_sig_size (const ecc_key * key) この関数は、ECC 署名の最悪の場合のサイズを返します。これは (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ で与えられます。実際の署名サイズは、wc_ecc_sign_hash で計算できます。

	Name
ecEncCtx *	wc_ecc_ctx_new (int flags, WC_RNG * rng) この関数は、ECC を使用した安全なメッセージ交換を可能にするために、新しい ECC コンテキストオブジェクト用のスペースを割り当てて初期化します。
void	wc_ecc_ctx_free (ecEncCtx * ctx) この関数は、メッセージの暗号化と復号に使用される ecEncCtx オブジェクトを解放します。
int	wc_ecc_ctx_reset (ecEncCtx * ctx, WC_RNG * rng) この関数は、新しいコンテキストオブジェクトを解放して割り当てる必要がないように、ecEncCtx 構造体をリセットします。
int	wc_ecc_ctx_set_algo (ecEncCtx * ctx, byte encAlgo, byte kdfAlgo, byte macAlgo) この関数は、wc_ecc_ctx_new の後にオプションで呼び出すことができます。ecEncCtx オブジェクトに暗号化、KDF、および MAC アルゴリズムを設定します。
const byte *	wc_ecc_ctx_get_own_salt (ecEncCtx *) この関数は、ecEncCtx オブジェクトのソルトを返します。この関数は、ecEncCtx の状態が ecSRV_INIT または ecCLI_INIT の場合にのみ呼び出す必要があります。
int	wc_ecc_ctx_set_peer_salt (ecEncCtx * ctx, const byte * salt) この関数は、ecEncCtx オブジェクトのピアソルトを設定します。
int	wc_ecc_ctx_set_kdf_salt (ecEncCtx * ctx, const byte * salt, word32 sz) この関数は、KDF で使用するソルトポインタと長さを ecEncCtx オブジェクトに設定します。
int	wc_ecc_ctx_set_info (ecEncCtx * ctx, const byte * info, int sz) この関数は、wc_ecc_ctx_set_peer_salt の前または後にオプションで呼び出すことができます。ecEncCtx オブジェクトのオプション情報を設定します。
int	wc_ecc_encrypt (ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx) この関数は、msg から out へ指定された入力メッセージを暗号化します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。

	Name
int	wc_ecc_encrypt_ex (ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx, int compressed) この関数は、msg から out へ指定された入力メッセージを暗号化します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。
int	wc_ecc_decrypt (ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx) この関数は、msg から out へ暗号文を復号します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。
int	wc_ecc_set_nonblock (ecc_key * key, ecc_nb_ctx_t * ctx) ノンブロッキング操作のための ECC サポートを有効にします。次のビルドオプションで Single Precision (SP) 数学でサポートされています：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) 指定されたサイズより大きいキーを持つカーブまたはカーブ ID に一致するカーブを比較し、より小さいキーサイズを持つカーブをキーに設定します。

B.23.2 Functions Documentation

B.23.2.1 function wc_ecc_make_key

```
int wc_ecc_make_key(
    WC_RNG * rng,
    int keysize,
    ecc_key * key
)
```

この関数は新しい ecc_key を生成し、key に格納します。

Parameters:

- **rng** キーの生成に使用する初期化された RNG オブジェクトへのポインタ
- **keysize** ecc_key の希望する長さ
- **key** キーを生成する ecc_key へのポインタ

See:

- [wc_ecc_init](#)
- [wc_ecc_shared_secret](#)

Return:

- 0 成功時に返されます。
- ECC_BAD_ARG_E rng または key が NULL と評価された場合に返されます
- BAD_FUNC_ARG 指定されたキーサイズがサポートされているキーの正しい範囲内でない場合に返されます
- MEMORY_E ecc キーの計算中にメモリ割り当てエラーがある場合に返されます
- MP_INIT_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_READ_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_CMP_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MUL_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_ADD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_TO_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MEM ecc キーの計算中にエラーがある場合に返される可能性があります

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key); // 32 バイトの ecc キーを初期化
```

B.23.2.2 function wc_ecc_make_key_ex

```
int wc_ecc_make_key_ex(
    WC_RNG * rng,
    int keysize,
    ecc_key * key,
    int curve_id
)
```

この関数は新しい ecc_key を生成し、key に格納します。

Parameters:

- **key** 作成されたキーを格納するポインタ。
- **keysize** 作成されるキーのサイズ (バイト単位)、curveId に基づいて設定
- **rng** キー作成で使用する Rng
- **curve_id** キーに使用するカーブ

See:

- [wc_ecc_make_key](#)
- [wc_ecc_get_curve_size_from_id](#)

Return:

- 0 成功時に返されます。
- ECC_BAD_ARG_E rng または key が NULL と評価された場合に返されます
- BAD_FUNC_ARG 指定されたキーサイズがサポートされているキーの正しい範囲内でない場合に返されます
- MEMORY_E ecc キーの計算中にメモリ割り当てエラーがある場合に返されます
- MP_INIT_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_READ_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_CMP_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MUL_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_ADD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_TO_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MEM ecc キーの計算中にエラーがある場合に返される可能性があります

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) {
    // エラー処理
}
```

B.23.2.3 function wc_ecc_make_pub

```
int wc_ecc_make_pub(
    ecc_key * key,
    ecc_point * pubOut
)
```

wc_ecc_make_pub は、既存の秘密成分を持つ ecc_key から公開成分を計算します。pubOut が提供されている場合、計算された公開鍵はそこに格納されます。そうでない場合は、提供された ecc_key の公開成分スロットに格納されます。

Parameters:

- **key** 有効な秘密成分を含む ecc_key へのポインタ
- **pubOut** 計算された公開鍵を格納する ecc_point 構造体へのオプションのポインタ

See:

- wc_ecc_make_pub_ex
- wc_ecc_make_key

Return:

- 0 成功時に返されます。
- ECC_BAD_ARG_E key が NULL の場合に返されます
- BAD_FUNC_ARG 提供されたキーが有効な ecc_key でない場合に返されます。
- MEMORY_E 公開鍵の計算中にメモリ割り当てエラーがある場合に返されます
- MP_INIT_E 公開鍵の計算中にエラーがある場合に返される可能性があります

- MP_READ_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_ADD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_TO_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MEM 公開鍵の計算中にエラーがある場合に返される可能性があります
- ECC_OUT_OF_RANGE_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- ECC_PRIV_KEY_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- ECC_INF_E 公開鍵の計算中にエラーがある場合に返される可能性があります

B.23.2.4 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 は、既存の秘密成分を持つ ecc_key から公開成分を計算します。pubOut が提供されている場合、計算された公開鍵はそこに格納されます。そうでない場合は、提供された ecc_key の公開成分スロットに格納されます。提供された rng が non-NULL の場合、計算で使用される秘密鍵値をブラインドするために使用されます。

Parameters:

- **key** 有効な秘密成分を含む ecc_key へのポインタ
- **pubOut** 計算された公開鍵を格納する ecc_point 構造体へのオプションのポインタ
- **rng** 公開鍵の計算で使用する Rng

See:

- [wc_ecc_make_pub](#)
- [wc_ecc_make_key](#)
- [wc_InitRng](#)

Return:

- 0 成功時に返されます。
- ECC_BAD_ARG_E key が NULL の場合に返されます
- BAD_FUNC_ARG 提供されたキーが有効な ecc_key でない場合に返されます。
- MEMORY_E 公開鍵の計算中にメモリ割り当てエラーがある場合に返されます
- MP_INIT_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_READ_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_ADD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_TO_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MEM 公開鍵の計算中にエラーがある場合に返される可能性があります
- ECC_OUT_OF_RANGE_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- ECC_PRIV_KEY_E 公開鍵の計算中にエラーがある場合に返される可能性があります

- ECC_INF_E 公開鍵の計算中にエラーがある場合に返される可能性があります

B.23.2.5 function wc_ecc_check_key

```
int wc_ecc_check_key(  
    ecc_key * key  
)
```

ecc キーの有効性に関する健全性チェックを実行します。

Parameters:

- **key** チェックするキーへのポインタ。

See: [wc_ecc_point_is_at_infinity](#)

Return:

- MP_OKAY 成功、キーは正常です。
- BAD_FUNC_ARG key が NULL の場合に返されます。
- ECC_INF_E wc_ecc_point_is_at_infinity が 1 を返す場合に返されます。

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)  
{  
    // キーチェック成功  
}  
else  
{  
    // キーチェック失敗  
}
```

B.23.2.6 function wc_ecc_key_free

```
void wc_ecc_key_free(  
    ecc_key * key  
)
```

この関数は、使用後に ecc_key キーを解放します。

Parameters:

- **key** 解放する ecc_key 構造体へのポインタ

See:

- [wc_ecc_key_new](#)
- [wc_ecc_init_ex](#)

Example

```
// キーを初期化して ECC 操作を実行
...
wc_ecc_key_free(&key);
```

B.23.2.7 function wc_ecc_shared_secret

```
int wc_ecc_shared_secret(
    ecc_key * private_key,
    ecc_key * public_key,
    byte * out,
    word32 * outlen
)
```

この関数は、ローカル秘密鍵と受信した公開鍵を使用して新しい共有秘密鍵を生成します。この共有秘密鍵をバッファ out に格納し、outlen を更新して出力バッファに書き込まれたバイト数を保持します。

Parameters:

- **private_key** ローカル秘密鍵を含む ecc_key 構造体へのポインタ
- **public_key** 受信した公開鍵を含む ecc_key 構造体へのポインタ
- **out** 生成された共有秘密鍵を格納する出力バッファへのポインタ
- **outlen** 出力バッファの長さを含む word32 オブジェクトへのポインタ。共有秘密鍵の生成に成功すると、出力バッファに書き込まれた長さで上書きされます

See:

- [wc_ecc_init](#)
- [wc_ecc_make_key](#)

Return:

- 0 共有秘密鍵の生成に成功した場合に返されます
- BAD_FUNC_ARG いずれかの入力パラメータが NULL と評価された場合に返されます
- ECC_BAD_ARG_E 引数として与えられた秘密鍵 private_key のタイプが ECC_PRIVATEKEY でない場合、または公開鍵と秘密鍵のタイプ (ecc->dp で指定) が同等でない場合に返されます
- MEMORY_E 新しい ecc ポイントの生成中にエラーがある場合に返されます
- BUFFER_E 生成された共有秘密鍵が提供されたバッファに格納するには長すぎる場合に返されます
- MP_INIT_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_READ_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_ADD_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_TO_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_MEM 共有鍵の計算中にエラーがある場合に返される可能性があります

Example

```
ecc_key priv, pub;
WC_RNG rng;
byte secret[1024]; // 1024 バイトの共有秘密鍵を保持可能
word32 secretSz = sizeof(secret);
int ret;

wc_InitRng(&rng); // rng を初期化
wc_ecc_init(&priv); // キーを初期化
```

```

wc_ecc_make_key(&rng, 32, &priv); // 公開/秘密鍵ペアを作成
// 公開鍵を受信し、pub に初期化
ret = wc_ecc_shared_secret(&priv, &pub, secret, &secretSz);
// 共有秘密鍵を生成
if ( ret != 0 ) {
    // 共有秘密鍵の生成エラー
}

```

B.23.2.8 function wc_ecc_shared_secret_ex

```

int wc_ecc_shared_secret_ex(
    ecc_key * private_key,
    ecc_point * point,
    byte * out,
    word32 * outlen
)

```

秘密鍵と公開ポイント間で ECC 共有秘密を作成します。

Parameters:

- **private_key** 秘密 ECC キー。
- **point** 使用するポイント (公開鍵)。
- **out** 共有秘密の出力先。ANSI X9.63 の EC-DH に準拠。
- **outlen** 最大サイズを入力し、共有秘密の結果サイズを出力。

See: [wc_ecc_verify_hash_ex](#)

Return:

- MP_OKAY 成功を示します。
- BAD_FUNC_ARG いずれかの引数が null の場合に返されるエラー。
- ECC_BAD_ARG_E private_key->type が ECC_PRIVATEKEY でない場合、または private_key->idx の検証に失敗した場合に返されるエラー。
- BUFFER_E outlen が小さすぎる場合のエラー。
- MEMORY_E 新しいポイントを作成する際のエラー。
- MP_VAL 初期化失敗が発生した場合に返される可能性があります。
- MP_MEM 初期化失敗が発生した場合に返される可能性があります。

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)
{
    // エラーを処理
}

```

B.23.2.9 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
)
```

この関数は、ecc_key オブジェクトを使用してメッセージダイジェストに署名し、真正性を保証します。

Parameters:

- **in** 署名するメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージハッシュの長さ
- **out** 生成された署名を格納するバッファ
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功すると、out に書き込まれたバイト数を格納します
- **key** 署名を生成するために使用する秘密 ECC キーへのポインタ

See: [wc_ecc_verify_hash](#)

Return:

- 0 メッセージダイジェストの署名を正常に生成した場合に返されます
- BAD_FUNC_ARG いずれかの入力パラメータが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます
- ECC_BAD_ARG_E 入力キーが秘密鍵でない場合、または ECC OID が無効な場合に返されます
- RNG_FAILURE_E rng が満足のいくキーを正常に生成できない場合に返されます
- MP_INIT_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_READ_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_ADD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_TO_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MEM メッセージ署名の計算中にエラーがある場合に返される可能性があります

Example

```
ecc_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[512]; // 生成された署名を保持
sigSz = sizeof(sig);
byte digest[] = { // メッセージハッシュで初期化 };
wc_InitRng(&rng); // rng を初期化
wc_ecc_init(&key); // キーを初期化
wc_ecc_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
ret = wc_ecc_sign_hash(digest, sizeof(digest), sig, &sigSz, &key);
if ( ret != 0 ) {
    // メッセージ署名の生成エラー
}
```


B.23.2.10 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
)
```

メッセージダイジェストに署名します。

Parameters:

- **in** 署名するメッセージダイジェスト。
- **inlen** ダイジェストの長さ。
- **rng** WC_RNG 構造体へのポインタ。
- **key** 秘密 ECC キー。
- **r** 署名の r 成分の出力先。
- **s** 署名の s 成分の出力先。

See: [wc_ecc_verify_hash_ex](#)

Return:

- MP_OKAY メッセージダイジェストの署名を正常に生成した場合に返されます
- ECC_BAD_ARG_E 入力キーが秘密鍵でない場合、または ECC_IDX が無効な場合、またはいずれかの入力パラメータが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます
- RNG_FAILURE_E rng が満足のいくキーを正常に生成できない場合に返されます
- MP_INIT_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_READ_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_ADD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_TO_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MEM メッセージ署名の計算中にエラーがある場合に返される可能性があります

Example

```
ecc_key key;
WC_RNG rng;
int ret, sigSz;
mp_int r; // 署名の r 成分の出力先。
mp_int s; // 署名の s 成分の出力先。

byte sig[512]; // 生成された署名を保持
sigSz = sizeof(sig);
byte digest[] = { メッセージハッシュで初期化 };
wc_InitRng(&rng); // rng を初期化
wc_ecc_init(&key); // キーを初期化
mp_init(&r); // r 成分を初期化
mp_init(&s); // s 成分を初期化
wc_ecc_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
```

```
ret = wc_ecc_sign_hash_ex(digest, sizeof(digest), &rng, &key, &r, &s);

if ( ret != MP_OKAY ) {
    // メッセージ署名の生成エラー
}
```

B.23.2.11 function wc_ecc_verify_hash

```
int wc_ecc_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashlen,
    int * stat,
    ecc_key * key
)
```

この関数は、真正性を確保するためにハッシュの ECC 署名を検証します。stat を通じて答えを返し、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ
- **siglen** 検証する署名の長さ
- **hash** 検証されたメッセージのハッシュを含むバッファへのポインタ
- **hashlen** 検証されたメッセージのハッシュの長さ
- **stat** 検証結果へのポインタ。1 はメッセージが正常に検証されたことを示します
- **key** 署名を検証するために使用する公開 ECC キーへのポインタ

See:

- [wc_ecc_sign_hash](#)
- [wc_ecc_verify_hash_ex](#)

Return:

- 0 署名検証の実行に成功した場合に返されます。注：これは署名が検証されたことを意味するものではありません。真正性情報は代わりに stat に格納されます
- BAD_FUNC_ARG いずれかの入力パラメータが NULL と評価された場合に返されます
- MEMORY_E メモリ割り当てエラーがある場合に返されます
- MP_INIT_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_READ_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_ADD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_TO_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MEM メッセージ署名の計算中にエラーがある場合に返される可能性があります

Example

```
ecc_key key;
int ret, verified = 0;

byte sig[1024] { 受信した署名で初期化 };
byte digest[] = { メッセージハッシュで初期化 };
```



```
// 受信した公開鍵でキーを初期化
ret = wc_ecc_verify_hash(sig, sizeof(sig), digest, sizeof(digest),
&verified, &key);
if ( ret != 0 ) {
    // 検証実行エラー
} else if ( verified == 0 ) {
    // 署名が無効
}
```

B.23.2.12 function wc_ecc_verify_hash_ex

```
int wc_ecc_verify_hash_ex(
    mp_int * r,
    mp_int * s,
    const byte * hash,
    word32 hashlen,
    int * stat,
    ecc_key * key
)
```

ECC 署名を検証します。結果は stat に書き込まれます。1 は有効、0 は無効です。注：有効性のテストに戻り値を使用しないでください。stat のみを使用してください。

Parameters:

- **r** 検証する署名の R 成分
- **s** 検証する署名の S 成分
- **hash** 署名されたハッシュ（メッセージダイジェスト）
- **hashlen** ハッシュの長さ（オクテット）
- **stat** 署名の結果、1== 有効、0== 無効
- **key** 対応する公開 ECC キー

See: [wc_ecc_verify_hash](#)

Return:

- MP_OKAY 成功した場合（署名が有効でない場合でも）
- ECC_BAD_ARG_E 引数が null の場合、または key-idx が無効な場合に返されます。
- MEMORY_E 整数またはポイントの割り当てエラー。

Example

```
mp_int r;
mp_int s;
int stat;
byte hash[] = { いくつかのハッシュ }
ecc_key key;

if(wc_ecc_verify_hash_ex(&r, &s, hash, hashlen, &stat, &key) == MP_OKAY)
{
    // stat をチェック
}
```

B.23.2.13 function wc_ecc_init

```
int wc_ecc_init(
    ecc_key * key
)
```

この関数は、メッセージ検証または鍵交換で将来使用するために ecc_key オブジェクトを初期化します。

Parameters:

- **key** 初期化する ecc_key オブジェクトへのポインタ

See:

- [wc_ecc_make_key](#)
- [wc_ecc_free](#)

Return:

- 0 ecc_key オブジェクトの初期化に成功した場合に返されます
- MEMORY_E メモリ割り当てエラーがある場合に返されます

Example

```
ecc_key key;  
wc_ecc_init(&key);
```

B.23.2.14 function wc_ecc_init_ex

```
int wc_ecc_init_ex(  
    ecc_key * key,  
    void * heap,  
    int devId  
)
```

この関数は、メッセージ検証または鍵交換で将来使用するために ecc_key オブジェクトを初期化します。

Parameters:

- **key** 初期化する ecc_key オブジェクトへのポインタ
- **heap** ヒープ識別子へのポインタ
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定

See:

- [wc_ecc_make_key](#)
- [wc_ecc_free](#)
- [wc_ecc_init](#)

Return:

- 0 ecc_key オブジェクトの初期化に成功した場合に返されます
- MEMORY_E メモリ割り当てエラーがある場合に返されます

Example

```
ecc_key key;  
wc_ecc_init_ex(&key, heap, devId);
```

B.23.2.15 function wc_ecc_key_new

```
ecc_key * wc_ecc_key_new(  
    void * heap  
)
```

この関数は、ユーザー定義のヒープを使用し、キー構造体用のスペースを割り当てます。

See:

- `wc_ecc_make_key`
- `wc_ecc_key_free`
- `wc_ecc_init`

Return:

- 0 `ecc_key` オブジェクトの初期化に成功した場合に返されます
- `MEMORY_E` メモリ割り当てエラーがある場合に返されます

Example

```
wc_ecc_key_new(&heap);
```

B.23.2.16 function `wc_ecc_free`

```
int wc_ecc_free(
    ecc_key * key
)
```

この関数は、使用後に `ecc_key` オブジェクトを解放します。

Parameters:

- **key** 解放する `ecc_key` オブジェクトへのポインタ

See: `wc_ecc_init`

Return: int wolfSSL のエラーまたは成功ステータスを示す整数が返されます。

Example

```
// キーを初期化して安全な交換を実行
...
wc_ecc_free(&key);
```

B.23.2.17 function `wc_ecc_fp_free`

```
void wc_ecc_fp_free(
    void
)
```

この関数は、固定小数点キャッシュを解放します。これは計算時間を高速化するために `ecc` で使用できます。この機能を使用するには、`FP_ECC`（固定小数点 `ecc`）を定義する必要があります。スレッド化されたアプリケーションは、スレッドを終了する前にこの関数を呼び出す必要があります。

Parameters:

- **none** パラメータなし。

See: `wc_ecc_free`

Return: none 返り値なし。

Example

```
ecc_key key;
// キーを初期化して安全な交換を実行
...

wc_ecc_fp_free();
```

B.23.2.18 function wc_ecc_is_valid_idx

```
int wc_ecc_is_valid_idx(  
    int n  
)
```

ECC idx が有効かどうかをチェックします。

Parameters:

- **n** チェックする idx 番号。

See: なし

Return:

- 1 有効な場合に返されます。
- 0 有効でない場合に返されます。

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 は有効  
}  
else if (is_valid == 0)  
{  
    // idx は無効  
}
```

B.23.2.19 function wc_ecc_new_point

```
ecc_point * wc_ecc_new_point(  
    void  
)
```

新しい ECC ポイントを割り当てます。

Parameters:

- **none** パラメータなし。

See:

- `wc_ecc_del_point`
- `wc_ecc_cmp_point`
- `wc_ecc_copy_point`

Return:

- p 新しく割り当てられたポイント。
- NULL エラー時に NULL を返します。

Example

```
ecc_point* point;  
point = wc_ecc_new_point();  
if (point == NULL)  
{  
    // ポイント作成エラーを処理  
}  
// ポイントで何かを行う
```

B.23.2.20 function wc_ecc_del_point

```
void wc_ecc_del_point(  
    ecc_point * p  
)
```

メモリから ECC ポイントを解放します。

Parameters:

- **p** 解放するポイント。

See:

- [wc_ecc_new_point](#)
- [wc_ecc_cmp_point](#)
- [wc_ecc_copy_point](#)

Return: none 返り値なし。

Example

```
ecc_point* point;  
point = wc_ecc_new_point();  
if (point == NULL)  
{  
    // ポイント作成エラーを処理  
}  
// ポイントで何かを行う  
wc_ecc_del_point(point);
```

B.23.2.21 function wc_ecc_copy_point

```
int wc_ecc_copy_point(  
    const ecc_point * p,  
    ecc_point * r  
)
```

あるポイントの値を別のポイントにコピーします。

Parameters:

- **p** コピーするポイント。
- **r** 作成されたポイント。

See:

- [wc_ecc_new_point](#)
- [wc_ecc_cmp_point](#)
- [wc_ecc_del_point](#)

Return:

- ECC_BAD_ARG_E p または r が null の場合にスローされるエラー。

- MP_OKAY ポイントが正常にコピーされました
- ret 内部関数からのエラー。次のような可能性があります...

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)
{
    // エラーを処理
}
```

B.23.2.22 function wc_ecc_cmp_point

```
int wc_ecc_cmp_point(
    ecc_point * a,
    ecc_point * b
)
```

ポイントの値を別のポイントと比較します。

Parameters:

- **a** 比較する最初のポイント。
- **b** 比較する 2 番目のポイント。

See:

- `wc_ecc_new_point`
- `wc_ecc_del_point`
- `wc_ecc_copy_point`

Return:

- BAD_FUNC_ARG 一方または両方の引数が NULL。
- MP_EQ ポイントが等しい。
- ret MP_LT または MP_GT のいずれかで、ポイントが等しくないことを示します。

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)
{
    // 引数が無効
}
else if (cmp_result == MP_EQ)
{
    // ポイントが等しい
}
else
```

```
{
    // ポイントが等しくない
}
```

B.23.2.23 function wc_ecc_point_is_at_infinity

```
int wc_ecc_point_is_at_infinity(
    ecc_point * p
)
```

ポイントが無限遠点にあるかどうかをチェックします。ポイントが無限遠点にある場合は 1 を返し、そうでない場合は 0 を返し、エラーの場合は <0 を返します

Parameters:

- **p** チェックするポイント。

See:

- `wc_ecc_new_point`
- `wc_ecc_del_point`
- `wc_ecc_cmp_point`
- `wc_ecc_copy_point`

Return:

- 1 p が無限遠点にある。
- 0 p が無限遠点にない。
- <0 エラー。

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)
{
    // エラーを処理
}
else if (is_infinity == 0)
{
    // ポイントは無限遠点にない
}
else if (is_infinity == 1)
{
    // ポイントは無限遠点にある
}
```

B.23.2.24 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
)
```

ECC 固定点乗算を実行します。

Parameters:

- **k** 被乗数。
- **G** 乗算する基点。
- **R** 積の出力先。
- **modulus** カーブのモジュラス。
- **map** ゼロでない場合、ポイントをアフィン座標に戻してマップします。そうでない場合は、ヤコビ・モンゴメリー形式のままです。

See: なし

Return:

- MP_OKAY 操作が成功した場合に返されます。
- MP_INIT_E 多精度整数 (mp_int) ライブラリで使用するために整数を初期化の際にエラーがある場合に返されます。

Example

```
ecc_point* base;
ecc_point* destination;
// ポイントを初期化
base = wc_ecc_new_point();
destination = wc_ecc_new_point();
// 他の引数を設定
mp_int multiplicand;
mp_int modulus;
int map;
```

B.23.2.25 function wc_ecc_export_x963

```
int wc_ecc_export_x963(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ecc_key 構造体から ECC キーをエクスポートし、結果を out に格納します。キーは ANSI X9.63 形式で格納されます。出力バッファに書き込まれたバイト数を outLen に格納します。

Parameters:

- **key** エクスポートする ecc_key オブジェクトへのポインタ
- **out** ANSI X9.63 形式のキーを格納するバッファへのポインタ
- **outLen** 出力バッファのサイズ。キーの格納に成功すると、出力バッファに書き込まれたバイト数を保持します

See:

- [wc_ecc_export_x963_ex](#)
- [wc_ecc_import_x963](#)
- [wc_ecc_make_pub](#)

Return:

- 0 ecc_key のエクスポートに成功した場合に返されます
- LENGTH_ONLY_E 出力バッファが NULL と評価されるが、他の 2 つの入力パラメータが有効な場合に返されます。関数がキーを格納するために必要な長さのみを返していることを示します

- ECC_BAD_ARG_E いずれかの入力パラメータが NULL の場合、またはキーがサポートされていない (無効なインデックスを持つ) 場合に返されます
- BUFFER_E 出力バッファが ecc キーを格納するには小さすぎる場合に返されます。出力バッファが小さすぎる場合、必要なサイズが outLen で返されます
- MEMORY_E XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- MP_INIT_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_READ_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_CMP_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_INVMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MUL_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_ADD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MULMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_TO_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MEM ecc_key の処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);

ecc_key key;
// キーを初期化し、キーを作成
ret = wc_ecc_export_x963(&key, buff, &buffSz);
if ( ret != 0 ) {
    // キーのエクスポートエラー
}
```

B.23.2.26 function wc_ecc_export_x963_ex

```
int wc_ecc_export_x963_ex(
    ecc_key * key,
    byte * out,
    word32 * outLen,
    int compressed
)
```

この関数は、ecc_key 構造体から公開鍵をエクスポートし、結果を out に格納します。キーは ANSI X9.63 形式で格納されます。出力バッファに書き込まれたバイト数を outLen に格納します。この関数は、compressed パラメータを通じて証明書を圧縮する追加オプションを提供します。このパラメータが true の場合、キーは ANSI X9.63 圧縮形式で格納されます。

Parameters:

- **key** エクスポートする ecc_key オブジェクトへのポインタ
- **out** ANSI X9.63 形式の公開鍵を格納するバッファへのポインタ
- **outLen** 出力バッファのサイズ。公開鍵の格納に成功すると、出力バッファに書き込まれたバイト数を保持します
- **compressed** キーを圧縮形式で格納するかどうかの指標。1== 圧縮、0== 非圧縮

See:

- [wc_ecc_export_x963](#)
- [wc_ecc_import_x963](#)
- [wc_ecc_make_pub](#)

Return:

- 0 ecc_key 公開成分のエクスポートに成功した場合に返されます
- ECC_PRIVATEKEY_ONLY ecc_key 公開成分が欠落している場合に返されます
- NOT_COMPILED_IN コンパイル時に HAVE_COMP_KEY が有効になっていないが、キーが圧縮形式で要求された場合に返されます
- LENGTH_ONLY_E 出力バッファが NULL と評価されるが、他の 2 つの入力パラメータが有効な場合に返されます。関数が公開鍵を格納するために必要な長さのみを返していることを示します
- ECC_BAD_ARG_E いずれかの入力パラメータが NULL の場合、またはキーがサポートされていない（無効なインデックスを持つ）場合に返されます
- BUFFER_E 出力バッファが公開鍵を格納するには小さすぎる場合に返されます。出力バッファが小さすぎる場合、必要なサイズが outLen で返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- MP_INIT_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_READ_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_CMP_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_INVMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MUL_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_ADD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MULMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_TO_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MEM ecc_key の処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);
ecc_key key;
// キーを初期化し、キーを作成
ret = wc_ecc_export_x963_ex(&key, buff, &buffSz, 1);
if ( ret != 0 ) {
    // キーのエクスポートエラー
}
```

B.23.2.27 function wc_ecc_import_x963

```
int wc_ecc_import_x963(
    const byte * in,
    word32 inLen,
    ecc_key * key
)
```

この関数は、ANSI X9.63 形式で格納されたキーを含むバッファから公開 ECC キーをインポートします。この関数は、コンパイル時に HAVE_COMP_KEY オプションを通じて圧縮キーが有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **in** ANSI x9.63 形式の ECC キーを含むバッファへのポインタ
- **inLen** 入力バッファの長さ
- **key** インポートされたキーを格納する ecc_key オブジェクトへのポインタ

See:

- [wc_ecc_export_x963](#)

- `wc_ecc_import_private_key`

Return:

- 0 `ecc_key` のインポートに成功した場合に返されます
- `NOT_COMPILED_IN` コンパイル時に `HAVE_COMP_KEY` が有効になっていないが、キーが圧縮形式で格納されている場合に返されます
- `ECC_BAD_ARG_E` `in` または `key` が `NULL` と評価される場合、または `inLen` が偶数の場合に返されます (x9.63 標準によると、キーは奇数でなければなりません)
- `MEMORY_E` メモリの割り当てエラーがある場合に返されます
- `ASN_PARSE_E` ECC キーの解析エラーがある場合に返されます。ECC キーが有効な ANSI X9.63 形式で格納されていないことを示す可能性があります
- `IS_POINT_E` エクスポートされた公開鍵が ECC 曲線上のポイントでない場合に返されます
- `MP_INIT_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_READ_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_CMP_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_INVMOD_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_EXPTMOD_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MOD_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MUL_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_ADD_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MULMOD_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_TO_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MEM` `ecc_key` の処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
byte buff[] = { ANSI_X9.63形式のキーで初期化 };

ecc_key pubKey;
wc_ecc_init(&pubKey);

ret = wc_ecc_import_x963(buff, sizeof(buff), &pubKey);
if ( ret != 0 ) {
    // キーのインポートエラー
}
```

B.23.2.28 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
)
```

この関数は、生の秘密鍵を含むバッファと ANSI X9.63 形式の公開鍵を含む 2 番目のバッファから公開/秘密 ECC キーペアをインポートします。この関数は、コンパイル時に `HAVE_COMP_KEY` オプションを通じて圧縮キーが有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **priv** 生の秘密鍵を含むバッファへのポインタ
- **privSz** 秘密鍵バッファのサイズ
- **pub** ANSI x9.63 形式の ECC 公開鍵を含むバッファへのポインタ
- **pubSz** 公開鍵入力バッファの長さ

- **key** インポートされた秘密/公開鍵ペアを格納する ecc_key オブジェクトへのポインタ

See:

- [wc_ecc_export_x963](#)
- [wc_ecc_import_private_key](#)

Return:

- 0 ecc_key のインポートに成功した場合に返されます NOT_COMPILED_IN コンパイル時に HAVE_COMP_KEY が有効になっていないが、キーが圧縮形式で格納されている場合に返されます
- ECC_BAD_ARG_E in または key が NULL と評価される場合、または inLen が偶数の場合に返されます (x9.63 標準によると、キーは奇数でなければなりません)
- MEMORY_E メモリの割り当てエラーがある場合に返されます
- ASN_PARSE_E ECC キーの解析エラーがある場合に返されます。ECC キーが有効な ANSI X9.63 形式で格納されていないことを示す可能性があります
- IS_POINT_E エクスポートされた公開鍵が ECC 曲線上のポイントでない場合に返されます
- MP_INIT_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_READ_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_CMP_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_INVMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MUL_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_ADD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MULMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_TO_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MEM ecc_key の処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
byte pub[] = { ANSI_X9.63形式のキーで初期化 };
byte priv[] = { 生の秘密鍵で初期化 };

ecc_key key;
wc_ecc_init(&key);
ret = wc_ecc_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
&key);
if ( ret != 0 ) {
    // キーのインポートエラー
}
```

B.23.2.29 function wc_ecc_rs_to_sig

```
int wc_ecc_rs_to_sig(
    const char * r,
    const char * s,
    byte * out,
    word32 * outlen
)
```

この関数は、ECC 署名の R と S 部分を DER エンコードされた ECDSA 署名に変換します。この関数は、出力バッファ out に書き込まれた長さも outlen に格納します。

Parameters:

- **r** 文字列として署名の R 部分を含むバッファへのポインタ
- **s** 文字列として署名の S 部分を含むバッファへのポインタ

- **out** DER エンコードされた ECDSA 署名を格納するバッファへのポインタ
- **outlen** 利用可能な出力バッファの長さ。署名を ECDSA 形式に正常に変換した後、バッファに書き込まれたバイト数を格納します

See:

- [wc_ecc_sign_hash](#)
- [wc_ecc_sig_size](#)

Return:

- 0 署名の変換に成功した場合に返されます
- `ECC_BAD_ARG_E` いずれかの入力パラメータが NULL と評価される場合、または入力バッファが DER エンコードされた ECDSA 署名を保持するのに十分な大きさでない場合に返されます
- `MP_INIT_E ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_READ_E ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_CMP_E ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_INVMOD_E ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_EXPTMOD_E ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MOD_E ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MUL_E ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_ADD_E ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MULMOD_E ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_TO_E ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MEM ecc_key` の処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
ecc_key key;
// キーを初期化し、R と S を生成

char r[] = { R で初期化 };
char s[] = { S で初期化 };
byte sig[wc_ecc_sig_size(key)];
// 署名サイズは 2 * ECC キーサイズ + ASN.1 オーバーヘッド用の約 10 バイトになります
word32 sigSz = sizeof(sig);
ret = wc_ecc_rs_to_sig(r, s, sig, &sigSz);
if (ret != 0) {
    // パラメータから署名への変換エラー
}
```

B.23.2.30 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
)
```

この関数は、ECC 署名の生の成分で `ecc_key` 構造体を埋めます。

Parameters:

- **key** 埋める `ecc_key` 構造体へのポインタ
- **qx** ASCII 16 進文字列として基点の x 成分を含むバッファへのポインタ
- **qy** ASCII 16 進文字列として基点の y 成分を含むバッファへのポインタ

- **d** ASCII 16 進文字列として秘密鍵を含むバッファへのポインタ
- **curveName** ecc_sets にある ECC 曲線名を含む文字列へのポインタ

See: [wc_ecc_import_private_key](#)

Return:

- 0 ecc_key 構造体へのインポートに成功した場合に返されます
- ECC_BAD_ARG_E いずれかの入力値が NULL と評価された場合に返されます
- MEMORY_E ecc_key のパラメータを格納するためのスペースを初期化する際にエラーがある場合に返されます
- ASN_PARSE_E 入力 curveName が ecc_sets で定義されていない場合に返されます
- MP_INIT_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_READ_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_CMP_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_INVMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MUL_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_ADD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MULMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_TO_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MEM 入力パラメータの処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
ecc_key key;
wc_ecc_init(&key);

char qx[] = { 基点の x 成分で初期化 };
char qy[] = { 基点の y 成分で初期化 };
char d[] = { 秘密鍵で初期化 };
ret = wc_ecc_import_raw(&key, qx, qy, d, "ECC-256");
if ( ret != 0 ) {
    // 指定された入力でキーを初期化する際のエラー
}
```

B.23.2.31 function wc_ecc_export_private_only

```
int wc_ecc_export_private_only(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ecc_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 秘密鍵をエクスポートする ecc_key 構造体へのポインタ
- **out** 秘密鍵を格納するバッファへのポインタ
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。秘密鍵のエクスポートに成功した後、out に書き込まれたバイト数で設定されます

See: [wc_ecc_import_private_key](#)

Return:

- 0 秘密鍵のエクスポートに成功した場合に返されます
- ECC_BAD_ARG_E いずれかの入力値が NULL と評価された場合に返されます
- MEMORY_E ecc_key のパラメータを格納するためのスペースを初期化する際にエラーがある場合に返されます
- ASN_PARSE_E 入力 curveName が ecc_sets で定義されていない場合に返されます
- MP_INIT_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_READ_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_CMP_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_INVMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MUL_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_ADD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MULMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_TO_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MEM 入力パラメータの処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
ecc_key key;
// キーを初期化し、キーを作成

char priv[ECC_KEY_SIZE];
word32 privSz = sizeof(priv);
ret = wc_ecc_export_private_only(&key, priv, &privSz);
if ( ret != 0 ) {
    // 秘密鍵のエクスポートエラー
}
```

B.23.2.32 function wc_ecc_export_point_der

```
int wc_ecc_export_point_der(
    const int curve_idx,
    ecc_point * point,
    byte * out,
    word32 * outLen
)
```

ポイントを der 形式にエクスポートします。

Parameters:

- **curve_idx** ecc_sets から使用されるカーブのインデックス。
- **point** der にエクスポートするポイント。
- **out** 出力の出力先。
- **outLen** 出力に許可される最大サイズ、出力の最終サイズの出力先

See: [wc_ecc_import_point_der](#)

Return:

- 0 成功時に返されます。
- ECC_BAD_ARG_E curve_idx が 0 未満または無効な場合に返されます。また、次の場合にも返されます
- LENGTH_ONLY_E outLen が設定されているが他は何もない。
- BUFFER_E outLen が 1 + 2 * カーブサイズ未満の場合に返されます。
- MEMORY_E メモリの割り当てに問題がある場合に返されます。

Example

```
int curve_idx;
ecc_point* point;
byte out[];
word32 outLen;
wc_ecc_export_point_der(curve_idx, point, out, &outLen);
```

B.23.2.33 function wc_ecc_import_point_der

```
int wc_ecc_import_point_der(
    const byte * in,
    word32 inLen,
    const int curve_idx,
    ecc_point * point
)
```

der 形式からポイントをインポートします。

Parameters:

- **in** ポイントをインポートする der バッファ。
- **inLen** der バッファの長さ。
- **curve_idx** カーブのインデックス。
- **point** ポイントの出力先。

See: [wc_ecc_export_point_der](#)

Return:

- ECC_BAD_ARG_E いずれかの引数が null の場合、または inLen が偶数の場合に返されます。
- MEMORY_E 初期化中にエラーがある場合に返されます
- NOT_COMPILED_IN HAVE_COMP_KEY が true でなく、in が圧縮証明書の場合に返されます
- MP_OKAY 操作が成功しました。

Example

```
byte in[];
word32 inLen;
int curve_idx;
ecc_point* point;
wc_ecc_import_point_der(in, inLen, curve_idx, point);
```

B.23.2.34 function wc_ecc_size

```
int wc_ecc_size(
    ecc_key * key
)
```

この関数は、ecc_key 構造体のキーサイズをオクテット単位で返します。

Parameters:

- **key** キーサイズを取得する ecc_key 構造体へのポインタ

See: [wc_ecc_make_key](#)

Return:

- Given a valid key, 有効なキーが与えられた場合、キーサイズをオクテット単位で返します
- 0 指定されたキーが NULL の場合に返されます

Example

```
int keySz;
ecc_key key;
// キーを初期化し、キーを作成
keySz = wc_ecc_size(&key);
if ( keySz == 0 ) {
    // キーサイズの判定エラー
}
```

B.23.2.35 function wc_ecc_sig_size_calc

```
int wc_ecc_sig_size_calc(
    int sz
)
```

この関数は、ECC 署名の最悪の場合のサイズを返します。これは $(keySz * 2) + SIG\_HEADER\_SZ + ECC\_MAX\_PAD\_SZ$ で与えられます。実際の署名サイズは、wc_ecc_sign_hash で計算できます。

Parameters:

- **key** size キーサイズ

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size

Return: returns 最大署名サイズをオクテット単位で返します

Example

```
int sigSz = wc_ecc_sig_size_calc(32);
if ( sigSz == 0 ) {
    // 署名サイズの判定エラー
}
```

B.23.2.36 function wc_ecc_sig_size

```
int wc_ecc_sig_size(
    const ecc_key * key
)
```

この関数は、ECC 署名の最悪の場合のサイズを返します。これは $(keySz * 2) + SIG\_HEADER\_SZ + ECC\_MAX\_PAD\_SZ$ で与えられます。実際の署名サイズは、wc_ecc_sign_hash で計算できます。

Parameters:

- **key** 署名サイズを取得する ecc_key 構造体へのポインタ

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size_calc

Return:

- Success 有効なキーが与えられた場合、最大署名サイズをオクテット単位で返します
- 0 指定されたキーが NULL の場合に返されます

Example

```

int sigSz;
ecc_key key;
// キーを初期化し、キーを作成

sigSz = wc_ecc_sig_size(&key);
if ( sigSz == 0) {
    // 署名サイズの判定エラー
}

```

B.23.2.37 function wc_ecc_ctx_new

```

ecEncCtx * wc_ecc_ctx_new(
    int flags,
    WC_RNG * rng
)

```

この関数は、ECC を使用した安全なメッセージ交換を可能にするために、新しい ECC コンテキストオブジェクト用のスペースを割り当てて初期化します。

Parameters:

- **flags** これがサーバーコンテキストかクライアントコンテキストかを示します オプションは：REQ_RESP_CLIENT および REQ_RESP_SERVER
- **rng** ソルトを生成するために使用する RNG オブジェクトへのポインタ

See:

- [wc_ecc_encrypt](#)
- [wc_ecc_encrypt_ex](#)
- [wc_ecc_decrypt](#)

Return:

- Success 新しい ecEncCtx オブジェクトの生成に成功すると、そのオブジェクトへのポインタを返します
- NULL 関数が新しい ecEncCtx オブジェクトの生成に失敗した場合に返されます

Example

```

ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
if(ctx == NULL) {
    // 新しい ecEncCtx オブジェクトの生成エラー
}

```

B.23.2.38 function wc_ecc_ctx_free

```

void wc_ecc_ctx_free(
    ecEncCtx * ctx
)

```

この関数は、メッセージの暗号化と復号に使用される ecEncCtx オブジェクトを解放します。

Parameters:

- **ctx** 解放する ecEncCtx オブジェクトへのポインタ

See: [wc_ecc_ctx_new](#)

Return: none 返り値なし。

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
// 安全な通信を行う
...
wc_ecc_ctx_free(&ctx);
```

B.23.2.39 function wc_ecc_ctx_reset

```
int wc_ecc_ctx_reset(
    ecEncCtx * ctx,
    WC_RNG * rng
)
```

この関数は、新しいコンテキストオブジェクトを解放して割り当てる必要がないように、ecEncCtx 構造体をリセットします。

Parameters:

- **ctx** リセットする ecEncCtx オブジェクトへのポインタ
- **rng** 新しいソルトを生成するために使用する RNG オブジェクトへのポインタ

See: [wc_ecc_ctx_new](#)

Return:

- 0 ecEncCtx 構造体が正常にリセットされた場合に返されます
- BAD_FUNC_ARG rng または ctx のいずれかが NULL の場合に返されます
- RNG_FAILURE_E ECC オブジェクト用の新しいソルトを生成する際にエラーがある場合に返されます

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
// 安全な通信を行う
...
wc_ecc_ctx_reset(&ctx, &rng);
// さらに安全な通信を行う
```

B.23.2.40 function wc_ecc_ctx_set_algo

```
int wc_ecc_ctx_set_algo(
    ecEncCtx * ctx,
    byte encAlgo,
    byte kdfAlgo,
    byte macAlgo
)
```

この関数は、wc_ecc_ctx_new の後にオプションで呼び出すことができます。ecEncCtx オブジェクトに暗号化、KDF、および MAC アルゴリズムを設定します。

Parameters:

- **ctx** 情報を設定する ecEncCtx へのポインタ
- **encAlgo** 使用する暗号化アルゴリズム。
- **kdfAlgo** 使用する KDF アルゴリズム。
- **macAlgo** 使用する MAC アルゴリズム。

See: [wc_ecc_ctx_new](#)

Return:

- 0 ecEncCtx オブジェクトの情報を正常に設定した場合に返されます。
- BAD_FUNC_ARG 指定された ecEncCtx オブジェクトが NULL の場合に返されます。

Example

```
ecEncCtx* ctx;
// ctx を初期化
if(wc_ecc_ctx_set_algo(&ctx, ecAES_128_CTR, ecHKDF_SHA256, ecHMAC_SHA256)) {
    // 情報設定エラー
}
```

B.23.2.41 function wc_ecc_ctx_get_own_salt

```
const byte * wc_ecc_ctx_get_own_salt(
    ecEncCtx *
```

この関数は、ecEncCtx オブジェクトのソルトを返します。この関数は、ecEncCtx の状態が ecSRV_INIT または ecCLI_INIT の場合にのみ呼び出す必要があります。

Parameters:

- **ctx** ソルトを取得する ecEncCtx オブジェクトへのポインタ

See:

- [wc_ecc_ctx_new](#)
- [wc_ecc_ctx_set_peer_salt](#)
- [wc_ecc_ctx_set_kdf_salt](#)

Return:

- Success 成功時に、ecEncCtx ソルトを返します
- NULL ecEncCtx オブジェクトが NULL の場合、または ecEncCtx の状態が ecSRV_INIT または ecCLI_INIT でない場合に返されます。後者の 2 つのケースでは、この関数は ecEncCtx の状態をそれぞれ ecSRV_BAD_STATE または ecCLI_BAD_STATE に設定します

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) {
    // ソルト取得エラー
}
```

B.23.2.42 function wc_ecc_ctx_set_peer_salt

```
int wc_ecc_ctx_set_peer_salt(
    ecEncCtx * ctx,
    const byte * salt
)
```

この関数は、ecEncCtx オブジェクトのピアソルトを設定します。

Parameters:

- **ctx** ソルトを設定する ecEncCtx へのポインタ
- **salt** ピアのソルトへのポインタ

See:

- [wc_ecc_ctx_get_own_salt](#)
- [wc_ecc_ctx_set_kdf_salt](#)

Return:

- 0 ecEncCtx オブジェクトのピアソルトを正常に設定した場合に返されます。
- BAD_FUNC_ARG 指定された ecEncCtx オブジェクトが NULL または無効なプロトコルを持つ場合、または指定されたソルトが NULL の場合に返されます
- BAD_ENC_STATE_E ecEncCtx の状態が ecSRV_SALT_GET または ecCLI_SALT_GET の場合に返されます。後者の 2 つのケースでは、この関数は ecEncCtx の状態をそれぞれ ecSRV_BAD_STATE または ecCLI_BAD_STATE に設定します

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);
```

B.23.2.43 function wc_ecc_ctx_set_kdf_salt

```
int wc_ecc_ctx_set_kdf_salt(
    ecEncCtx * ctx,
    const byte * salt,
    word32 sz
)
```

この関数は、KDF で使用するソルトポインタと長さを ecEncCtx オブジェクトに設定します。

Parameters:

- **ctx** ソルトを設定する ecEncCtx へのポインタ
- **salt** ソルトバッファへのポインタ
- **sz** ソルトの長さ (バイト単位)

See:

- [wc_ecc_ctx_get_own_salt](#)

- `wc_ecc_ctx_get_peer_salt`

Return:

- 0 `ecEncCtx` オブジェクトのソルトを正常に設定した場合に返されます。
- `BAD_FUNC_ARG` 指定された `ecEncCtx` オブジェクトが `NULL` の場合、または指定されたソルトが `NULL` で長さが `NULL` でない場合に返されます。

Example

```
ecEncCtx* srvCtx;
WC_RNG rng;
byte cliSalt[] = { 固定ソルトデータ };
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);
```

B.23.2.44 function `wc_ecc_ctx_set_info`

```
int wc_ecc_ctx_set_info(
    ecEncCtx * ctx,
    const byte * info,
    int sz
)
```

この関数は、`wc_ecc_ctx_set_peer_salt` の前または後にオプションで呼び出すことができます。`ecEncCtx` オブジェクトのオプション情報を設定します。

Parameters:

- **ctx** 情報を設定する `ecEncCtx` へのポインタ
- **info** 設定する情報を含むバッファへのポインタ
- **sz** `info` バッファのサイズ

See: `wc_ecc_ctx_new`

Return:

- 0 `ecEncCtx` オブジェクトの情報を正常に設定した場合に返されます。
- `BAD_FUNC_ARG` 指定された `ecEncCtx` オブジェクトが `NULL` の場合、入力 `info` が `NULL` の場合、またはそのサイズが無効な場合に返されます

Example

```
ecEncCtx* ctx;
byte info[] = { 情報で初期化 };
// ctx を初期化、ソルトを取得、
if(wc_ecc_ctx_set_info(&ctx, info, sizeof(info))) {
    // 情報設定エラー
}
```

B.23.2.45 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
)

```

この関数は、msg から out へ指定された入力メッセージを暗号化します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。

Parameters:

- **privKey** 暗号化に使用する秘密鍵を含む ecc_key オブジェクトへのポインタ
- **pubKey** 通信したいピアの公開鍵を含む ecc_key オブジェクトへのポインタ
- **msg** 暗号化するメッセージを保持するバッファへのポインタ
- **msgSz** 暗号化するバッファのサイズ
- **out** 暗号化された暗号文を格納するバッファへのポインタ
- **outSz** out バッファで利用可能なサイズを含む word32 オブジェクトへのポインタ。メッセージの暗号化に成功すると、出力バッファに書き込まれたバイト数を保持します
- **ctx** オプション：使用する異なる暗号化アルゴリズムを指定する ecEncCtx オブジェクトへのポインタ

See:

- [wc_ecc_encrypt_ex](#)
- [wc_ecc_decrypt](#)

Return:

- 0 入力メッセージの暗号化に成功した場合に返されます
- BAD_FUNC_ARG privKey、pubKey、msg、msgSz、out、または outSz が NULL の場合、または ctx オブジェクトがサポートされていない暗号化タイプを指定している場合に返されます
- BAD_ENC_STATE_E 指定された ctx オブジェクトが暗号化に適していない状態にある場合に返されます
- BUFFER_E 提供された出力バッファが暗号化された暗号文を格納するには小さすぎる場合に返されます
- MEMORY_E 共有秘密鍵用のメモリを割り当てる際にエラーがある場合に返されます

Example

```

byte msg[] = { 暗号化するメッセージで初期化。ブロックサイズにパディングされていることを確認
    ↪ };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// cli を秘密鍵で初期化
// serv を受信した公開鍵で初期化

ecEncCtx* cliCtx, servCtx;
// cliCtx と servCtx を初期化
// ソルトを交換
ret = wc_ecc_encrypt(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx);
if(ret != 0) {
    // メッセージの暗号化エラー
}

```

B.23.2.46 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
)
```

この関数は、msg から out へ指定された入力メッセージを暗号化します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。

Parameters:

- **privKey** 暗号化に使用する秘密鍵を含む ecc_key オブジェクトへのポインタ
- **pubKey** 通信したいピアの公開鍵を含む ecc_key オブジェクトへのポインタ
- **msg** 暗号化するメッセージを保持するバッファへのポインタ
- **msgSz** 暗号化するバッファのサイズ
- **out** 暗号化された暗号文を格納するバッファへのポインタ
- **outSz** out バッファで利用可能なサイズを含む word32 オブジェクトへのポインタ。メッセージの暗号化に成功すると、出力バッファに書き込まれたバイト数を保持します
- **ctx** オプション：使用する異なる暗号化アルゴリズムを指定する ecEncCtx オブジェクトへのポインタ
- **compressed** 公開鍵フィールドを圧縮形式で出力する。

See:

- [wc_ecc_encrypt](#)
- [wc_ecc_decrypt](#)

Return:

- 0 入力メッセージの暗号化に成功した場合に返されます
- BAD_FUNC_ARG privKey、pubKey、msg、msgSz、out、または outSz が NULL の場合、または ctx オブジェクトがサポートされていない暗号化タイプを指定している場合に返されます
- BAD_ENC_STATE_E 指定された ctx オブジェクトが暗号化に適していない状態にある場合に返されます
- BUFFER_E 提供された出力バッファが暗号化された暗号文を格納するには小さすぎる場合に返されます
- MEMORY_E 共有秘密鍵用のメモリを割り当てる際にエラーがある場合に返されます

Example

```
byte msg[] = { 暗号化するメッセージで初期化。ブロックサイズにパディングされていることを確認
    ↪ };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// cli を秘密鍵で初期化
// serv を受信した公開鍵で初期化
```



```

ecEncCtx* cliCtx, servCtx;
// cliCtx と servCtx を初期化
// ソルトを交換
ret = wc_ecc_encrypt_ex(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx,
1);
if(ret != 0) {
    // メッセージの暗号化エラー
}

```

B.23.2.47 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
)

```

この関数は、msg から out へ暗号文を復号します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。

Parameters:

- **privKey** 復号に使用する秘密鍵を含む ecc_key オブジェクトへのポインタ
- **pubKey** 通信したいピアの公開鍵を含む ecc_key オブジェクトへのポインタ
- **msg** 復号する暗号文を保持するバッファへのポインタ
- **msgSz** 復号するバッファのサイズ
- **out** 復号された平文を格納するバッファへのポインタ
- **outSz** out バッファで利用可能なサイズを含む word32 オブジェクトへのポインタ。暗号文の復号に成功すると、出力バッファに書き込まれたバイト数を保持します
- **ctx** オプション：使用する異なる復号アルゴリズムを指定する ecEncCtx オブジェクトへのポインタ

See:

- [wc_ecc_encrypt](#)
- [wc_ecc_encrypt_ex](#)

Return:

- 0 入力メッセージの復号に成功した場合に返されます
- BAD_FUNC_ARG privKey、pubKey、msg、msgSz、out、または outSz が NULL の場合、または ctx オブジェクトがサポートされていない暗号化タイプを指定している場合に返されます
- BAD_ENC_STATE_E 指定された ctx オブジェクトが復号に適していない状態にある場合に返されます
- BUFFER_E 提供された出力バッファが復号された平文を格納するには小さすぎる場合に返されます
- MEMORY_E 共有秘密鍵用のメモリを割り当てる際にエラーがある場合に返されます

Example

```

byte cipher[] = { 復号する暗号文で初期化。ブロックサイズにパディングされていることを確認 };
byte plain[sizeof(cipher)];
word32 plainSz = sizeof(plain);
int ret;
ecc_key cli, serv;

```

```
// cli を秘密鍵で初期化
// serv を受信した公開鍵で初期化
ecEncCtx* cliCtx, servCtx;
// cliCtx と servCtx を初期化
// ソルトを交換
ret = wc_ecc_decrypt(&cli, &serv, cipher, sizeof(cipher),
plain, &plainSz, cliCtx);

if(ret != 0) {
    // メッセージの復号エラー
}
```

B.23.2.48 function wc_ecc_set_nonblock

```
int wc_ecc_set_nonblock(
    ecc_key * key,
    ecc_nb_ctx_t * ctx
)
```

ノンブロッキング操作のための ECC サポートを有効にします。次のビルドオプションで Single Precision (SP) 数学でサポートされています: WOLFSSL_SP_NONBLOCK WOLFSSL_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK

Parameters:

- **key** ecc_key オブジェクトへのポインタ
- **ctx** SP 用のスタックデータキャッシュを持つ ecc_nb_ctx_t 構造体へのポインタ

Return: 0 入力メッセージのコールバックコンテキストの設定に成功した場合に返されます

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, // mp_int としての r/s
                hash, hashSz, // 計算されたハッシュダイジェスト
                &verify_res, // 検証結果 1= 成功
                &key
            );

            // TODO: リアルタイム作業をここで呼び出すことができます
        } while (ret == FP_WOULDBLOCK);
    }
    wc_ecc_free(&key);
}
```

B.23.2.49 function wc_ecc_set_curve

```
int wc_ecc_set_curve(
    ecc_key * key,
```

```

    int keysize,
    int curve_id
)

```

指定されたサイズより大きいキーを持つカーブまたはカーブ ID に一致するカーブを比較し、より小さいキーサイズを持つカーブをキーに設定します。

Parameters:

- **keysize** キーサイズ (バイト単位)
- **curve_id** カーブ 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 キーの設定に成功した場合に返されます

B.24 アルゴリズム - ED25519

B.24.1 Functions

	Name
int	wc_ed25519_make_public (ed25519_key * key, unsigned char * pubKey, word32 pubKeySz) この関数は、ed25519_key オブジェクトに格納された秘密鍵から Ed25519 公開鍵を生成します。公開鍵をバッファ pubKey に格納します。
int	wc_ed25519_make_key (WC_RNG * rng, int keysize, ed25519_key * key) この関数は新しい Ed25519 キーを生成し、それを key に格納します。
int	wc_ed25519_sign_msg (const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key) この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージに署名します。
int	wc_ed25519ctx_sign_msg (const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部です。

	Name
int	wc_ed25519ph_sign_hash (const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed25519_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されます。
int	wc_ed25519ph_sign_msg (const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されます。
int	wc_ed25519_verify_msg (const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key) この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
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) この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
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) この関数は、真正性を保証するためにメッセージのダイジェストの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。メッセージダイジェストの作成に使用されるハッシュアルゴリズムは SHA-512 である必要があります。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

	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) この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。メッセージは検証前に事前ハッシュ化されます。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
int	wc_ed25519_init (ed25519_key * key) この関数は、メッセージ検証での将来の使用のために ed25519_key オブジェクトを初期化します。
void	wc_ed25519_free (ed25519_key * key) この関数は、使用後に Ed25519 オブジェクトを解放します。
int	wc_ed25519_import_public (const byte * in, word32 inLen, ed25519_key * key) この関数は、公開鍵を含むバッファから公開 ed25519_key をインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は、秘密鍵が存在する場合にそれと一致するかチェックされます。
int	wc_ed25519_import_public_ex (const byte * in, word32 inLen, ed25519_key * key, int trusted) この関数は、公開鍵を含むバッファから公開 ed25519_key をインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、秘密鍵が存在するときに公開鍵が秘密鍵と一致するかチェックします。
int	wc_ed25519_import_private_only (const byte * priv, word32 privSz, ed25519_key * key) この関数は、バッファから Ed25519 秘密鍵のみをインポートします。
int	wc_ed25519_import_private_key (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key) この関数は、一対のバッファから公開/秘密 Ed25519 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は信頼されていないと想定され、秘密鍵に対してチェックされます。
int	wc_ed25519_import_private_key_ex (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key, int trusted) この関数は、一対のバッファから公開/秘密 Ed25519 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、公開鍵が秘密鍵に対してチェックされます。

	Name
int	wc_ed25519_export_public (const ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体から公開鍵をエクスポートします。公開鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ed25519_export_private_only (const ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ed25519_export_private (const ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ed25519_export_key (const ed25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、ed25519_key 構造体から秘密鍵と公開鍵を個別にエクスポートします。秘密鍵をバッファ priv に格納し、このバッファに書き込まれたバイト数を privSz に設定します。公開鍵をバッファ pub に格納し、このバッファに書き込まれたバイト数を pubSz に設定します。
int	wc_ed25519_check_key (ed25519_key * key) この関数は、ed25519_key 構造体内の公開鍵が秘密鍵と一致するかチェックします。
int	wc_ed25519_size (const ed25519_key * key) この関数は、Ed25519 のサイズ (32 バイト) を返します。
int	wc_ed25519_priv_size (const ed25519_key * key) この関数は、秘密鍵のサイズ (秘密 + 公開) をバイト単位で返します。
int	wc_ed25519_pub_size (const ed25519_key * key) この関数は、圧縮鍵のサイズをバイト単位で返します (公開鍵)。
int	wc_ed25519_sig_size (const ed25519_key * key) この関数は、Ed25519 署名のサイズ (バイト単位で 64) を返します。

B.24.2 Functions Documentation

B.24.2.1 function wc_ed25519_make_public

```
int wc_ed25519_make_public(
    ed25519_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

この関数は、ed25519_key オブジェクトに格納された秘密鍵から Ed25519 公開鍵を生成します。公開鍵をバッファ pubKey に格納します。

Parameters:

- **key** キーを生成する ed25519_key へのポインタ。
- **pubKey** 公開鍵を格納するバッファへのポインタ。
- **pubKeySz** 公開鍵のサイズ。ED25519_PUB_KEY_SIZE である必要があります。

See:

- [wc_ed25519_init](#)
- [wc_ed25519_import_private_only](#)
- [wc_ed25519_make_key](#)

Return:

- 0 公開鍵の作成に成功した場合に返されます。
- BAD_FUNC_ARG key または pubKey が NULL と評価された場合、または指定されたキーサイズが 32 バイトでない場合に返されます (Ed25519 は 32 バイトのキーを持ちます)。
- ECC_PRIV_KEY_E ed25519_key オブジェクトに秘密鍵が含まれていない場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
int ret;

ed25519_key key;
byte priv[] = { 32バイトの秘密鍵で初期化 };
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) {
    // 公開鍵の作成エラー
}
```

B.24.2.2 function wc_ed25519_make_key

```
int wc_ed25519_make_key(
    WC_RNG * rng,
    int keysz,
    ed25519_key * key
)
```

この関数は新しい Ed25519 キーを生成し、それを key に格納します。

Parameters:

- **rng** キーを生成するために使用する初期化済み RNG オブジェクトへのポインタ。
- **keysize** 生成するキーの長さ。Ed25519 の場合は常に 32 である必要があります。
- **key** キーを生成する ed25519_key へのポインタ。

See: [wc_ed25519_init](#)

Return:

- 0 ed25519_key の作成に成功した場合に返されます。

- BAD_FUNC_ARG rng または key が NULL と評価された場合、または指定されたキーサイズが 32 バイトでない場合に返されます (Ed25519 は 32 バイトのキーを持ちます)。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

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) {
    // キー作成エラー
}
```

B.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
)
```

この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージに署名します。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージの長さ。
- **out** 生成された署名を格納するバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 ed25519_key へのポインタ。

See:

- [wc_ed25519ctx_sign_msg](#)
- [wc_ed25519ph_sign_hash](#)
- [wc_ed25519ph_sign_msg](#)
- [wc_ed25519_verify_msg](#)

Return:

- 0 メッセージの署名の生成に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // 生成された署名を保持
```



```

sigSz = sizeof(sig);
byte message[] = { メッセージで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed25519_init(&key); // key を初期化
wc_ed25519_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed25519_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0) {
    // メッセージ署名の生成エラー
}

```

B.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
)

```

この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部です。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージの長さ。
- **out** 生成された署名を格納するバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 ed25519_key へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed25519_sign_msg](#)
- [wc_ed25519ph_sign_hash](#)
- [wc_ed25519ph_sign_msg](#)
- [wc_ed25519_verify_msg](#)

Return:

- 0 メッセージの署名の生成に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```

ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // 生成された署名を保持
sigSz = sizeof(sig);

```

```

byte message[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed25519_init(&key); // key を初期化
wc_ed25519_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed25519ctx_sign_msg(message, sizeof(message), sig, &sigSz, &key,
    context, sizeof(context));
if (ret != 0) {
    // メッセージ署名の生成エラー
}

```

B.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
)

```

この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されます。

Parameters:

- **hash** 署名するメッセージのハッシュを含むバッファへのポインタ。
- **hashLen** 署名するメッセージのハッシュの長さ。
- **out** 生成された署名を格納するバッファ。
- **outLen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 ed25519_key へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed25519_sign_msg](#)
- [wc_ed25519ctx_sign_msg](#)
- [wc_ed25519ph_sign_msg](#)
- [wc_ed25519_verify_msg](#)

Return:

- 0 メッセージダイジェストの署名の生成に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```

ed25519_key key;
WC_RNG rng;
int ret, sigSz;

```

```

byte sig[64]; // 生成された署名を保持
sigSz = sizeof(sig);
byte hash[] = { メッセージの SHA-512ハッシュで初期化 };
byte context[] = { 署名のコンテキストで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed25519_init(&key); // key を初期化
wc_ed25519_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed25519ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
    context, sizeof(context));
if (ret != 0) {
    // メッセージ署名の生成エラー
}

```

B.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
)

```

この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されます。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージの長さ。
- **out** 生成された署名を格納するバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 ed25519_key へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed25519_sign_msg](#)
- [wc_ed25519ctx_sign_msg](#)
- [wc_ed25519ph_sign_hash](#)
- [wc_ed25519_verify_msg](#)

Return:

- 0 メッセージの署名の生成に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```

ed25519_key key;
WC_RNG rng;

```

```

int ret, sigSz;

byte sig[64]; // 生成された署名を保持
sigSz = sizeof(sig);
byte message[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed25519_init(&key); // key を初期化
wc_ed25519_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed25519ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
                             context, sizeof(context));
if (ret != 0) {
    // メッセージ署名の生成エラー
}

```

B.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
)

```

この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **msg** 検証するメッセージを含むバッファへのポインタ。
- **msgLen** 検証するメッセージの長さ。
- **ret** 検証結果へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するために使用する公開 Ed25519 鍵へのポインタ。

See:

- [wc_ed25519ctx_verify_msg](#)
- [wc_ed25519ph_verify_hash](#)
- [wc_ed25519ph_verify_msg](#)
- [wc_ed25519_sign_msg](#)

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さとは一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```

ed25519_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };

```

```

byte msg[] = { メッセージで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed25519_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
    &key);
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}

```

B.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
)

```

この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **msg** 検証するメッセージを含むバッファへのポインタ。
- **msgLen** 検証するメッセージの長さ。
- **ret** 検証結果へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するために使用する公開 Ed25519 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed25519_verify_msg](#)
- [wc_ed25519ph_verify_hash](#)
- [wc_ed25519ph_verify_msg](#)
- [wc_ed25519_sign_msg](#)

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```

ed25519_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };
byte msg[] = { メッセージで初期化 };

```

```

byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed25519ctx_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
    &verified, &key, );
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}

```

B.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
)

```

この関数は、真正性を保証するためにメッセージのダイジェストの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。メッセージダイジェストの作成に使用されるハッシュアルゴリズムは SHA-512 である必要があります。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **hash** 検証するメッセージのハッシュを含むバッファへのポインタ。
- **hashLen** 検証するハッシュの長さ。
- **ret** 検証結果へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するために使用する公開 Ed25519 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed25519_verify_msg](#)
- [wc_ed25519ctx_verify_msg](#)
- [wc_ed25519ph_verify_msg](#)
- [wc_ed25519_sign_msg](#)

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```

ed25519_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };

```

```

byte hash[] = { メッセージの SHA-512ハッシュで初期化 };
byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed25519ph_verify_hash(sig, sizeof(sig), hash, sizeof(hash),
    &verified, &key, context, sizeof(context));
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}

```

B.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
)

```

この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。メッセージは検証前に事前ハッシュ化されます。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **msg** 検証するメッセージを含むバッファへのポインタ。
- **msgLen** 検証するメッセージの長さ。
- **ret** 検証結果へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するために使用する公開 Ed25519 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed25519_verify_msg](#)
- [wc_ed25519ctx_verify_msg](#)
- [wc_ed25519ph_verify_hash](#)
- [wc_ed25519_sign_msg](#)

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```

ed25519_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };

```



```

byte msg[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed25519ph_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
    &verified, &key, context, sizeof(context));
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}

```

B.24.2.11 function wc_ed25519_init

```

int wc_ed25519_init(
    ed25519_key * key
)

```

この関数は、メッセージ検証での将来の使用のために ed25519_key オブジェクトを初期化します。

Parameters:

- **key** 初期化する ed25519_key オブジェクトへのポインタ。

See:

- [wc_ed25519_make_key](#)
- [wc_ed25519_free](#)

Return:

- 0 ed25519_key オブジェクトの初期化に成功した場合に返されます。
- BAD_FUNC_ARG key が NULL の場合に返されます。

Example

```

ed25519_key key;
wc_ed25519_init(&key);

```

B.24.2.12 function wc_ed25519_free

```

void wc_ed25519_free(
    ed25519_key * key
)

```

この関数は、使用後に Ed25519 オブジェクトを解放します。

Parameters:

- **key** 解放する ed25519_key オブジェクトへのポインタ

See: [wc_ed25519_init](#)

Example

```

ed25519_key key;
// key を初期化し、安全な交換を実行
...
wc_ed25519_free(&key);

```


B.24.2.13 function wc_ed25519_import_public

```
int wc_ed25519_import_public(
    const byte * in,
    word32 inLen,
    ed25519_key * key
)
```

この関数は、公開鍵を含むバッファから公開 ed25519_key をインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は、秘密鍵が存在する場合にそれと一致するかチェックされます。

Parameters:

- **in** 公開鍵を含むバッファへのポインタ。
- **inLen** 公開鍵を含むバッファの長さ。
- **key** 公開鍵を格納する ed25519_key オブジェクトへのポインタ。

See:

- [wc_ed25519_import_public_ex](#)
- [wc_ed25519_import_private_key](#)
- [wc_ed25519_import_private_key_ex](#)
- [wc_ed25519_export_public](#)

Return:

- 0 ed25519_key のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または inLen が Ed25519 鍵のサイズより小さい場合に返されます。

Example

```
int ret;
byte pub[] = { Ed25519 公開鍵で初期化 };

ed_25519 key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.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
)
```

この関数は、公開鍵を含むバッファから公開 ed25519_key をインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、秘密鍵が存在するときに公開鍵が秘密鍵と一致するかチェックします。

Parameters:

- **in** 公開鍵を含むバッファへのポインタ。
- **inLen** 公開鍵を含むバッファの長さ。
- **key** 公開鍵を格納する ed25519_key オブジェクトへのポインタ。
- **trusted** 公開鍵データが信頼されているかどうか。

See:

- [wc_ed25519_import_public](#)
- [wc_ed25519_import_private_key](#)
- [wc_ed25519_import_private_key_ex](#)
- [wc_ed25519_export_public](#)

Return:

- 0 ed25519_key のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または inLen が Ed25519 鍵のサイズより小さい場合に返されます。

Example

```
int ret;
byte pub[] = { Ed25519 公開鍵で初期化 };

ed_25519 key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.24.2.15 function wc_ed25519_import_private_only

```
int wc_ed25519_import_private_only(
    const byte * priv,
    word32 privSz,
    ed25519_key * key
)
```

この関数は、バッファから Ed25519 秘密鍵のみをインポートします。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **key** インポートされた秘密鍵を格納する ed25519_key オブジェクトへのポインタ。

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 Ed25519 鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG priv または key が NULL と評価された場合、または privSz が ED25519_KEY_SIZE と等しくない場合に返されます。

Example

```
int ret;
byte priv[] = { 32バイトの秘密鍵で初期化 };

ed25519_key key;
```

```

wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // 秘密鍵のインポートエラー
}

```

B.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
)

```

この関数は、一対のバッファから公開/秘密 Ed25519 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は信頼されていないと想定され、秘密鍵に対してチェックされます。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **pub** 公開鍵を含むバッファへのポインタ。
- **pubSz** 公開鍵の長さ。
- **key** インポートされた秘密/公開鍵ペアを格納する ed25519_key オブジェクトへのポインタ。

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 ed25519_key のインポートに成功した場合に返されます。
- BAD_FUNC_ARG priv または key が NULL と評価された場合、または privSz が ED25519_KEY_SIZE にも ED25519_PRV_KEY_SIZE にも等しくない、もしくは pubSz が ED25519_PUB_KEY_SIZE より小さい場合に返されます。

Example

```

int ret;
byte priv[] = { 32バイトの秘密鍵で初期化 };
byte pub[] = { 対応する公開鍵で初期化 };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
    &key);
if (ret != 0) {
    // 鍵のインポートエラー
}

```

B.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
)
```

この関数は、一対のバッファから公開/秘密 Ed25519 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、公開鍵が秘密鍵に対してチェックされます。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **pub** 公開鍵を含むバッファへのポインタ。
- **pubSz** 公開鍵の長さ。
- **key** インポートされた秘密/公開鍵ペアを格納する ed25519_key オブジェクトへのポインタ。
- **trusted** 公開鍵データが信頼されているかどうか。

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 ed25519_key のインポートに成功した場合に返されます。
- BAD_FUNC_ARG priv または key が NULL と評価された場合、または privSz が ED25519_KEY_SIZE にも ED25519_PRV_KEY_SIZE にも等しくない、もしくは pubSz が ED25519_PUB_KEY_SIZE より小さい場合に返されます。

Example

```
int ret;
byte priv[] = { 32バイトの秘密鍵で初期化 };
byte pub[] = { 対応する公開鍵で初期化 };

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) {
    // 鍵のインポートエラー
}
```

B.24.2.18 function wc_ed25519_export_public

```
int wc_ed25519_export_public(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed25519_key 構造体から公開鍵をエクスポートします。公開鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 公開鍵をエクスポートする ed25519_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。公開鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed25519_import_public](#)
- [wc_ed25519_import_public_ex](#)
- [wc_ed25519_export_private](#)
- [wc_ed25519_export_private_only](#)

Return:

- 0 公開鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが公開鍵を格納するのに十分な大きさでない場合に返されます。このエラーを返す際、関数は outLen に必要なサイズを設定します。

Example

```
int ret;
ed25519_key key;
// key を初期化し、key を作成

char pub[32];
word32 pubSz = sizeof(pub);

ret = wc_ed25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // 公開鍵のエクスポートエラー
}
```

B.24.2.19 function wc_ed25519_export_private_only

```
int wc_ed25519_export_private_only(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed25519_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 秘密鍵をエクスポートする ed25519_key 構造体へのポインタ。
- **out** 秘密鍵を格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。秘密鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed25519_export_public](#)
- [wc_ed25519_export_private](#)
- [wc_ed25519_import_private_key](#)

- [wc_ed25519_import_private_key_ex](#)

Return:

- 0 秘密鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが秘密鍵を格納するのに十分な大きさでない場合に返されます。

Example

```
int ret;
ed25519_key key;
// key を初期化し、key を作成

char priv[32]; // 秘密鍵のみなので 32 バイト
word32 privSz = sizeof(priv);
ret = wc_ed25519_export_private_only(&key, priv, &privSz);
if (ret != 0) {
    // 秘密鍵のエクスポートエラー
}
```

B.24.2.20 function wc_ed25519_export_private

```
int wc_ed25519_export_private(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed25519_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 鍵ペアをエクスポートする ed25519_key 構造体へのポインタ。
- **out** 鍵ペアを格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。鍵ペアのエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed25519_import_private_key](#)
- [wc_ed25519_import_private_key_ex](#)
- [wc_ed25519_export_private_only](#)

Return:

- 0 鍵ペアのエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが鍵ペアを格納するのに十分な大きさでない場合に返されます。

Example

```
ed25519_key key;
wc_ed25519_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // 32 バイトの Ed25519 鍵を初期化
```

```

byte out[64]; // out は十分なバッファサイズである必要があります
word32 outLen = sizeof(out);
int key_size = wc_ed25519_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // 関数が outLen をリセットしたかどうかを確認するため、out のサイズと outLen を比較
}

```

B.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
)

```

この関数は、ed25519_key 構造体から秘密鍵と公開鍵を個別にエクスポートします。秘密鍵をバッファ priv に格納し、このバッファに書き込まれたバイト数を privSz に設定します。公開鍵をバッファ pub に格納し、このバッファに書き込まれたバイト数を pubSz に設定します。

Parameters:

- **key** 鍵ペアをエクスポートする ed25519_key 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。秘密鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。公開鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed25519_export_private](#)
- [wc_ed25519_export_public](#)

Return:

- 0 鍵ペアのエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが鍵ペアを格納するのに十分な大きさでない場合に返されます。

Example

```

int ret;
ed25519_key key;
// key を初期化し、key を作成

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) {
    // 公開鍵のエクスポートエラー
}

```

B.24.2.22 function wc_ed25519_check_key

```
int wc_ed25519_check_key(
    ed25519_key * key
)
```

この関数は、ed25519_key 構造体内の公開鍵が秘密鍵と一致するかチェックします。

Parameters:

- **key** 秘密鍵と公開鍵を保持する ed25519_key 構造体へのポインタ。

See:

- [wc_ed25519_import_private_key](#)
- [wc_ed25519_import_private_key_ex](#)

Return:

- 0 秘密鍵と公開鍵が一致した場合に返されます。
- BAD_FUNC_ARG 指定された key が NULL の場合に返されます。
- PUBLIC_KEY_E 公開鍵が利用できない、または無効な場合に返されます。

Example

```
int ret;
byte priv[] = { 57バイトの秘密鍵で初期化 };
byte pub[] = { 対応する公開鍵で初期化 };

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) {
    // 鍵のチェックエラー
}
```

B.24.2.23 function wc_ed25519_size

```
int wc_ed25519_size(
    const ed25519_key * key
)
```

この関数は、Ed25519 のサイズ（32 バイト）を返します。

Parameters:

- **key** 鍵サイズを取得する ed25519_key 構造体へのポインタ。

See: [wc_ed25519_make_key](#)**Return:**

- ED25519_KEY_SIZE 有効な秘密鍵のサイズ（32 バイト）。
- BAD_FUNC_ARG 指定された key が NULL の場合に返されます。

Example

```
int keySz;
ed25519_key key;
// key を初期化し、key を作成
keySz = wc_ed25519_size(&key);
```



```
if (keySz == 0) {  
    // 鍵サイズの決定エラー  
}
```

B.24.2.24 function wc_ed25519_priv_size

```
int wc_ed25519_priv_size(  
    const ed25519_key * key  
)
```

この関数は、秘密鍵のサイズ（秘密 + 公開）をバイト単位で返します。

Parameters:

- **key** 鍵サイズを取得する ed25519_key 構造体へのポインタ。

See: [wc_ed25519_pub_size](#)

Return:

- ED25519_PRIV_KEY_SIZE 秘密鍵のサイズ（64 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```
ed25519_key key;  
wc_ed25519_init(&key);  
  
WC_RNG rng;  
wc_InitRng(&rng);  
  
wc_ed25519_make_key(&rng, 32, &key); // 32 バイトの Ed25519 鍵を初期化  
int key_size = wc_ed25519_priv_size(&key);
```

B.24.2.25 function wc_ed25519_pub_size

```
int wc_ed25519_pub_size(  
    const ed25519_key * key  
)
```

この関数は、圧縮鍵のサイズをバイト単位で返します（公開鍵）。

Parameters:

- **key** 鍵サイズを取得する ed25519_key 構造体へのポインタ。

See: [wc_ed25519_priv_size](#)

Return:

- ED25519_PUB_KEY_SIZE 圧縮公開鍵のサイズ（32 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```
ed25519_key key;  
wc_ed25519_init(&key);  
WC_RNG rng;  
wc_InitRng(&rng);  
  
wc_ed25519_make_key(&rng, 32, &key); // 32 バイトの Ed25519 鍵を初期化  
int key_size = wc_ed25519_pub_size(&key);
```

B.24.2.26 function wc_ed25519_sig_size

```
int wc_ed25519_sig_size(
    const ed25519_key * key
)
```

この関数は、Ed25519 署名のサイズ（バイト単位で 64）を返します。

Parameters:

- **key** 署名サイズを取得する ed25519_key 構造体へのポインタ。

See: [wc_ed25519_sign_msg](#)

Return:

- ED25519_SIG_SIZE Ed25519 署名のサイズ（64 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```
int sigSz;
ed25519_key key;
// key を初期化し、key を作成

sigSz = wc_ed25519_sig_size(&key);
if (sigSz == 0) {
    // 署名サイズの決定エラー
}
```

B.25 アルゴリズム - ED448**B.25.1 Functions**

	Name
int	wc_ed448_make_public (ed448_key * key, unsigned char * pubKey, word32 pubKeySz) この関数は、秘密鍵から Ed448 公開鍵を生成します。公開鍵をバッファ pubKey に格納し、このバッファに書き込まれたバイト数を pubKeySz に設定します。
int	wc_ed448_make_key (WC_RNG * rng, int keysize, ed448_key * key) この関数は新しい Ed448 キーを生成し、それを key に格納します。
int	wc_ed448_sign_msg (const byte * in, word32 inLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed448_key オブジェクトを使用してメッセージに署名します。

	Name
int	wc_ed448ph_sign_hash (const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed448_key オブジェクトを使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。
int	wc_ed448ph_sign_msg (const byte * in, word32 inlen, byte * out, word32 * outlen, ed448_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed448_key オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されます。
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) この関数は、真正性を保証するためにメッセージの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。答えは res を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
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) この関数は、真正性を保証するためにメッセージのダイジェストの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。答えは res を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
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) この関数は、真正性を保証するためにメッセージの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。メッセージは検証前に事前ハッシュ化されます。答えは res を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
int	wc_ed448_init (ed448_key * key) この関数は、メッセージ検証での将来の使用のために ed448_key オブジェクトを初期化します。
void	wc_ed448_free (ed448_key * key) この関数は、使用後に Ed448 オブジェクトを解放します。

	Name
int	wc_ed448_import_public (const byte * in, word32 inLen, ed448_key * key) この関数は、公開鍵を含むバッファから公開 ed448_key ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は、秘密鍵が存在する場合にそれと一致するかチェックされます。
int	wc_ed448_import_public_ex (const byte * in, word32 inLen, ed448_key * key, int trusted) この関数は、公開鍵を含むバッファから公開 ed448_key ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、秘密鍵が存在するときに公開鍵が秘密鍵と一致するかチェックします。
int	wc_ed448_import_private_only (const byte * priv, word32 privSz, ed448_key * key) この関数は、バッファから Ed448 秘密鍵のみをインポートします。
int	wc_ed448_import_private_key (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key) この関数は、一対のバッファから公開/秘密 Ed448 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。
int	wc_ed448_import_private_key_ex (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key, int trusted) この関数は、一対のバッファから公開/秘密 Ed448 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、公開鍵が秘密鍵に対してチェックされます。
int	wc_ed448_export_public (const ed448_key * key, byte * out, word32 * outLen) この関数は、ed448_key 構造体から秘密鍵をエクスポートします。公開鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ed448_export_private_only (const ed448_key * key, byte * out, word32 * outLen) この関数は、ed448_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ed448_export_private (const ed448_key * key, byte * out, word32 * outLen) この関数は、ed448_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

	Name
int	wc_ed448_export_key (const ed448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、ed448_key 構造体から秘密鍵と公開鍵を個別にエクスポートします。秘密鍵をバッファ priv に格納し、このバッファに書き込まれたバイト数を privSz に設定します。公開鍵をバッファ pub に格納し、このバッファに書き込まれたバイト数を pubSz に設定します。
int	wc_ed448_check_key (ed448_key * key) この関数は、ed448_key 構造体内の公開鍵が秘密鍵と一致するかチェックします。
int	wc_ed448_size (const ed448_key * key) この関数は、Ed448 秘密鍵のサイズ (57 バイト) を返します。
int	wc_ed448_priv_size (const ed448_key * key) この関数は、秘密鍵のサイズ (秘密 + 公開) をバイト単位で返します。
int	wc_ed448_pub_size (const ed448_key * key) この関数は、圧縮鍵のサイズをバイト単位で返します (公開鍵)。
int	wc_ed448_sig_size (const ed448_key * key) この関数は、Ed448 署名のサイズ (バイト単位で 114) を返します。

B.25.2 Functions Documentation

B.25.2.1 function wc_ed448_make_public

```
int wc_ed448_make_public(
    ed448_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

この関数は、秘密鍵から Ed448 公開鍵を生成します。公開鍵をバッファ pubKey に格納し、このバッファに書き込まれたバイト数を pubKeySz に設定します。

Parameters:

- **key** キーを生成する ed448_key へのポインタ。
- **pubKey** 公開鍵を格納するバッファへのポインタ。
- **pubKeySz** pubKey バッファのサイズ (バイト単位)。

See:

- [wc_ed448_init](#)
- [wc_ed448_import_private_only](#)
- [wc_ed448_make_key](#)

Return:

- 0 公開鍵の作成に成功した場合に返されます。
- BAD_FUNC_ARG key または pubKey が NULL と評価された場合、または指定されたキーサイズが 57 バイトでない場合に返されます (Ed448 は 57 バイトのキーを持ちます)。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
int ret;

ed448_key key;
byte priv[] = { 57バイトの秘密鍵で初期化 };
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) {
    // 公開鍵の作成エラー
}
```

B.25.2.2 function wc_ed448_make_key

```
int wc_ed448_make_key(
    WC_RNG * rng,
    int keysize,
    ed448_key * key
)
```

この関数は新しい Ed448 キーを生成し、それを key に格納します。

Parameters:

- **rng** キーを生成するために使用する初期化済み RNG オブジェクトへのポインタ。
- **keysize** 生成するキーの長さ。Ed448 の場合は常に 57 である必要があります。
- **key** キーを生成する ed448_key へのポインタ。

See: [wc_ed448_init](#)

Return:

- 0 ed448_key の作成に成功した場合に返されます。
- BAD_FUNC_ARG rng または key が NULL と評価された場合、または指定されたキーサイズが 57 バイトでない場合に返されます (Ed448 は 57 バイトのキーを持ちます)。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

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) {
    // キー作成エラー
}
```

B.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
)

```

この関数は、真正性を保証するために ed448_key オブジェクトを使用してメッセージに署名します。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inLen** 署名するメッセージの長さ。
- **out** 生成された署名を格納するバッファ。
- **outLen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 ed448_key へのポインタ。
- **context** メッセージが署名されるコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed448ph_sign_hash](#)
- [wc_ed448ph_sign_msg](#)
- [wc_ed448_verify_msg](#)

Return:

- 0 メッセージの署名の生成に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```

ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // 生成された署名を保持
sigSz = sizeof(sig);
byte message[] = { メッセージで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed448_init(&key); // key を初期化
wc_ed448_make_key(&rng, 57, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed448_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0) {
    // メッセージ署名の生成エラー
}

```

B.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
)

```

この関数は、真正性を保証するために ed448_key オブジェクトを使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。

Parameters:

- **hash** 署名するメッセージのハッシュを含むバッファへのポインタ。
- **hashLen** 署名するメッセージのハッシュの長さ。
- **out** 生成された署名を格納するバッファ。
- **outLen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 ed448_key へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed448_sign_msg](#)
- [wc_ed448ph_sign_msg](#)
- [wc_ed448ph_verify_hash](#)

Return:

- 0 メッセージダイジェストの署名の生成に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```

ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // 生成された署名を保持
sigSz = sizeof(sig);
byte hash[] = { メッセージのハッシュで初期化 };
byte context[] = { 署名のコンテキストで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed448_init(&key); // key を初期化
wc_ed448_make_key(&rng, 57, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed448ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
    context, sizeof(context));
if (ret != 0) {
    // メッセージ署名の生成エラー
}

```

B.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
)

```

この関数は、真正性を保証するために ed448_key オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されません。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inLen** 署名するメッセージの長さ。
- **out** 生成された署名を格納するバッファ。
- **outLen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 ed448_key へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed448_sign_msg](#)
- [wc_ed448ph_sign_hash](#)
- [wc_ed448ph_verify_msg](#)

Return:

- 0 メッセージの署名の生成に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```

ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // 生成された署名を保持
sigSz = sizeof(sig);
byte message[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed448_init(&key); // key を初期化
wc_ed448_make_key(&rng, 57, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed448ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
    context, sizeof(context));
if (ret != 0) {
    // メッセージ署名の生成エラー
}

```

B.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
)

```

この関数は、真正性を保証するためにメッセージの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。答えは res を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **msg** 検証するメッセージを含むバッファへのポインタ。
- **msgLen** 検証するメッセージの長さ。
- **res** 検証完了後、有効な署名の場合は 1、無効な署名の場合は 0 が設定される int 型へのポインタ。
- **key** 署名を検証するために使用する公開 Ed448 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed448ph_verify_hash](#)
- [wc_ed448ph_verify_msg](#)
- [wc_ed448_sign_msg](#)

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さとは一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```

ed448_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };
byte msg[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed448_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
    &key, context, sizeof(context));
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}

```

B.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
)

```

この関数は、真正性を保証するためにメッセージのダイジェストの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。答えは res を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **hash** 検証するメッセージのハッシュを含むバッファへのポインタ。
- **hashlen** 検証するハッシュの長さ。
- **res** 検証完了後、有効な署名の場合は 1、無効な署名の場合は 0 が設定される int 型へのポインタ。
- **key** 署名を検証するために使用する公開 Ed448 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed448_verify_msg](#)
- [wc_ed448ph_verify_msg](#)
- [wc_ed448ph_sign_hash](#)

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さとは一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```

ed448_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };
byte hash[] = { メッセージのハッシュで初期化 };
byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed448ph_verify_hash(sig, sizeof(sig), hash, sizeof(hash),
    &verified, &key, context, sizeof(context));
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}

```

B.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
)

```

この関数は、真正性を保証するためにメッセージの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。メッセージは検証前に事前ハッシュ化されます。答えは res を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **msg** 検証するメッセージを含むバッファへのポインタ。
- **msgLen** 検証するメッセージの長さ。
- **res** 検証完了後、有効な署名の場合は 1、無効な署名の場合は 0 が設定される int 型へのポインタ。
- **key** 署名を検証するために使用する公開 Ed448 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed448_verify_msg](#)
- [wc_ed448ph_verify_hash](#)
- [wc_ed448ph_sign_msg](#)

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```

ed448_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };
byte msg[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed448ph_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
    &key, context, sizeof(context));
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}

```

B.25.2.9 function wc_ed448_init

```

int wc_ed448_init(
    ed448_key * key
)

```

この関数は、メッセージ検証での将来の使用のために ed448_key オブジェクトを初期化します。

Parameters:

- **key** 初期化する ed448_key オブジェクトへのポインタ。

See:

- `wc_ed448_make_key`
- `wc_ed448_free`

Return:

- 0 ed448_key オブジェクトの初期化に成功した場合に返されます。
- BAD_FUNC_ARG key が NULL の場合に返されます。

Example

```
ed448_key key;  
wc_ed448_init(&key);
```

B.25.2.10 function wc_ed448_free

```
void wc_ed448_free(  
    ed448_key * key  
)
```

この関数は、使用後に Ed448 オブジェクトを解放します。

Parameters:

- **key** 解放する ed448_key オブジェクトへのポインタ

See: `wc_ed448_init`

Example

```
ed448_key key;  
// key を初期化し、安全な交換を実行  
...  
wc_ed448_free(&key);
```

B.25.2.11 function wc_ed448_import_public

```
int wc_ed448_import_public(  
    const byte * in,  
    word32 inLen,  
    ed448_key * key  
)
```

この関数は、公開鍵を含むバッファから公開 ed448_key ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は、秘密鍵が存在する場合にそれと一致するかチェックされます。

Parameters:

- **in** 公開鍵を含むバッファへのポインタ。
- **inLen** 公開鍵を含むバッファの長さ。
- **key** 公開鍵を格納する ed448_key オブジェクトへのポインタ。

See:

- `wc_ed448_import_public_ex`
- `wc_ed448_import_private_key`
- `wc_ed448_import_private_key_ex`
- `wc_ed448_export_public`

Return:

- 0 ed448_key のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または inLen が Ed448 鍵のサイズより小さい場合に返されます。

Example

```
int ret;
byte pub[] = { Ed448 公開鍵で初期化 };

ed_448 key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.25.2.12 function wc_ed448_import_public_ex

```
int wc_ed448_import_public_ex(
    const byte * in,
    word32 inLen,
    ed448_key * key,
    int trusted
)
```

この関数は、公開鍵を含むバッファから公開 ed448_key ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、秘密鍵が存在するときに公開鍵が秘密鍵と一致するかチェックします。

Parameters:

- **in** 公開鍵を含むバッファへのポインタ。
- **inLen** 公開鍵を含むバッファの長さ。
- **key** 公開鍵を格納する ed448_key オブジェクトへのポインタ。
- **trusted** 公開鍵データが信頼されているかどうか。

See:

- [wc_ed448_import_public](#)
- [wc_ed448_import_private_key](#)
- [wc_ed448_import_private_key_ex](#)
- [wc_ed448_export_public](#)

Return:

- 0 ed448_key のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または inLen が Ed448 鍵のサイズより小さい場合に返されます。

Example

```
int ret;
byte pub[] = { Ed448 公開鍵で初期化 };

ed_448 key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.25.2.13 function wc_ed448_import_private_only

```
int wc_ed448_import_private_only(
    const byte * priv,
    word32 privSz,
    ed448_key * key
)
```

この関数は、バッファから Ed448 秘密鍵のみをインポートします。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **key** インポートされた秘密鍵を格納する ed448_key オブジェクトへのポインタ。

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 Ed448 秘密鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または privSz が ED448_KEY_SIZE より小さい場合に返されます。

Example

```
int ret;
byte priv[] = { 57バイトの秘密鍵で初期化 };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // 秘密鍵のインポートエラー
}
```

B.25.2.14 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
)
```

この関数は、一対のバッファから公開/秘密 Ed448 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **pub** 公開鍵を含むバッファへのポインタ。
- **pubSz** 公開鍵の長さ。

- **key** インポートされた秘密/公開鍵ペアを格納する ed448_key オブジェクトへのポインタ。

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 Ed448 鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または privSz が ED448_KEY_SIZE より小さい、もしくは pubSz が ED448_PUB_KEY_SIZE より小さい場合に返されます。

Example

```
int ret;
byte priv[] = { 57バイトの秘密鍵で初期化 };
byte pub[] = { 対応する公開鍵で初期化 };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
    &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

B.25.2.15 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
)
```

この関数は、一対のバッファから公開/秘密 Ed448 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、公開鍵が秘密鍵に対してチェックされます。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **pub** 公開鍵を含むバッファへのポインタ。
- **pubSz** 公開鍵の長さ。
- **key** インポートされた秘密/公開鍵ペアを格納する ed448_key オブジェクトへのポインタ。
- **trusted** 公開鍵データが信頼されているかどうか。

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 Ed448 鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または privSz が ED448_KEY_SIZE より小さい、もしくは pubSz が ED448_PUB_KEY_SIZE より小さい場合に返されます。

Example

```
int ret;
byte priv[] = { 57バイトの秘密鍵で初期化 };
byte pub[] = { 対応する公開鍵で初期化 };

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) {
    // 鍵のインポートエラー
}
```

B.25.2.16 function wc_ed448_export_public

```
int wc_ed448_export_public(
    const ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed448_key 構造体から秘密鍵をエクスポートします。公開鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 公開鍵をエクスポートする ed448_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。公開鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed448_import_public](#)
- [wc_ed448_import_public_ex](#)
- [wc_ed448_export_private_only](#)

Return:

- 0 公開鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが秘密鍵を格納するのに十分な大きさでない場合に返されます。このエラーを返す際、関数は outLen に必要なサイズを設定します。

Example

```
int ret;
ed448_key key;
// key を初期化し、key を作成

char pub[57];
word32 pubSz = sizeof(pub);
```

```
ret = wc_ed448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // 公開鍵のエクスポートエラー
}
```

B.25.2.17 function wc_ed448_export_private_only

```
int wc_ed448_export_private_only(
    const ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed448_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 秘密鍵をエクスポートする ed448_key 構造体へのポインタ。
- **out** 秘密鍵を格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。秘密鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed448_export_public](#)
- [wc_ed448_import_private_key](#)
- [wc_ed448_import_private_key_ex](#)

Return:

- 0 秘密鍵のエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが秘密鍵を格納するのに十分な大きさでない場合に返されます。

Example

```
int ret;
ed448_key key;
// key を初期化し、key を作成

char priv[57]; // 秘密鍵のみなので 57 バイト
word32 privSz = sizeof(priv);
ret = wc_ed448_export_private_only(&key, priv, &privSz);
if (ret != 0) {
    // 秘密鍵のエクスポートエラー
}
```

B.25.2.18 function wc_ed448_export_private

```
int wc_ed448_export_private(
    const ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed448_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 鍵ペアをエクスポートする ed448_key 構造体へのポインタ。
- **out** 鍵ペアを格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。鍵ペアのエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- wc_ed448_import_private
- [wc_ed448_export_private_only](#)

Return:

- 0 鍵ペアのエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが鍵ペアを格納するのに十分な大きさでない場合に返されます。

Example

```
ed448_key key;
wc_ed448_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // 57 バイトの Ed448 鍵を初期化

byte out[114]; // out は十分なバッファサイズである必要があります
word32 outLen = sizeof(out);
int key_size = wc_ed448_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // 関数が outLen をリセットしたかどうかを確認するため、out のサイズと outLen を比較
}
```

B.25.2.19 function wc_ed448_export_key

```
int wc_ed448_export_key(
    const ed448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

この関数は、ed448_key 構造体から秘密鍵と公開鍵を個別にエクスポートします。秘密鍵をバッファ priv に格納し、このバッファに書き込まれたバイト数を privSz に設定します。公開鍵をバッファ pub に格納し、このバッファに書き込まれたバイト数を pubSz に設定します。

Parameters:

- **key** 鍵ペアをエクスポートする ed448_key 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。秘密鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。公開鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed448_export_private](#)

- `wc_ed448_export_public`

Return:

- 0 鍵ペアのエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが鍵ペアを格納するのに十分な大きさでない場合に返されます。

Example

```
int ret;
ed448_key key;
// key を初期化し、key を作成

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) {
    // 秘密鍵と公開鍵のエクスポートエラー
}
```

B.25.2.20 function wc_ed448_check_key

```
int wc_ed448_check_key(
    ed448_key * key
)
```

この関数は、ed448_key 構造体内の公開鍵が秘密鍵と一致するかチェックします。

Parameters:

- **key** 秘密鍵と公開鍵を保持する ed448_key 構造体へのポインタ。

See:

- `wc_ed448_import_private_key`
- `wc_ed448_import_private_key_ex`

Return:

- 0 秘密鍵と公開鍵が一致した場合に返されます。
- BAD_FUNC_ARGS 指定された key が NULL の場合に返されます。

Example

```
int ret;
byte priv[] = { 57バイトの秘密鍵で初期化 };
byte pub[] = { 対応する公開鍵で初期化 };

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) {
    // 鍵のチェックエラー
}
```

B.25.2.21 function wc_ed448_size

```
int wc_ed448_size(  
    const ed448_key * key  
)
```

この関数は、Ed448 秘密鍵のサイズ（57 バイト）を返します。

Parameters:

- **key** 鍵サイズを取得する ed448_key 構造体へのポインタ。

See: [wc_ed448_make_key](#)

Return:

- ED448_KEY_SIZE 有効な秘密鍵のサイズ（57 バイト）。
- BAD_FUNC_ARGS 指定された key が NULL の場合に返されます。

Example

```
int keySz;  
ed448_key key;  
// key を初期化し、key を作成  
keySz = wc_ed448_size(&key);  
if (keySz == 0) {  
    // 鍵サイズの決定エラー  
}
```

B.25.2.22 function wc_ed448_priv_size

```
int wc_ed448_priv_size(  
    const ed448_key * key  
)
```

この関数は、秘密鍵のサイズ（秘密 + 公開）をバイト単位で返します。

Parameters:

- **key** 鍵サイズを取得する ed448_key 構造体へのポインタ。

See: [wc_ed448_pub_size](#)

Return:

- ED448_PRIV_KEY_SIZE 秘密鍵のサイズ（114 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```
ed448_key key;  
wc_ed448_init(&key);  
  
WC_RNG rng;  
wc_InitRng(&rng);  
  
wc_ed448_make_key(&rng, 57, &key); // 57 バイトの Ed448 鍵を初期化  
int key_size = wc_ed448_priv_size(&key);
```

B.25.2.23 function wc_ed448_pub_size

```
int wc_ed448_pub_size(  
    const ed448_key * key  
)
```

この関数は、圧縮鍵のサイズをバイト単位で返します（公開鍵）。

Parameters:

- **key** 鍵サイズを取得する ed448_key 構造体へのポインタ。

See: [wc_ed448_priv_size](#)

Return:

- ED448_PUB_KEY_SIZE 圧縮公開鍵のサイズ（57 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```
ed448_key key;  
wc_ed448_init(&key);  
WC_RNG rng;  
wc_InitRng(&rng);  
  
wc_ed448_make_key(&rng, 57, &key); // 57 バイトの Ed448 鍵を初期化  
int key_size = wc_ed448_pub_size(&key);
```

B.25.2.24 function wc_ed448_sig_size

```
int wc_ed448_sig_size(  
    const ed448_key * key  
)
```

この関数は、Ed448 署名のサイズ（バイト単位で 114）を返します。

Parameters:

- **key** 署名サイズを取得する ed448_key 構造体へのポインタ。

See: [wc_ed448_sign_msg](#)

Return:

- ED448_SIG_SIZE Ed448 署名のサイズ（114 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```
int sigSz;  
ed448_key key;  
// key を初期化し、key を作成  
  
sigSz = wc_ed448_sig_size(&key);  
if (sigSz == 0) {  
    // 署名サイズの決定エラー  
}
```

B.26 プラットフォームセキュリティアーキテクチャ (PSA)API

B.26.1 Functions

	Name
int	wolfSSL_CTX_psa_enable (WOLFSSL_CTX * ctx) この関数は、指定されたコンテキストで PSA サポートを有効にします。
int	wolfSSL_set_psa_ctx (WOLFSSL * ssl, struct psa_ssl_ctx * ctx) この関数は、指定された SSL セッションの PSA コンテキストをセットアップします
void	wolfSSL_free_psa_ctx (struct psa_ssl_ctx * ctx) この関数は、PSA コンテキストによって使用されるリソースを解放します
int	wolfSSL_psa_set_private_key_id (struct psa_ssl_ctx * ctx, psa_key_id_t id) この関数は、SSL セッションで使用される秘密鍵を設定します

B.26.2 Functions Documentation

B.26.2.1 function wolfSSL_CTX_psa_enable

```
int wolfSSL_CTX_psa_enable(  
    WOLFSSL_CTX * ctx  
)
```

この関数は、指定されたコンテキストで PSA サポートを有効にします。

Parameters:

- **ctx** PSA サポートを有効にする必要がある WOLFSSL_CTX オブジェクトへのポインタ

See: [wolfSSL_set_psa_ctx](#)

Return:

- WOLFSSL_SUCCESS 成功時
- BAD_FUNC_ARG ctx == NULL の場合

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");
```

B.26.2.2 function wolfSSL_set_psa_ctx

```
int wolfSSL_set_psa_ctx(  
    WOLFSSL * ssl,  
    struct psa_ssl_ctx * ctx  
)
```

この関数は、指定された SSL セッションの PSA コンテキストをセットアップします

Parameters:

- **ssl** ctx を有効にする WOLFSSL へのポインタ
- **ctx** struct psa_ssl_ctx へのポインタ (SSL セッションごとに一意である必要があります)

See:

- [wolfSSL_psa_set_private_key_id](#)
- [wolfSSL_psa_free_psa_ctx](#)

Return:

- WOLFSSL_SUCCESS 成功時
- BAD_FUNC_ARG ssl または ctx が NULL の場合

この関数は、指定された SSL セッションへの TLS コールバック用の PSA コンテキストをセットアップします。セッションの終了時に、コンテキストによって使用されたリソースは、[wolfSSL_free_psa_ctx\(\)](#) を使用して解放する必要があります。

Example

```
// 新しい SSL セッションを作成
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// PSA コンテキストをセットアップ
ret = wolfSSL_set_psa_ctx(ssl, ctx);
```

B.26.2.3 function wolfSSL_free_psa_ctx

```
void wolfSSL_free_psa_ctx(
    struct psa_ssl_ctx * ctx
)
```

この関数は、PSA コンテキストによって使用されるリソースを解放します

Parameters:

- **ctx** struct psa_ssl_ctx へのポインタ

See: [wolfSSL_set_psa_ctx](#)

B.26.2.4 function wolfSSL_psa_set_private_key_id

```
int wolfSSL_psa_set_private_key_id(
    struct psa_ssl_ctx * ctx,
    psa_key_id_t id
)
```

この関数は、SSL セッションで使用される秘密鍵を設定します

Parameters:

- **ctx** struct psa_ssl_ctx へのポインタ
- **id** 秘密鍵として使用されるキーの PSA ID

See: [wolfSSL_set_psa_ctx](#)

Example


```
// 新しい SSL セッションを作成
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
psa_key_id_t key_id;

// キープロビジョニングは既に完了
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);
```

B.27 アルゴリズム - SipHash

B.27.1 Functions

	Name
int	wc_InitSipHash (SipHash * siphash, const unsigned char * key, unsigned char outSz) この関数は、MAC サイズに対するキーを使用して SipHash を初期化します。
int	wc_SipHashUpdate (SipHash * siphash, const unsigned char * in, word32 inSz) 長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。
int	wc_SipHashFinal (SipHash * siphash, unsigned char * out, unsigned char outSz) データの MAC 処理を完了します。結果は out に格納されます。
int	wc_SipHash (const unsigned char * key, const unsigned char * in, word32 inSz, unsigned char * out, unsigned char outSz) この関数は、キーに基づいて MAC を計算するために、SipHash を使用してデータをワンショットで処理します。

B.27.2 Functions Documentation

B.27.2.1 function wc_InitSipHash

```
int wc_InitSipHash(
    SipHash * siphash,
    const unsigned char * key,
    unsigned char outSz
)
```

この関数は、MAC サイズに対するキーを使用して SipHash を初期化します。

Parameters:

- **siphash** MAC に使用する SipHash 構造体へのポインタ
- **key** 16 バイト配列へのポインタ
- **outSz** MAC として出力するバイト数

See:

- [wc_SipHash](#)
- [wc_SipHashUpdate](#)
- [wc_SipHashFinal](#)

Return:

- 0 初期化に成功した場合に返されます
- BAD_FUNC_ARG siphash または key が NULL の場合に返されます
- BAD_FUNC_ARG outSz が 8 でも 16 でもない場合に返されます

Example

```
SipHash siphash[1];
unsigned char key[16] = { ... };
byte macSz = 8; // 8 または 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");
}
```

B.27.2.2 function wc_SipHashUpdate

```
int wc_SipHashUpdate(
    SipHash * siphash,
    const unsigned char * in,
    word32 inSz
)
```

長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。

Parameters:

- **siphash** MAC に使用する SipHash 構造体へのポインタ
- **in** MAC するデータ
- **inSz** MAC するデータのサイズ

See:

- [wc_SipHash](#)
- [wc_InitSipHash](#)
- [wc_SipHashFinal](#)

Return:

- 0 MAC へのデータ追加に成功した場合に返されます
- BAD_FUNC_ARG siphash が NULL の場合に返されます
- BAD_FUNC_ARG in が NULL で inSz がゼロでない場合に返されます

Example

```
SipHash siphash[1];
byte data[] = { MAC するデータ };
word32 len = sizeof(data);
```

```

if ((ret = wc_InitSipHash(siphhash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphhash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphhash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}

```

B.27.2.3 function wc_SipHashFinal

```

int wc_SipHashFinal(
    SipHash * siphhash,
    unsigned char * out,
    unsigned char outSz
)

```

データの MAC 処理を完了します。結果は out に格納されます。

Parameters:

- **siphhash** MAC に使用する SipHash 構造体へのポインタ
- **out** MAC 値を保持するバイト配列
- **outSz** MAC として出力するバイト数

See:

- [wc_SipHash](#)
- [wc_InitSipHash](#)
- [wc_SipHashUpdate](#)

Return:

- 0 完了に成功した場合に返されます。
- BAD_FUNC_ARG siphhash または out が NULL の場合に返されます
- BAD_FUNC_ARG outSz が初期化された値と同じでない場合に返されます

Example

```

SipHash siphhash[1];
byte mac[8] = { ... }; // 8 または 16 バイト
byte macSz = sizeof(mac);

if ((ret = wc_InitSipHash(siphhash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphhash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphhash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}

```

B.27.2.4 function wc_SipHash

```

int wc_SipHash(
    const unsigned char * key,

```

```
    const unsigned char * in,  
    word32 inSz,  
    unsigned char * out,  
    unsigned char outSz  
)
```

この関数は、キーに基づいて MAC を計算するために、SipHash を使用してデータをワンショットで処理します。

Parameters:

- **key** 16 バイト配列へのポインタ
- **in** MAC するデータ
- **inSz** MAC するデータのサイズ
- **out** MAC 値を保持するバイト配列
- **outSz** MAC として出力するバイト数

See:

- [wc_InitSipHash](#)
- [wc_SipHashUpdate](#)
- [wc_SipHashFinal](#)

Return:

- 0 MAC に成功した場合に返されます
- BAD_FUNC_ARG key または out が NULL の場合に返されます
- BAD_FUNC_ARG in が NULL で inSz がゼロでない場合に返されます
- BAD_FUNC_ARG outSz が 8 でも 16 でもない場合に返されます

Example

```
unsigned char key[16] = { ... };  
byte data[] = { MAC するデータ };  
word32 len = sizeof(data);  
byte mac[8] = { ... }; // 8 または 16 バイト  
byte macSz = sizeof(mac);  
  
if ((ret = wc_SipHash(key, data, len, mac, macSz)) != 0) {  
    WOLFSSL_MSG("wc_SipHash failed");  
}
```

C API ヘッダーファイル

C.1 dox_comments/header_files-ja/aes.h

C.1.1 Functions

	Name
int	wc_AesSetKey (Aes * aes, const byte * key, word32 len, const byte * iv, int dir) この関数は、キーを設定し、初期化ベクトルを設定することで AES 構造体を初期化します。
int	wc_AesSetIV (Aes * aes, const byte * iv) この関数は、特定の AES オブジェクトの初期化ベクトルを設定します。この関数を呼び出す前に AES オブジェクトを初期化する必要があります。
int	wc_AesCbcEncrypt (Aes * aes, byte * out, const byte * in, word32 sz) 入力バッファ in から平文メッセージを暗号化し、AES を使用した暗号ブロック連鎖により結果の暗号文を出力バッファ out に格納します。この関数は、メッセージを暗号化する前に AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、入力メッセージが AES ブロック長に整列していることを前提としており、入力長がブロック長の倍数であることを期待します。ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合、これはオプションでチェックおよび強制されます。ブロック倍数の入力を保証するために、事前に PKCS#7 スタイルのパディングを追加する必要があります。これは、自動的にパディングを追加する OpenSSL の AES-CBC メソッドとは異なります。wolfSSL と対応する OpenSSL 関数を相互運用させるには、OpenSSL コマンドライン関数で -nopad オプションを指定して、wolfSSL の AesCbcEncrypt メソッドのように動作し、暗号化中に余分なパディングを追加しないようにする必要があります。

	Name
int	wc_AesCbcDecrypt(Aes * aes, byte * out, const byte * in, word32 sz) 入力バッファ in から暗号を復号し、AES を使用した暗号ブロック連鎖により結果の平文を出力バッファ out に格納します。この関数は、メッセージを復号する前に AesSetKey を呼び出して AES 構造体が初期化されている必要があります。この関数は、元のメッセージが AES ブロック長に整列していたことを前提としており、入力長がブロック長の倍数であることを期待します。ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合、これはオプションでチェックおよび強制されます。これは、PKCS#7 パディングを自動的に追加する OpenSSL の AES-CBC メソッドとは異なり、ブロック倍数の入力を必要としません。wolfSSL 関数と同等の OpenSSL 関数を相互運用させるには、OpenSSL コマンドライン関数で -nopad オプションを指定して、wolfSSL の AesCbcEncrypt メソッドのように動作し、復号中にエラーを発生させないようにする必要があります。
int	wc_AesCtrEncrypt(Aes * aes, byte * out, const byte * in, word32 sz) AES を使用した CTR モードで、入力バッファ in からメッセージを暗号化/復号し、結果の暗号文を出力バッファ out に格納します。この関数は、コンパイル時に WOLFSSL_AES_COUNTER が有効になっている場合のみ有効です。この関数を呼び出す前に、AesSetKey を介して AES 構造体を初期化する必要があります。この関数は復号と暗号化の両方に使用されることに注意してください。注：暗号化と復号に同じ API を使用することについて。ユーザは暗号化/復号用の Aes 構造体を区別する必要があります。
int	wc_AesEncryptDirect(Aes * aes, byte * out, const byte * in) この関数は、入力ブロック in の 1 ブロック暗号化を出力ブロック out に行います。提供された AES 構造体のキーを使用し、この関数を呼び出す前に wc_AesSetKey で初期化する必要があります。wc_AesSetKey は iv を NULL に設定して呼び出す必要があります。これは、構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

	Name
int	wc_AesDecryptDirect (Aes * aes, byte * out, const byte * in) この関数は、入力ブロック in の 1 ブロック復号を出力ブロック out に行います。提供された AES 構造体のキーを使用し、この関数を呼び出す前に wc_AesSetKey で初期化する必要があります。wc_AesSetKey は iv を NULL に設定して呼び出す必要があります。これは、構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。
int	wc_AesSetKeyDirect (Aes * aes, const byte * key, word32 len, const byte * iv, int dir) この関数は、AES を使用した CTR モードの AES キーを設定するために使用されます。指定されたキー、iv (初期化ベクトル)、および暗号化 dir (方向) で AES オブジェクトを初期化します。構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。現在、wc_AesSetKeyDirect は内部的に wc_AesSetKey を使用しています。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。
int	wc_AesGcmSetKey (Aes * aes, const byte * key, word32 len) この関数は、AES GCM (Galois/Counter Mode) のキーを設定するために使用されます。指定されたキーで AES オブジェクトを初期化します。コンパイル時に構成オプション HAVE_AESGCM が有効になっている場合のみ有効です。
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) この関数は、バッファ in に保持されている入力メッセージを暗号化し、結果の暗号文を出力バッファ out に格納します。暗号化の呼び出しごとに新しい iv (初期化ベクトル) が必要です。また、入力認証ベクトル authIn を認証タグ authTag にエンコードします。

	Name
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) この関数は、バッファ in に保持されている入力暗号文を復号し、結果のメッセージテキストを出力バッファ out に格納します。また、入力認証ベクトル authIn を、提供された認証タグ authTag と照合してチェックします。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。
int	wc_GmacSetKey (Gmac * gmac, const byte * key, word32 len) この関数は、Galois メッセージ認証に使用される GMAC オブジェクトのキーを初期化および設定します。
int	wc_GmacUpdate (Gmac * gmac, const byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, byte * authTag, word32 authTagSz) この関数は、authIn 入力の Gmac ハッシュを生成し、結果を authTag バッファに格納します。wc_GmacUpdate を実行した後、生成された authTag を既知の認証タグと比較して、メッセージの真正性を検証する必要があります。
int	wc_AesCcmSetKey (Aes * aes, const byte * key, word32 keySz) この関数は、CCM (Counter with CBC-MAC) を使用して AES オブジェクトのキーを設定します。AES 構造体へのポインタを受け取り、提供されたキーで初期化します。
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) この関数は、CCM (Counter with CBC-MAC) を使用して、入力メッセージ in を出力バッファ out に暗号化します。その後、authIn 入力から認証タグ authTag を計算して格納します。
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) この関数は、CCM (Counter with CBC-MAC) を使用して、入力暗号文 in を出力バッファ out に復号します。その後、authIn 入力から認証タグ authTag を計算します。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

	Name
int	wc_AesXtsInit (XtsAes * aes, void * heap, int devId) これは AES-XTS コンテキストを初期化するためのものです。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。
int	wc_AesXtsSetKeyNoInit (XtsAes * aes, const byte * key, word32 len, int dir) これは、最初に wc_AesXtsInit() を呼び出した後、キーを正しい暗号化または復号タイプに設定するのに役立ちます。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。
int	wc_AesXtsSetKey (XtsAes * aes, const byte * key, word32 len, int dir, void * heap, int devId) これは、キーを正しい暗号化または復号タイプに設定するのに役立ちます。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。
int	wc_AesXtsEncryptSector (XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)wc_AesXtsEncrypt と同じプロセスですが、バイト配列の代わりに word64 型を tweak 値として使用します。これは word64 をバイト配列に変換し、wc_AesXtsEncrypt を呼び出すだけです。
int	wc_AesXtsDecryptSector (XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)wc_AesXtsDecrypt と同じプロセスですが、バイト配列の代わりに word64 型を tweak 値として使用します。これは word64 をバイト配列に変換するだけです。
int	wc_AesXtsEncrypt (XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)XTS モードの AES。(XTS) Tweak と Cipher Text Stealing を使用した XEX 暗号化。
int	wc_AesXtsDecrypt (XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz) 暗号化と同じプロセスですが、Aes キーは AES_DECRYPTION タイプです。
int	wc_AesXtsFree (XtsAes * aes) これは、XtsAes 構造体によって使用されるリソースを解放するためのものです。
int	wc_AesInit (Aes * aes, void * heap, int devId)Aes 構造体を初期化します。使用するヒープヒントと非同期ハードウェアで使用する ID を設定します。使用が完了したら、ユーザが Aes 構造体に対して wc_AesFree を呼び出す必要があります。

	Name
void	wc_AesFree (Aes * aes) 該当する場合、Aes 構造体に関連付けられたリソースを解放します。内部的には時々 no-op になることもありますが、新しい環境で使用するためにアプリケーションコードが移植される場合（呼び出しが適用される場合）など、一般的なベストプラクティスとしてすべてのケースで呼び出すことをお勧めします。
int	wc_AesCfbEncrypt (Aes * aes, byte * out, const byte * in, word32 sz) CFB モードの AES。
int	wc_AesCfbDecrypt (Aes * aes, byte * out, const byte * in, word32 sz) CFB モードの AES。
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) この関数は、RFC 5297 で説明されている SIV（合成初期化ベクトル）暗号化を実行します。
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) この関数は、RFC 5297 で説明されている SIV（合成初期化ベクトル）復号を実行します。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。
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) この関数は、「EAX: A Conventional Authenticated-Encryption Mode」(https://eprint.iacr.org/2003/069) で説明されている AES EAX 暗号化と認証を実行します。これは、すべての暗号化と認証操作を 1 つの関数呼び出しで実行する「ワンショット」API です。

	Name
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) この関数は、「EAX: A Conventional Authenticated-Encryption Mode」(https://eprint.iacr.org/2003/069) で説明されている AES EAX 復号と認証を実行します。これは、すべての復号と認証操作を 1 つの関数呼び出しで実行する「ワンショット」API です。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。
WOLFSSL_API int	wc_AesEaxInit (AesEax * eax, const byte * key, word32 keySz, const byte * nonce, word32 nonceSz, const byte * authIn, word32 authInSz) この関数は、認証付き暗号化または復号で使用する AesEax オブジェクトを初期化します。この関数は、AES EAX 増分 API のいずれかと使用する前に、AesEax オブジェクトに対して呼び出す必要があります。ワンショット EAX API 関数を使用する場合は呼び出す必要はありません。この関数で初期化されたすべての AesEax インスタンスは、インスタンスの使用が完了したら wc_AesEaxFree() の呼び出しで解放する必要があります。
WOLFSSL_API int	**wc_AesEaxEncryptUpdate の呼び出しで事前に初期化されている必要があります。
WOLFSSL_API int	**wc_AesEaxDecryptUpdate の呼び出しで事前に初期化されている必要があります。
WOLFSSL_API int	**wc_AesEaxAuthDataUpdate の呼び出しで事前に初期化されている必要があります。
WOLFSSL_API int	**wc_AesEaxEncryptFinal を使用して必ず解放してください。
WOLFSSL_API int	**wc_AesEaxDecryptFinal を使用して必ず解放してください。
WOLFSSL_API int	wc_AesEaxFree (AesEax * eax) この関数は、AesEax ラッパー構造体内の Aes インスタンスによって使用されるリソース、特にキーを解放します。wc_AesEaxInit で初期化された後、すべての必要な EAX 操作が完了した時点で AesEax 構造体に対して呼び出す必要があります。
int	wc_AesCtsEncrypt (const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv) この関数は、CTS モードを使用して AES 暗号化を実行します。これは、すべての操作を 1 回の呼び出しで処理するワンショット API です。

	Name
int	wc_AesCtsDecrypt (const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv) この関数は、CTS モードを使用して AES 復号を実行します。これは、すべての操作を 1 回の呼び出しで処理するワンショット API です。
int	wc_AesCtsEncryptUpdate (Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz) この関数は、AES CTS 暗号化の更新ステップを実行します。平文のチャンクを処理し、中間データを保存します。
int	wc_AesCtsEncryptFinal (Aes * aes, byte * out, word32 * outSz) この関数は、AES CTS 暗号化操作を完了します。残りの平文を処理し、暗号化を完了します。
int	wc_AesCtsDecryptUpdate (Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz) この関数は、AES CTS 復号の更新ステップを実行します。暗号文のチャンクを処理し、中間データを保存します。
int	wc_AesCtsDecryptFinal (Aes * aes, byte * out, word32 * outSz) この関数は、AES CTS 復号操作を完了します。残りの暗号文を処理し、復号を完了します。

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
)
```

この関数は、キーを設定し、初期化ベクトルを設定することで AES 構造体を初期化します。

Parameters:

- **aes** 変更する AES 構造体へのポインタ
- **key** 暗号化と復号のための 16、24、または 32 バイトの秘密鍵
- **len** 渡されたキーの長さ
- **iv** キーの初期化に使用される初期化ベクトルへのポインタ
- **dir** 暗号の方向。暗号化する場合は AES_ENCRYPTION を設定し、復号する場合は AES_DECRYPTION を設定します。一部のモード（CFB および CTR）の方向は常に AES_ENCRYPTION です。

See:

- **wc_AesSetKeyDirect**
- **wc_AesSetIV**

Return:

- 0 キーと初期化ベクトルの設定に成功した場合に返されます。

- BAD_FUNC_ARG キーの長さが無効な場合に返されます。

Example

```
Aes enc;
int ret = 0;
byte key[] = { 16、24、または32バイトのキー };
byte iv[] = { 16バイトの iv };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // aes key の初期化に失敗
}
if (ret = wc_AesSetKey(&enc, key, AES_BLOCK_SIZE, iv,
AES_ENCRYPTION) != 0) {
    // aes key の設定に失敗
}
```

C.1.2.2 function wc_AesSetIV

```
int wc_AesSetIV(
    Aes * aes,
    const byte * iv
)
```

この関数は、特定の AES オブジェクトの初期化ベクトルを設定します。この関数を呼び出す前に AES オブジェクトを初期化する必要があります。

Parameters:

- **aes** 初期化ベクトルを設定する AES 構造体へのポインタ
- **iv** AES 構造体を初期化するために使用される初期化ベクトル。値が NULL の場合、デフォルトの動作は iv を 0 に初期化します。

See:

- [wc_AesSetKeyDirect](#)
- [wc_AesSetKey](#)

Return:

- 0 初期化ベクトルの設定に成功した場合に返されます。
- BAD_FUNC_ARG AES ポインタが NULL の場合に返されます。

Example

```
Aes enc;
// enc key を設定
byte iv[] = { 16バイトの iv };
if (ret = wc_AesSetIV(&enc, iv) != 0) {
    // aes iv の設定に失敗
}
```

C.1.2.3 function wc_AesCbcEncrypt

```
int wc_AesCbcEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

入力バッファ in から平文メッセージを暗号化し、AES を使用した暗号ブロック連鎖により結果の暗号文を出力バッファ out に格納します。この関数は、メッセージを暗号化する前に AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、入力メッセージが AES ブロック長に整列していることを前提としており、入力長がブロック長の倍数であることを期待します。ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合、これはオプションでチェックおよび強制されます。ブロック倍数の入力を保証するために、事前に PKCS#7 スタイルのパディングを追加する必要があります。これは、自動的にパディングを追加する OpenSSL の AES-CBC メソッドとは異なります。wolfSSL と対応する OpenSSL 関数を相互運用させるには、OpenSSL コマンドライン関数で -nopad オプションを指定して、wolfSSL の AesCbcEncrypt メソッドのように動作し、暗号化中に余分なパディングを追加しないようにする必要があります。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 暗号化されたメッセージの暗号文を格納する出力バッファへのポインタ
- **in** 暗号化されるメッセージを含む入力バッファへのポインタ
- **sz** 入力メッセージのサイズ

See:

- [wc_AesInit](#)
- [wc_AesSetKey](#)
- [wc_AesSetIV](#)
- [wc_AesCbcDecrypt](#)

Return:

- 0 メッセージの暗号化に成功した場合に返されます。
- BAD_ALIGN_E ブロック整列エラーで返される場合があります。
- BAD_LENGTH_E ライブラリが WOLFSSL_AES_CBC_LENGTH_CHECKS でビルドされている場合、入力長が AES ブロック長の倍数でない場合に返されます。

Example

```
Aes enc;
int ret = 0;
// wc_AesInit と wc_AesSetKey を使用して enc を初期化、方向は
// AES_ENCRYPTION を使用
byte msg[AES_BLOCK_SIZE * n]; // 16 バイトの倍数
// msg にデータを入力
byte cipher[AES_BLOCK_SIZE * n]; // 16 バイトの倍数
if ((ret = wc_AesCbcEncrypt(&enc, cipher, message, sizeof(msg))) != 0) {
// ブロック整列エラー
}
```

C.1.2.4 function wc_AesCbcDecrypt

```
int wc_AesCbcDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

入力バッファ in から暗号を復号し、AES を使用した暗号ブロック連鎖により結果の平文を出力バッファ out に格納します。この関数は、メッセージを復号する前に AesSetKey を呼び出して AES 構造体が初期化されている必要があります。この関数は、元のメッセージが AES ブロック長に整列していたことを前提としており、入力長がブロック長の倍数であることを期待します。ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合、これはオプションでチェックおよび強制されます。これは、PKCS#7 パディング

を自動的に追加する OpenSSL の AES-CBC メソッドとは異なり、ブロック倍数の入力を必要としません。wolfSSL 関数と同等の OpenSSL 関数を相互運用させるには、OpenSSL コマンドライン関数で `-nopad` オプションを指定して、wolfSSL の `AesCbcEncrypt` メソッドのように動作し、復号中にエラーを発生させないようにする必要があります。

Parameters:

- **aes** データを復号するために使用される AES オブジェクトへのポインタ。
- **out** 復号されたメッセージの平文を格納する出力バッファへのポインタ。サイズは `AES_BLOCK_LENGTH` の倍数である必要があり、必要に応じてパディングされます
- **in** 復号される暗号文を含む入力バッファへのポインタ。サイズは `AES_BLOCK_LENGTH` の倍数である必要があり、必要に応じてパディングされます
- **sz** 入力メッセージのサイズ。

See:

- `wc_AesInit`
- `wc_AesSetKey`
- `wc_AesCbcEncrypt`

Return:

- 0 メッセージの復号に成功した場合に返されます。
- `BAD_ALIGN_E` ブロック整列エラーで返される場合があります。
- `BAD_LENGTH_E` ライブラリが `WOLFSSL_AES_CBC_LENGTH_CHECKS` でビルドされている場合、入力長が AES ブロック長の倍数でない場合に返されます。

Example

```
Aes dec;
int ret = 0;
// wc_AesInit と wc_AesSetKey を使用して dec を初期化、方向は
// AES_DECRYPTION を使用
byte cipher[AES_BLOCK_SIZE * n]; // 16 バイトの倍数
// cipher に暗号文を入力
byte plain [AES_BLOCK_SIZE * n];
if ((ret = wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher))) != 0 ) {
// ブロック整列エラー
}
```

C.1.2.5 function wc_AesCtrEncrypt

```
int wc_AesCtrEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES を使用した CTR モードで、入力バッファ `in` からメッセージを暗号化/復号し、結果の暗号文を出力バッファ `out` に格納します。この関数は、コンパイル時に `WOLFSSL_AES_COUNTER` が有効になっている場合のみ有効です。この関数を呼び出す前に、`AesSetKey` を介して AES 構造体を初期化する必要があります。この関数は復号と暗号化の両方に使用されることに注意してください。注：暗号化と復号に同じ API を使用することについて。ユーザは暗号化/復号用の `Aes` 構造体を区別する必要があります。

Parameters:

- **aes** データを復号するために使用される AES オブジェクトへのポインタ
- **out** 暗号化されたメッセージの暗号文を格納する出力バッファへのポインタ サイズは `AES_BLOCK_LENGTH` の倍数である必要があり、必要に応じてパディングされます

- **in** 暗号化される平文を含む入力バッファへのポインタ サイズは AES_BLOCK_LENGTH の倍数である
必要があり、必要に応じてパディングされます
- **sz** 入力平文のサイズ

See: [wc_AesSetKey](#)

Return: int wolfSSL エラーまたは成功ステータスに対応する整数値

Example

```
Aes enc;
Aes dec;
// wc_AesInit と wc_AesSetKeyDirect を使用して enc と dec を初期化、方向は
// AES_ENCRYPTION を使用。基盤となる API は暗号化のみを呼び出し、
// デフォルトで暗号に対して暗号化を呼び出すと暗号の復号が
// 行われるため

byte msg[AES_BLOCK_SIZE * n]; // n は正の整数で、msg を
16バイトの倍数にする
// plain にメッセージテキストを入力
byte cipher[AES_BLOCK_SIZE * n];
byte decrypted[AES_BLOCK_SIZE * n];
wc_AesCtrEncrypt(&enc, cipher, msg, sizeof(msg)); // plain を暗号化
wc_AesCtrEncrypt(&dec, decrypted, cipher, sizeof(cipher));
// 暗号文を復号
```

C.1.2.6 function wc_AesEncryptDirect

```
int wc_AesEncryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

この関数は、入力ブロック in の 1 ブロック暗号化を出力ブロック out に行います。提供された AES 構造体のキーを使用し、この関数を呼び出す前に wc_AesSetKey で初期化する必要があります。wc_AesSetKey は iv を NULL に設定して呼び出す必要があります。これは、構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 暗号化されたメッセージの暗号文を格納する出力バッファへのポインタ
- **in** 暗号化される平文を含む入力バッファへのポインタ

See:

- [wc_AesDecryptDirect](#)
- [wc_AesSetKeyDirect](#)

Return: int wolfSSL エラーまたは成功ステータスに対応する整数値

Example

```
Aes enc;
// wc_AesInit と wc_AesSetKey を使用して enc を初期化、方向は
// AES_ENCRYPTION を使用
byte msg [AES_BLOCK_SIZE]; // 16 バイト
// msg を暗号化する平文で初期化
```



```
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
)
```

この関数は、入力ブロック in の 1 ブロック復号を出力ブロック out に行います。提供された AES 構造体のキーを使用し、この関数を呼び出す前に wc_AesSetKey で初期化する必要があります。wc_AesSetKey は iv を NULL に設定して呼び出す必要があります。これは、構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 復号された暗号文の平文を格納する出力バッファへのポインタ
- **in** 復号される暗号文を含む入力バッファへのポインタ

See:

- [wc_AesEncryptDirect](#)
- [wc_AesSetKeyDirect](#)

Return: int wolfSSL エラーまたは成功ステータスに対応する整数値

Example

```
Aes dec;
// wc_AesInit と wc_AesSetKey を使用して enc を初期化、方向は
// AES_DECRYPTION を使用
byte cipher [AES_BLOCK_SIZE]; // 16 バイト
// cipher を復号する暗号文で初期化
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
)
```

この関数は、AES を使用した CTR モードの AES キーを設定するために使用されます。指定されたキー、iv (初期化ベクトル)、および暗号化 dir (方向) で AES オブジェクトを初期化します。構成オプション WOLFSSL_AES_DIRECT が有効になっている場合のみ有効です。現在、wc_AesSetKeyDirect は内部的に wc_AesSetKey を使用しています。警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **key** 暗号化と復号のための 16、24、または 32 バイトの秘密鍵

- **len** 渡されたキーの長さ
- **iv** キーの初期化に使用される初期化ベクトル
- **dir** 暗号の方向。暗号化する場合は AES_ENCRYPTION を設定し、復号する場合は AES_DECRYPTION を設定します。(wolfssl/wolfcrypt/aes.h の列挙型を参照) (注: Aes カウンタモード (ストリーム暗号) で wc_AesSetKeyDirect を使用する場合、暗号化と復号の両方に AES_ENCRYPTION のみを使用)

See:

- [wc_AesEncryptDirect](#)
- [wc_AesDecryptDirect](#)
- [wc_AesSetKey](#)

Return:

- 0 キーの設定に成功した場合に返されます。
- BAD_FUNC_ARG 指定されたキーの長が無効な場合に返されます。

Example

```
Aes enc;
int ret = 0;
byte key[] = { 16, 24, または32バイトのキー };
byte iv[] = { 16バイトの iv };

if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // aes key の初期化に失敗
}
if (ret = wc_AesSetKeyDirect(&enc, key, sizeof(key), iv,
AES_ENCRYPTION) != 0) {
    // aes key の設定に失敗
}
```

C.1.2.9 function wc_AesGcmSetKey

```
int wc_AesGcmSetKey(
    Aes * aes,
    const byte * key,
    word32 len
)
```

この関数は、AES GCM (Galois/Counter Mode) のキーを設定するために使用されます。指定されたキーで AES オブジェクトを初期化します。コンパイル時に構成オプション HAVE_AESGCM が有効になっている場合のみ有効です。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **key** 暗号化と復号のための 16、24、または 32 バイトの秘密鍵
- **len** 渡されたキーの長さ

See:

- [wc_AesGcmEncrypt](#)
- [wc_AesGcmDecrypt](#)

Return:

- 0 キーの設定に成功した場合に返されます。
- BAD_FUNC_ARG 指定されたキーの長が無効な場合に返されます。

Example

```

Aes enc;
int ret = 0;
byte key[] = { 16, 24, 32バイトのキー };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // aes key の初期化に失敗
}
if (ret = wc_AesGcmSetKey(&enc, key, sizeof(key)) != 0) {
    // aes key の設定に失敗
}

```

C.1.2.10 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
)

```

この関数は、バッファ in に保持されている入力メッセージを暗号化し、結果の暗号文を出力バッファ out に格納します。暗号化の呼び出しごとに新しい iv（初期化ベクトル）が必要です。また、入力認証ベクトル authIn を認証タグ authTag にエンコードします。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 暗号文を格納する出力バッファへのポインタ サイズは in のサイズ (sz) と一致する必要があります
- **in** 暗号化するメッセージを保持する入力バッファへのポインタ サイズは AES_BLOCK_LENGTH の倍数である必要があります、必要に応じてパディングされます
- **sz** 暗号化する入力メッセージの長さ
- **iv** 初期化ベクトルを含むバッファへのポインタ
- **ivSz** 初期化ベクトルの長さ
- **authTag** 認証タグを格納するバッファへのポインタ
- **authTagSz** 希望する認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ
- **authInSz** 入力認証ベクトルの長さ

See:

- [wc_AesGcmSetKey](#)
- [wc_AesGcmDecrypt](#)

Return: 0 入力メッセージの暗号化に成功した場合に返されます

Example

```

Aes enc;
// wc_AesInit() と wc_AesGcmSetKey を呼び出して Aes 構造体を初期化

byte plain[AES_BLOCK_LENGTH * n]; // n は正の整数で
plain を16バイトの倍数にする

```

```
// plain を暗号化するメッセージで初期化
byte cipher[sizeof(plain)];
byte iv[] = // 16 バイトの iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // 認証ベクトル

wc_AesGcmEncrypt(&enc, cipher, plain, sizeof(cipher), iv, sizeof(iv),
    authTag, sizeof(authTag), authIn, sizeof(authIn));
```

C.1.2.11 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
)
```

この関数は、バッファ in に保持されている入力暗号文を復号し、結果のメッセージテキストを出力バッファ out に格納します。また、入力認証ベクトル authIn を、提供された認証タグ authTag と照合してチェックします。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** メッセージテキストを格納する出力バッファへのポインタ サイズは in のサイズ (sz) と一致する必要があります
- **in** 復号する暗号文を保持する入力バッファへのポインタ サイズは AES_BLOCK_LENGTH の倍数である必要があります、必要に応じてパディングされます
- **sz** 復号する暗号文の長さ
- **iv** 初期化ベクトルを含むバッファへのポインタ
- **ivSz** 初期化ベクトルの長さ
- **authTag** 認証タグを含むバッファへのポインタ
- **authTagSz** 希望する認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ
- **authInSz** 入力認証ベクトルの長さ

See:

- [wc_AesGcmSetKey](#)
- [wc_AesGcmEncrypt](#)

Return:

- 0 入力メッセージの復号と認証に成功した場合に返されます
- AES_GCM_AUTH_E 認証タグが提供された認証コードベクトル authTag と一致しない場合に返されます。

Example

```
Aes enc; // wc_AesGcmEncrypt に渡されたものと同じ構造体を使用可能
// まだ完了していない場合は、wc_AesInit と wc_AesGcmSetKey を呼び出して
```

```
// aes 構造体を初期化
```

```
byte cipher[AES_BLOCK_LENGTH * n]; // n は正の整数で
cipher を16バイトの倍数にする
// cipher を復号する暗号文で初期化
byte output[sizeof(cipher)];
byte iv[] = // 16 バイトの iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // 認証ベクトル

wc_AesGcmDecrypt(&enc, output, cipher, sizeof(cipher), iv, sizeof(iv),
    authTag, sizeof(authTag), authIn, sizeof(authIn));
```

C.1.2.12 function wc_GmacSetKey

```
int wc_GmacSetKey(
    Gmac * gmac,
    const byte * key,
    word32 len
)
```

この関数は、Galois メッセージ認証に使用される GMAC オブジェクトのキーを初期化および設定します。

Parameters:

- **gmac** 認証に使用される gmac オブジェクトへのポインタ
- **key** 認証のための 16、24、または 32 バイトの秘密鍵
- **len** キーの長さ

See:

- [wc_GmacUpdate](#)
- [wc_AesInit](#)

Return:

- 0 キーの設定に成功した場合に返されます
- BAD_FUNC_ARG キーの長さが無効な場合に返されます。

Example

```
Gmac gmac;
key[] = { 16、24、または32バイト長のキー };
wc_AesInit(gmac.aes, HEAP_HINT, INVALID_DEVID); // devId が更新されていることを確認
wc_GmacSetKey(&gmac, key, sizeof(key));
```

C.1.2.13 function wc_GmacUpdate

```
int wc_GmacUpdate(
    Gmac * gmac,
    const byte * iv,
    word32 ivSz,
    const byte * authIn,
    word32 authInSz,
    byte * authTag,
    word32 authTagSz
)
```

この関数は、authIn 入力の Gmac ハッシュを生成し、結果を authTag バッファに格納します。wc_GmacUpdate を実行した後、生成された authTag を既知の認証タグと比較して、メッセージの真正性を検証する必要があります。

Parameters:

- **gmac** 認証に使用される gmac オブジェクトへのポインタ
- **iv** ハッシュに使用される初期化ベクトル
- **ivSz** 使用される初期化ベクトルのサイズ
- **authIn** 検証する認証ベクトルを含むバッファへのポインタ
- **authInSz** 認証ベクトルのサイズ
- **authTag** Gmac ハッシュを格納する出力バッファへのポインタ
- **authTagSz** Gmac ハッシュを格納するために使用される出力バッファのサイズ

See:

- [wc_GmacSetKey](#)
- [wc_AesInit](#)

Return: 0 Gmac ハッシュの計算に成功した場合に返されます。

Example

```
Gmac gmac;
key[] = { 16, 24, または32バイト長のキー };
iv[] = { 16バイト長の iv };

wc_AesInit(&gmac.aes, HEAP_HINT, INVALID_DEVID); // devId が更新されていることを確認
wc_GmacSetKey(&gmac, key, sizeof(key));
authIn[] = { 16バイトの認証入力 };
tag[AES_BLOCK_SIZE]; // 認証コードを格納

wc_GmacUpdate(&gmac, iv, sizeof(iv), authIn, sizeof(authIn), tag,
sizeof(tag));
```

C.1.2.14 function wc_AesCcmSetKey

```
int wc_AesCcmSetKey(
    Aes * aes,
    const byte * key,
    word32 keySz
)
```

この関数は、CCM（Counter with CBC-MAC）を使用して AES オブジェクトのキーを設定します。AES 構造体へのポインタを受け取り、提供されたキーで初期化します。

Parameters:

- **aes** 提供されたキーを格納する aes 構造体
- **key** 暗号化と復号のための 16、24、または 32 バイトの秘密鍵
- **keySz** 提供されたキーのサイズ

See:

- [wc_AesCcmEncrypt](#)
- [wc_AesCcmDecrypt](#)

Return: none

Example

```
Aes enc;
key[] = { 16、24、または32バイト長のキー };

wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID); // devId が更新されていることを確認
wc_AesCcmSetKey(&enc, key, sizeof(key));
```

C.1.2.15 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
)
```

この関数は、CCM（Counter with CBC-MAC）を使用して、入力メッセージ in を出力バッファ out に暗号化します。その後、authIn 入力から認証タグ authTag を計算して格納します。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 暗号文を格納する出力バッファへのポインタ
- **in** 暗号化するメッセージを保持する入力バッファへのポインタ
- **sz** 暗号化する入力メッセージの長さ
- **nonce** ナンス（1 回のみ使用される数値）を含むバッファへのポインタ
- **nonceSz** ナンスの長さ
- **authTag** 認証タグを格納するバッファへのポインタ
- **authTagSz** 希望する認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ
- **authInSz** 入力認証ベクトルの長さ

See:

- [wc_AesCcmSetKey](#)
- [wc_AesCcmDecrypt](#)

Return: none

Example

```
Aes enc;
// wc_AesInit と wc_AesCcmSetKey で enc を初期化

nonce[] = { ナンスを初期化 };
plain[] = { 平文メッセージ };
cipher[sizeof(plain)];

authIn[] = { 16バイトの認証入力 };
tag[AES_BLOCK_SIZE]; // 認証コードを格納

wc_AesCcmEncrypt(&enc, cipher, plain, sizeof(plain), nonce, sizeof(nonce),
    tag, sizeof(tag), authIn, sizeof(authIn));
```

C.1.2.16 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
)
```

この関数は、CCM（Counter with CBC-MAC）を使用して、入力暗号文 in を出力バッファ out に復号します。その後、authIn 入力から認証タグ authTag を計算します。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

Parameters:

- **aes** データを暗号化するために使用される AES オブジェクトへのポインタ
- **out** 暗号文を格納する出力バッファへのポインタ
- **in** 暗号化するメッセージを保持する入力バッファへのポインタ
- **sz** 復号する入力暗号文の長さ
- **nonce** ナンス（1 回のみ使用される数値）を含むバッファへのポインタ
- **nonceSz** ナンスの長さ
- **authTag** 認証タグを格納するバッファへのポインタ
- **authTagSz** 希望する認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ
- **authInSz** 入力認証ベクトルの長さ

See:

- [wc_AesCcmSetKey](#)
- [wc_AesCcmEncrypt](#)

Return:

- 0 入力メッセージの復号に成功した場合に返されます
- AES_CCM_AUTH_E 認証タグが提供された認証コードベクトル authTag と一致しない場合に返されます。

Example

```
Aes dec;
// wc_AesInit と wc_AesCcmSetKey で dec を初期化

nonce[] = { ナンスを初期化 };
cipher[] = { 暗号化されたメッセージ };
plain[sizeof(cipher)];

authIn[] = { 16バイトの認証入力 };
tag[AES_BLOCK_SIZE] = { 検証のために受信した認証タグ };

int return = wc_AesCcmDecrypt(&dec, plain, cipher, sizeof(cipher),
    nonce, sizeof(nonce), tag, sizeof(tag), authIn, sizeof(authIn));
if(return != 0) {
```



```
// 復号エラー、無効な認証コード  
}
```

C.1.2.17 function wc_AesXtsInit

```
int wc_AesXtsInit(  
    XtsAes * aes,  
    void * heap,  
    int devId  
)
```

これは AES-XTS コンテキストを初期化するためのものです。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。

Parameters:

- **aes** 暗号化/復号プロセスのための AES キー
- **heap** メモリに使用するヒープヒント。NULL でも可
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定

See:

- wc_AesXtsSetKey
- wc_AesXtsSetKeyNoInit
- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 成功

Example

```
XtsAes aes;
```

```
if(wc_AesXtsInit(&aes, NULL, INVALID_DEVID) != 0)  
{  
    // エラーを処理  
}  
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION) != 0)  
{  
    // エラーを処理  
}  
wc_AesXtsFree(&aes);
```

C.1.2.18 function wc_AesXtsSetKeyNoInit

```
int wc_AesXtsSetKeyNoInit(  
    XtsAes * aes,  
    const byte * key,  
    word32 len,  
    int dir  
)
```

これは、最初に wc_AesXtsInit() を呼び出した後、キーを正しい暗号化または復号タイプに設定するのに役立ちます。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。

Parameters:

- **aes** 暗号化/復号プロセスのための AES キー

- **key** aes キー | tweak キーを保持するバッファ
- **len** キーバッファの長さ (バイト単位)。キーサイズの 2 倍である必要があります。つまり、16 バイトのキーの場合は 32 です。
- **dir** 方向、AES_ENCRYPTION または AES_DECRYPTION

See:

- [wc_AesXtsEncrypt](#)
- [wc_AesXtsDecrypt](#)
- [wc_AesXtsFree](#)

Return: 0 成功

Example

```
XtsAes aes;
```

```
if(wc_AesXtsInit(&aes, NULL, 0) != 0)
{
    // エラーを処理
}
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0)
    != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

C.1.2.19 function wc_AesXtsSetKey

```
int wc_AesXtsSetKey(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir,
    void * heap,
    int devId
)
```

これは、キーを正しい暗号化または復号タイプに設定するのに役立ちます。使用が完了したら、ユーザが aes キーに対して wc_AesXtsFree を呼び出す必要があります。

Parameters:

- **aes** 暗号化/復号プロセスのための AES キー
- **key** aes キー | tweak キーを保持するバッファ
- **len** キーバッファの長さ (バイト単位)。キーサイズの 2 倍である必要があります。つまり、16 バイトのキーの場合は 32 です。
- **dir** 方向、AES_ENCRYPTION または AES_DECRYPTION
- **heap** メモリに使用するヒープヒント。NULL でも可
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定

See:

- [wc_AesXtsInit](#)
- [wc_AesXtsSetKeyNoInit](#)
- [wc_AesXtsEncrypt](#)
- [wc_AesXtsDecrypt](#)

- `wc_AesXtsFree`

Return: 0 成功

Example

```
XtsAes aes;
```

```
if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, INVALID_DEVID)
    ↪ != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

C.1.2.20 function `wc_AesXtsEncryptSector`

```
int wc_AesXtsEncryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

`wc_AesXtsEncrypt` と同じプロセスですが、バイト配列の代わりに `word64` 型を `tweak` 値として使用します。これは `word64` をバイト配列に変換し、`wc_AesXtsEncrypt` を呼び出すだけです。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 暗号文を保持する出力バッファ
- **in** 暗号化する入力平文バッファ
- **sz** `out` と `in` バッファの両方のサイズ
- **sector** `tweak` に使用する値

See:

- `wc_AesXtsEncrypt`
- `wc_AesXtsDecrypt`
- `wc_AesXtsInit`
- `wc_AesXtsSetKeyNoInit`
- `wc_AesXtsSetKey`
- `wc_AesXtsFree`

Return: 0 成功

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
word64 s = VALUE;

// AES_ENCRYPTION を dir としてキーを設定
if(wc_AesXtsEncryptSector(&aes, cipher, plain, SIZE, s) != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

C.1.2.21 function wc_AesXtsDecryptSector

```
int wc_AesXtsDecryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

wc_AesXtsDecrypt と同じプロセスですが、バイト配列の代わりに word64 型を tweak 値として使用します。これは word64 をバイト配列に変換するだけです。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 平文を保持する出力バッファ
- **in** 復号する入力暗号文バッファ
- **sz** out と in バッファの両方のサイズ
- **sector** tweak に使用する値

See:

- [wc_AesXtsEncrypt](#)
- [wc_AesXtsDecrypt](#)
- [wc_AesXtsInit](#)
- [wc_AesXtsSetKeyNoInit](#)
- [wc_AesXtsSetKey](#)
- [wc_AesXtsFree](#)

Return: 0 成功

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
word64 s = VALUE;
```

// AES_DECRYPTION を dir として aes キーを設定し、tweak は AES_ENCRYPTION で設定

```
if(wc_AesXtsDecryptSector(&aes, plain, cipher, SIZE, s) != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

C.1.2.22 function wc_AesXtsEncrypt

```
int wc_AesXtsEncrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

XTS モードの AES。(XTS) Tweak と Cipher Text Stealing を使用した XEX 暗号化。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 暗号文を保持する出力バッファ
- **in** 暗号化する入力平文バッファ
- **sz** out と in バッファの両方のサイズ
- **i** tweak に使用する値
- **iSz** i バッファのサイズ、常に AES_BLOCK_SIZE である必要がありますが、この入力を持つことで、ユーザが関数を呼び出す方法についてのサニティチェックが追加されます。

See:

- `wc_AesXtsDecrypt`
- `wc_AesXtsInit`
- `wc_AesXtsSetKeyNoInit`
- `wc_AesXtsSetKey`
- `wc_AesXtsFree`

Return: 0 成功*Example*

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];

// AES_ENCRYPTION を dir としてキーを設定

if(wc_AesXtsEncrypt(&aes, cipher, plain, SIZE, i, sizeof(i)) != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

C.1.2.23 function wc_AesXtsDecrypt

```
int wc_AesXtsDecrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

暗号化と同じプロセスですが、Aes キーは AES_DECRYPTION タイプです。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 平文を保持する出力バッファ
- **in** 復号する入力暗号文バッファ
- **sz** out と in バッファの両方のサイズ
- **i** tweak に使用する値
- **iSz** i バッファのサイズ、常に AES_BLOCK_SIZE である必要がありますが、この入力を持つことで、ユーザが関数を呼び出す方法についてのサニティチェックが追加されます。

See:

- `wc_AesXtsEncrypt`
- `wc_AesXtsInit`
- `wc_AesXtsSetKeyNoInit`
- `wc_AesXtsSetKey`
- `wc_AesXtsFree`

Return: 0 成功

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];
```

// AES_DECRYPTION を dir としてキーを設定し、tweak は AES_ENCRYPTION で設定

```
if(wc_AesXtsDecrypt(&aes, plain, cipher, SIZE, i, sizeof(i)) != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

C.1.2.24 function `wc_AesXtsFree`

```
int wc_AesXtsFree(
    XtsAes * aes
)
```

これは、XtsAes 構造体によって使用されるリソースを解放するためのものです

Parameters:

- `aes` 解放する AES キー

See:

- `wc_AesXtsEncrypt`
- `wc_AesXtsDecrypt`
- `wc_AesXtsInit`
- `wc_AesXtsSetKeyNoInit`
- `wc_AesXtsSetKey`

Return: 0 成功

Example

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0) != 0)
{
    // エラーを処理
}
wc_AesXtsFree(&aes);
```

C.1.2.25 function `wc_AesInit`

```
int wc_AesInit(
    Aes * aes,
    void * heap,
```

```
    int devId
)
```

Aes 構造体を初期化します。使用するヒープヒントと非同期ハードウェアで使用する ID を設定します。使用が完了したら、ユーザが Aes 構造体に対して wc_AesFree を呼び出す必要があります。

Parameters:

- **aes** 初期化する aes 構造体
- **heap** 必要に応じて malloc / free に使用するヒープヒント
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesFree

Return: 0 成功

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; // 非同期を使用しない場合は INVALID_DEVID がデフォルト

// 使用する場合はここでヒープヒントを設定可能

wc_AesInit(&enc, hint, devId);
```

C.1.2.26 function wc_AesFree

```
void wc_AesFree(
    Aes * aes
)
```

該当する場合、Aes 構造体に関連付けられたリソースを解放します。内部的には時々 no-op になることもありますが、新しい環境で使用するためにアプリケーションコードが移植される場合（呼び出しが適用される場合）など、一般的なベストプラクティスとしてすべてのケースで呼び出すことをお勧めします。

Parameters:

- **aes** 解放する aes 構造体

See: wc_AesInit

Return: no return (void 関数)

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; // 非同期を使用しない場合は INVALID_DEVID がデフォルト

// 使用する場合はここでヒープヒントを設定可能

wc_AesInit(&enc, hint, devId);
// ... 興味深いことをいくつか行う ...
wc_AesFree(&enc);
```

C.1.2.27 function wc_AesCfbEncrypt

```
int wc_AesCfbEncrypt(  
    Aes * aes,  
    byte * out,  
    const byte * in,  
    word32 sz  
)
```

CFB モードの AES。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 暗号文を保持する出力バッファ（少なくとも入力バッファと同じ大きさである必要があります）
- **in** 暗号化する入力平文バッファ
- **sz** 入力バッファのサイズ

See:

- [wc_AesCfbDecrypt](#)
- [wc_AesSetKey](#)

Return: 0 成功、失敗時は負のエラー値

Example

```
Aes aes;  
unsigned char plain[SIZE];  
unsigned char cipher[SIZE];  
  
// 暗号化と復号の両方に AES_ENCRYPTION を dir としてキーを設定  
  
if(wc_AesCfbEncrypt(&aes, cipher, plain, SIZE) != 0)  
{  
    // エラーを処理  
}
```

C.1.2.28 function wc_AesCfbDecrypt

```
int wc_AesCfbDecrypt(  
    Aes * aes,  
    byte * out,  
    const byte * in,  
    word32 sz  
)
```

CFB モードの AES。

Parameters:

- **aes** ブロック暗号化/復号に使用する AES キー
- **out** 復号されたテキストを保持する出力バッファ（少なくとも入力バッファと同じ大きさである必要があります）
- **in** 復号する入力バッファ
- **sz** 入力バッファのサイズ

See:

- [wc_AesCfbEncrypt](#)
- [wc_AesSetKey](#)

Return: 0 成功、失敗時は負のエラー値

Example

```
Aes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];

// 暗号化と復号の両方に AES_ENCRYPTION を dir としてキーを設定

if(wc_AesCfbDecrypt(&aes, plain, cipher, SIZE) != 0)
{
    // エラーを処理
}
```

C.1.2.29 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
)
```

この関数は、RFC 5297 で説明されている SIV（合成初期化ベクトル）暗号化を実行します。

Parameters:

- **key** 使用するキーを含むバイトバッファ。
- **keySz** キーバッファの長さ（バイト単位）。
- **assoc** 追加の認証された関連データ（AD）。
- **assocSz** AD バッファの長さ（バイト単位）。
- **nonce** 1 回のみ使用される数値。アルゴリズムによって AD と同じ方法で使用されます。
- **nonceSz** nonce バッファの長さ（バイト単位）。
- **in** 暗号化する平文バッファ。
- **inSz** 平文バッファの長さ。
- **siv** S2V によって出力される SIV（RFC 5297 2.4 を参照）。
- **out** 暗号文を保持するバッファ。平文バッファと同じ長さである必要があります。

See: [wc_AesSivDecrypt](#)

Return:

- 0 暗号化に成功した場合。
- BAD_FUNC_ARG key、SIV、または出力バッファが NULL の場合に返されます。また、キーサイズが 32、48、または 64 バイトでない場合にも返されます。
- Other AES または CMAC 操作が失敗した場合に返されるその他の負のエラー値。

Example

```
byte key[] = { 32、48、または64バイトのキー };
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) {
    // 暗号化に失敗
}

```

C.1.2.30 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
)

```

この関数は、RFC 5297 で説明されている SIV（合成初期化ベクトル）復号を実行します。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

Parameters:

- **key** 使用するキーを含むバイトバッファ。
- **keySz** キーバッファの長さ（バイト単位）。
- **assoc** 追加の認証された関連データ（AD）。
- **assocSz** AD バッファの長さ（バイト単位）。
- **nonce** 1 回のみ使用される数値。基盤となるアルゴリズムによって AD と同じ方法で使用されます。
- **nonceSz** nonce バッファの長さ（バイト単位）。
- **in** 復号する暗号文バッファ。
- **inSz** 暗号文バッファの長さ。
- **siv** 暗号文に付随する SIV（RFC 5297 2.4 を参照）。
- **out** 復号された平文を保持するバッファ。暗号文バッファと同じ長さである必要があります。

See: [wc_AesSivEncrypt](#)

Return:

- 0 復号に成功した場合。
- BAD_FUNC_ARG key、SIV、または出力バッファが NULL の場合に返されます。また、キーサイズが 32、48、または 64 バイトでない場合にも返されます。
- AES_SIV_AUTH_E S2V によって導出された SIV が入力 SIV と一致しない場合（RFC 5297 2.7 を参照）。
- Other AES または CMAC 操作が失敗した場合に返されるその他の負のエラー値。

Example

```

byte key[] = { 32、48、または64バイトのキー };
byte assoc[] = {0x01, 0x2, 0x3};
byte nonce[] = {0x04, 0x5, 0x6};
byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte siv[AES_BLOCK_SIZE] = { 暗号文に付属していた SIV };
byte plainText[sizeof(cipherText)];

```

```

if (wc_AesSivDecrypt(key, sizeof(key), assoc, sizeof(assoc), nonce,
    sizeof(nonce), cipherText, sizeof(cipherText), siv, plainText) != 0) {
    // 復号に失敗
}

```

C.1.2.31 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
)

```

この関数は、「EAX: A Conventional Authenticated-Encryption Mode」(<https://eprint.iacr.org/2003/069>)で説明されている AES EAX 暗号化と認証を実行します。これは、すべての暗号化と認証操作を 1 つの関数呼び出しで実行する「ワンショット」API です。

Parameters:

- **key** 使用するキーを含むバッファ
- **keySz** キーバッファの長さ (バイト単位)
- **out** 暗号文を保持するバッファ。平文バッファと同じ長さである必要があります
- **in** 暗号化する平文バッファ
- **inSz** 平文バッファの長さ
- **nonce** EAX 操作に使用する暗号ナンス
- **nonceSz** nonce バッファの長さ (バイト単位)
- **authTag** 認証タグを格納するバッファへのポインタ
- **authTagSz** 希望する認証タグの長さ
- **authIn** 認証する入力データを含むバッファへのポインタ
- **authInSz** 入力認証データの長さ

See: [wc_AesEaxDecryptAuth](#)

Return:

- 0 暗号化に成功した場合。
- BAD_FUNC_ARG 入力または出力バッファが NULL の場合に返されます。また、キーサイズが有効な AES キーサイズ (16、24、または 32 バイト) でない場合にも返されます
- other AES または CMAC 操作が失敗した場合に返されるその他の負のエラー値。

Example

```

byte key[] = { 32、48、または64バイトのキー };
byte nonce[] = {0x04, 0x5, 0x6};
byte plainText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte cipherText[sizeof(plainText)]; // 出力暗号文
byte authTag[length, up to AES_BLOCK_SIZE]; // 出力 authTag

```

```

if (wc_AesEaxEncrypt(key, sizeof(key),
                    cipherText, plainText, sizeof(plainText),
                    nonce, sizeof(nonce),
                    authTag, sizeof(authTag),
                    authIn, sizeof(authIn)) != 0) {
    // 暗号化に失敗
}

```

C.1.2.32 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
)

```

この関数は、「EAX: A Conventional Authenticated-Encryption Mode」(<https://eprint.iacr.org/2003/069>)で説明されている AES EAX 復号と認証を実行します。これは、すべての復号と認証操作を 1 つの関数呼び出しで実行する「ワンショット」API です。ゼロ以外のエラーコードが返された場合、出力データは未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

Parameters:

- **key** 使用するキーを含むバイトバッファ
- **keySz** キーバッファの長さ (バイト単位)
- **out** 平文を保持するバッファ。入力暗号文バッファと同じ長さである必要があります
- **in** 復号する暗号文バッファ
- **inSz** 暗号文バッファの長さ
- **nonce** EAX 操作に使用する暗号ナンス
- **nonceSz** nonce バッファの長さ (バイト単位)
- **authTag** データの真正性をチェックするために照合する認証タグを保持するバッファ
- **authTagSz** 入力認証タグの長さ
- **authIn** 認証する入力データを含むバッファへのポインタ
- **authInSz** 入力認証データの長さ

See: `wc_AesEaxEncryptAuth`

Return:

- 0 復号に成功した場合
- BAD_FUNC_ARG 入力または出力バッファが NULL の場合に返されます。また、キーサイズが有効な AES キーサイズ (16、24、または 32 バイト) でない場合にも返されます
- AES_EAX_AUTH_E 認証タグが提供された認証コードベクトル authTag と一致しない場合
- other AES または CMAC 操作が失敗した場合に返されるその他の負のエラー値。

Example

```

byte key[] = { 32、48、または64バイトのキー };
byte nonce[] = {0x04, 0x5, 0x6};

```

```

byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte plainText[sizeof(cipherText)]; // 出力平文
byte authTag[length, up to AES_BLOCK_SIZE]; // 出力 authTag

if (wc_AesEaxDecrypt(key, sizeof(key),
                    cipherText, plainText, sizeof(plainText),
                    nonce, sizeof(nonce),
                    authTag, sizeof(authTag),
                    authIn, sizeof(authIn)) != 0) {
    // 暗号化に失敗
}

```

C.1.2.33 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
)

```

この関数は、認証付き暗号化または復号で使用する AesEax オブジェクトを初期化します。この関数は、AES EAX 増分 API のいずれかと使用する前に、AesEax オブジェクトに対して呼び出す必要があります。ワンショット EAX API 関数を使用する場合は呼び出す必要はありません。この関数で初期化されたすべての AesEax インスタンスは、インスタンスの使用が完了したら wc_AesEaxFree() の呼び出しで解放する必要があります。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体
- **key** 暗号化と復号のための 16、24、または 32 バイトの秘密鍵
- **keySz** 提供されたキーの長さ（バイト単位）
- **nonce** EAX 操作に使用する暗号ナンス
- **nonceSz** nonce バッファの長さ（バイト単位）
- **authIn** （オプション）認証ストリームに追加する入力データ 使用しない場合、この引数は NULL である必要があります
- **authInSz** 入力認証データのサイズ（バイト単位）

See:

- [wc_AesEaxEncryptUpdate](#)
- [wc_AesEaxDecryptUpdate](#)
- [wc_AesEaxAuthDataUpdate](#)
- [wc_AesEaxEncryptFinal](#)
- [wc_AesEaxDecryptFinal](#)
- [wc_AesEaxFree](#)

Return:

- 0 成功した場合
- error code 失敗した場合のエラーコード

Example

```

AesEax eax;
key[] = { 16、24、または32バイト長のキー };
nonce[] = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
plainText[] = {暗号化する平文データ};

cipherText[sizeof(plainText)]; // cipherText を保持するバッファ
authTag[length, up to AES_BLOCK_SIZE]; // 計算された認証データを保持するバッファ

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        authIn, authInSz)) != 0) {
    goto cleanup;
}

// さらに認証データを追加したい場合は、この時点で提供できます
// そうでない場合は、authIn パラメータに NULL を使用し、authIn サイズは 0 です
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.1.2.34 function wc_AesEaxEncryptUpdate

```

WOLFSSL_API int wc_AesEaxEncryptUpdate(
    AesEax * eax,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
    word32 authInSz
)

```

この関数は、AES EAX を使用して入力データを暗号化し、オプションで認証ストリームにさらに入力データを追加します。eax は、wc_AesEaxInit の呼び出しで事前に初期化されている必要があります。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体
- **out** 暗号文を保持する出力バッファ
- **in** 暗号化する平文を保持する入力バッファ
- **inSz** 入力データバッファのサイズ (バイト単位)
- **authIn** (オプション) 認証ストリームに追加する入力データ 使用しない場合、この引数は NULL である必要があります
- **authInSz** 入力認証データのサイズ (バイト単位)

See:

- `wc_AesEaxInit`
- `wc_AesEaxDecryptUpdate`
- `wc_AesEaxAuthDataUpdate`
- `wc_AesEaxEncryptFinal`
- `wc_AesEaxDecryptFinal`
- `wc_AesEaxFree`

Return:

- 0 成功した場合
- error code 失敗した場合のエラーコード

Example

```
AesEax eax;
key[]    = { 16、24、または32バイト長のキー };
nonce[]  = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
plainText[] = {暗号化する平文データ};

cipherText[sizeof(plainText)]; // cipherText を保持するバッファ
authTag[length, up to AES_BLOCK_SIZE]; // 計算された認証データを保持するバッファ

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        authIn, authInSz)) != 0) {
    goto cleanup;
}

// さらに認証データを追加したい場合は、この時点で提供できます
// そうでない場合は、authIn パラメータに NULL を使用し、authInSz は 0 です
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_AesEaxDecryptUpdate

```
WOLFSSL_API int wc_AesEaxDecryptUpdate(
    AesEax * eax,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
```

```
    word32 authInSz
)
```

この関数は、AES EAX を使用して入力データを復号し、オプションで認証ストリームにさらに入力データを追加します。eax は、`wc_AesEaxInit` の呼び出しで事前に初期化されている必要があります。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体
- **out** 復号された平文を保持する出力バッファ
- **in** 暗号文を保持する入力バッファ
- **inSz** 入力データバッファのサイズ (バイト単位)
- **authIn** (オプション) 認証ストリームに追加する入力データ 使用しない場合、この引数は NULL である必要があります
- **authInSz** 入力認証データのサイズ (バイト単位)

See:

- `wc_AesEaxInit`
- `wc_AesEaxEncryptUpdate`
- `wc_AesEaxAuthDataUpdate`
- `wc_AesEaxEncryptFinal`
- `wc_AesEaxDecryptFinal`
- `wc_AesEaxFree`

Return:

- 0 成功した場合
- error code 失敗した場合のエラーコード

Example

```
AesEax eax;
key[]    = { 16, 24, または32バイト長のキー };
nonce[]  = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
cipherText[] = {暗号化されたデータ};
```

```
plainText[sizeof(cipherText)]; // 復号されたデータを保持するバッファ
// 認証タグは、暗号化 AEAD 操作によって別の場所で生成されます
authTag[length, up to AES_BLOCK_SIZE] = { 認証タグ };
```

```
AesEax eax;
```

```
if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        authIn, authInSz)) != 0) {
    goto cleanup;
}
```

```
// さらに認証データを追加したい場合は、この時点で提供できます
// そうでない場合は、authIn パラメータに NULL を使用し、authInSz は 0 です
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.1.2.36 function wc_AesEaxAuthDataUpdate

```

WOLFSSL_API int wc_AesEaxAuthDataUpdate(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)

```

この関数は、認証ストリームに入力データを追加します。eax は、wc_AesEaxInit の呼び出しで事前に初期化されている必要があります。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体
- **authIn** 認証ストリームに追加する入力データ
- **authInSz** 入力認証データのサイズ (バイト単位)

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 成功した場合
- error code 失敗した場合のエラーコード

Example

```

AesEax eax;
key[] = { 16、24、または32バイト長のキー };
nonce[] = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
cipherText[] = { 暗号化されたデータ };

plainText[sizeof(cipherText)]; // 復号されたデータを保持するバッファ
// 認証タグは、暗号化 AEAD 操作によって別の場所で生成されます
authTag[length, up to AES_BLOCK_SIZE] = { 認証タグ };

```

```

AesEax eax;

```

```

// ここでは追加する認証データなし
if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        NULL, 0)) != 0) {
    goto cleanup;
}

```

```

}

// ここでは追加する認証データなし、後で wc_AesEaxAuthDataUpdate で追加
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.1.2.37 function wc_AesEaxEncryptFinal

```

WOLFSSL_API int wc_AesEaxEncryptFinal(
    AesEax * eax,
    byte * authTag,
    word32 authTagSz
)

```

この関数は、暗号化 AEAD 操作を完了し、現在の認証ストリームに対して認証タグを生成します。eax は、`wc_AesEaxInit` を使用して必ず解放してください。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体
- **authTag[out]** 計算された認証タグを保持するバッファ
- **authTagSz** authTag のサイズ (バイト単位)

See:

- `wc_AesEaxInit`
- `wc_AesEaxEncryptUpdate`
- `wc_AesEaxDecryptUpdate`
- `wc_AesEaxAuthDataUpdate`
- `wc_AesEaxDecryptFinal`
- `wc_AesEaxFree`

Return:

- 0 成功した場合
- error code 失敗した場合のエラーコード

Example

```

AesEax eax;
key[] = { 16、24、または32バイト長のキー };
nonce[] = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
plainText[] = { 暗号化する平文データ };
cipherText[sizeof(plainText)]; // cipherText を保持するバッファ

```

```

authTag[length, up to AES_BLOCK_SIZE]; // 計算された認証データを保持するバッファ

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        authIn, authInSz)) != 0) {
    goto cleanup;
}

// さらに認証データを追加したい場合は、この時点で提供できます
// そうでない場合は、authIn パラメータに NULL を使用し、authInSz は 0 です
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.1.2.38 function wc_AesEaxDecryptFinal

```

WOLFSSL_API int wc_AesEaxDecryptFinal(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)

```

この関数は、復号 AEAD 操作を完了し、認証タグの計算を完了して、ユーザが提供したタグに対してその有効性をチェックします。eax は、`wc_AesEaxInit` を使用して必ず解放してください。

Parameters:

- **eax** AEAD 操作のコンテキストを保持する AES EAX 構造体
- **authIn** 計算された認証タグと照合するための入力認証タグ
- **authInSz** authIn のサイズ (バイト単位)

See:

- `wc_AesEaxInit`
- `wc_AesEaxEncryptUpdate`
- `wc_AesEaxDecryptUpdate`
- `wc_AesEaxAuthDataUpdate`
- `wc_AesEaxEncryptFinal`
- `wc_AesEaxFree`

Return:

- 0 データが正常に認証された場合
- AES_EAX_AUTH_E 認証タグが提供された認証コードベクトル authIn と一致しない場合
- other error code 失敗した場合のその他のエラーコード

Example

```

AesEax eax;
key[] = { 16、24、または32バイト長のキー };
nonce[] = { 任意の長さの nonce };
authIn[] = { 認証ストリームに追加するデータ };
cipherText[] = {暗号化されたデータ};

plainText[sizeof(cipherText)]; // 復号されたデータを保持するバッファ
// 認証タグは、暗号化 AEAD 操作によって別の場所で生成されます
authTag[length, up to AES_BLOCK_SIZE] = { 認証タグ };

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                        key, keySz,
                        nonce, nonceSz,
                        authIn, authInSz)) != 0) {
    goto cleanup;
}

// さらに認証データを追加したい場合は、この時点で提供できます
// そうでない場合は、authIn パラメータに NULL を使用し、authInSz は 0 です
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.1.2.39 function wc_AesEaxFree

```

WOLFSSL_API int wc_AesEaxFree(
    AesEax * eax
)

```

この関数は、AesEax ラッパー構造体内の Aes インスタンスによって使用されるリソース、特にキーを解放します。wc_AesEaxInit で初期化された後、すべての必要な EAX 操作が完了した時点で AesEax 構造体に対して呼び出す必要があります。

Parameters:

- **eax** 解放する AES EAX インスタンス

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal

- `wc_AesEaxDecryptFinal`

Return: 0 成功

Example

```
AesEax eax;
```

```
if(wc_AesEaxInit(eax, key, keySz, nonce, nonceSz, authIn, authInSz) != 0) {
    // エラーを処理し、その後解放
    wc_AesEaxFree(&eax);
}
```

C.1.2.40 function `wc_AesCtsEncrypt`

```
int wc_AesCtsEncrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * iv
)
```

この関数は、CTS モードを使用して AES 暗号化を実行します。これは、すべての操作を 1 回の呼び出しで処理するワンショット API です。

Parameters:

- **key** 暗号化に使用される AES キーへのポインタ。
- **keySz** AES キーのサイズ (バイト単位) (16、24、または 32 バイト)。
- **out** 暗号化された暗号文を保持するバッファ。少なくとも入力と同じサイズである必要があります。
- **in** 暗号化する平文入力データへのポインタ。
- **inSz** 平文入力データのサイズ (バイト単位)。
- **iv** 暗号化に使用される初期化ベクトル (IV) へのポインタ。16 バイトである必要があります。
- **key** 暗号化に使用される AES キーへのポインタ。
- **keySz** AES キーのサイズ (バイト単位) (16、24、または 32 バイト)。
- **out** 暗号化された暗号文を保持するバッファ。少なくとも入力平文と同じサイズである必要があります。
- **in** 暗号化する平文入力データへのポインタ。
- **inSz** 平文入力データのサイズ (バイト単位)。
- **iv** 暗号化に使用される初期化ベクトル (IV) へのポインタ。16 バイトである必要があります。 *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) {
    // 暗号化エラーを処理
}
```

See:

- `wc_AesCtsDecrypt`
- `wc_AesCtsDecrypt`

Return:

- 0 暗号化に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。
- other negative error codes 暗号化失敗のためのその他の負のエラーコード。
- 0 暗号化に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。
- other negative error codes 暗号化失敗のためのその他の負のエラーコード。

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) {
    // 暗号化エラーを処理
}

```

C.1.2.41 function wc_AesCtsDecrypt

```

int wc_AesCtsDecrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * iv
)

```

この関数は、CTS モードを使用して AES 復号を実行します。これは、すべての操作を 1 回の呼び出しで処理するワンショット API です。

Parameters:

- **key** 復号に使用される AES キーへのポインタ。
- **keySz** AES キーのサイズ (バイト単位) (16、24、または 32 バイト)。
- **out** 復号された平文を保持するバッファ。少なくとも入力暗号文と同じサイズである必要があります。
- **in** 復号する暗号文入力データへのポインタ。
- **inSz** 暗号文入力データのサイズ (バイト単位)。
- **iv** 復号に使用される初期化ベクトル (IV) へのポインタ。16 バイトである必要があります。 *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) {
    // 復号エラーを処理
}

```

See: `wc_AesCtsEncrypt`

Return:

- 0 復号に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。

- other negative error codes 復号失敗のためのその他の負のエラーコード。

C.1.2.42 function wc_AesCtsEncryptUpdate

```
int wc_AesCtsEncryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)
```

この関数は、AES CTS 暗号化の更新ステップを実行します。平文のチャンクを処理し、中間データを保存します。

Parameters:

- **aes** 操作のコンテキストを保持する Aes 構造体へのポインタ。
- **out** 暗号化された暗号文を保持するバッファ。この更新ステップからの出力を保存するのに十分な大きさである必要があります。
- **outSz** out バッファに書き込まれた出力データのサイズ (バイト単位)。入力時には、out バッファに書き込むことができる最大バイト数を含める必要があります。
- **in** 暗号化する平文入力データへのポインタ。
- **inSz** 平文入力データのサイズ (バイト単位)。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) {
    // エラーを処理
}
wc_AesFree(&aes);
```

See: [wc_AesCtsDecryptUpdate](#)

Return:

- 0 処理に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。

C.1.2.43 function wc_AesCtsEncryptFinal

```
int wc_AesCtsEncryptFinal(
    Aes * aes,
    byte * out,
    word32 * outSz
)
```

この関数は、AES CTS 暗号化操作を完了します。残りの平文を処理し、暗号化を完了します。

Parameters:

- **aes** 操作のコンテキストを保持する Aes 構造体へのポインタ。
- **out** 最終的な暗号化された暗号文を保持するバッファ。この最終ステップから残りの暗号文を保存するのに十分な大きさである必要があります。
- **outSz** out バッファに書き込まれた出力データのサイズ (バイト単位)。入力時には、out バッファに書き込むことができる最大バイト数を含める必要があります。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);
// wc_AesCtsEncryptUpdate を使用して必要な更新ステップを実行
int ret = wc_AesCtsEncryptFinal(&aes, ciphertext, &outSz);
if (ret != 0) {
    // エラーを処理
}
wc_AesFree(&aes);
```

See: [wc_AesCtsDecryptFinal](#)

Return:

- 0 暗号化の完了に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。

C.1.2.44 function wc_AesCtsDecryptUpdate

```
int wc_AesCtsDecryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)
```

この関数は、AES CTS 復号の更新ステップを実行します。暗号文のチャンクを処理し、中間データを保存します。

Parameters:

- **aes** 操作のコンテキストを保持する Aes 構造体へのポインタ。
- **out** 復号された平文を保持するバッファ。この更新ステップからの出力を保存するのに十分な大きさである必要があります。
- **outSz** out バッファに書き込まれた出力データのサイズ (バイト単位)。入力時には、out バッファに書き込むことができる最大バイト数を含める必要があります。
- **in** 復号する暗号文入力データへのポインタ。
- **inSz** 暗号文入力データのサイズ (バイト単位)。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) {
    // エラーを処理
}
wc_AesFree(&aes);

```

See: [wc_AesCtsEncryptUpdate](#)

Return:

- 0 処理に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。

C.1.2.45 function wc_AesCtsDecryptFinal

```

int wc_AesCtsDecryptFinal(
    Aes * aes,
    byte * out,
    word32 * outSz
)

```

この関数は、AES CTS 復号操作を完了します。残りの暗号文を処理し、復号を完了します。

Parameters:

- **aes** 操作のコンテキストを保持する Aes 構造体へのポインタ。
- **out** 最終的な復号された平文を保持するバッファ。この最終ステップから残りの平文を保存するのに十分な大きさである必要があります。
- **outSz** out バッファに書き込まれた出力データのサイズ（バイト単位）。入力時には、out バッファに書き込むことができる最大バイト数を含める必要があります。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);
// wc_AesCtsDecryptUpdate を使用して必要な更新ステップを実行
int ret = wc_AesCtsDecryptFinal(&aes, plaintext, &outSz);
if (ret != 0) {
    // エラーを処理
}
wc_AesFree(&aes);

```

See: [wc_AesCtsEncryptFinal](#)

Return:

- 0 復号の完了に成功した場合。
- BAD_FUNC_ARG 入力引数が無効な場合。

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_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,
    /* 計算された認証タグの出力 */
    byte* authTag, word32 authTagSz,
    /* 認証する入力データ */
    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,
                                     /* 検証する認証タグ */
                                     const byte* authTag, word32 authTagSz,
                                     /* 認証する入力データ */
                                     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);
```

C.2 dox_comments/header_files-ja/arc4.h

C.2.1 Functions

	Name
int	wc_Arc4Process (Arc4 * arc4, byte * out, const byte * in, word32 length) この関数は、バッファ in から入力メッセージを暗号化し、暗号文を出力バッファ out に格納するか、またはバッファ in から暗号文を復号し、平文を出力バッファ out に格納します。ARC4 暗号化を使用します。この関数は暗号化と復号の両方に使用されます。このメソッドを呼び出す前に、wc_Arc4SetKey を使用して ARC4 構造体を初期化する必要があります。
int	wc_Arc4SetKey (Arc4 * arc4, const byte * key, word32 length) この関数は、ARC4 オブジェクトのキーを設定し、暗号として使用するために初期化します。wc_Arc4Process で暗号化に使用する前に呼び出す必要があります。

C.2.2 Functions Documentation

C.2.2.1 function wc_Arc4Process

```
int wc_Arc4Process(
    Arc4 * arc4,
    byte * out,
    const byte * in,
    word32 length
)
```

この関数は、バッファ in から入力メッセージを暗号化し、暗号文を出力バッファ out に格納するか、またはバッファ in から暗号文を復号し、平文を出力バッファ out に格納します。ARC4 暗号化を使用します。この関数は暗号化と復号の両方に使用されます。このメソッドを呼び出す前に、wc_Arc4SetKey を使用して ARC4 構造体を初期化する必要があります。

Parameters:

- **arc4** メッセージを処理するために使用される ARC4 構造体へのポインタ
- **out** 処理されたメッセージを格納する出力バッファへのポインタ
- **in** 処理するメッセージを含む入力バッファへのポインタ
- **length** 処理するメッセージの長さ

See: **wc_Arc4SetKey**

Return: none

Example

```
Arc4 enc;
byte key[] = { 暗号化に使用するキー };
wc_Arc4SetKey(&enc, key, sizeof(key));
```

```
byte plain[] = { エンコードする平文 };
byte cipher[sizeof(plain)];
byte decrypted[sizeof(plain)];
// plain を cipher に暗号化
```

```

wc_Arc4Process(&enc, cipher, plain, sizeof(plain));
// cipher を復号
wc_Arc4Process(&enc, decrypted, cipher, sizeof(cipher));

```

C.2.2.2 function wc_Arc4SetKey

```

int wc_Arc4SetKey(
    Arc4 * arc4,
    const byte * key,
    word32 length
)

```

この関数は、ARC4 オブジェクトのキーを設定し、暗号として使用するために初期化します。wc_Arc4Process で暗号化に使用する前に呼び出す必要があります。

Parameters:

- **arc4** 暗号化に使用される arc4 構造体へのポインタ
- **key** arc4 構造体を初期化するために使用するキー
- **length** arc4 構造体を初期化するために使用するキーの長さ

See: [wc_Arc4Process](#)

Return: none

Example

```

Arc4 enc;
byte key[] = { 暗号化に使用するキーで初期化 };
wc_Arc4SetKey(&enc, key, sizeof(key));

```

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);

```

C.3 dox_comments/header_files-ja/asn.h

C.4 dox_comments/header_files-ja/asn_public.h

C.4.1 Functions

	Name
int	wc_InitCert (Cert * cert) この関数は、デフォルトのオプションでデフォルトの証明書を初期化します: version = 3 (0x2)、serial = 0、sigType = SHA_WITH_RSA、issuer = 空白、daysValid = 500、selfSigned = 1 (true) 発行者として subject を使用、subject = 空白

	Name
Cert *	wc_CertNew (void * heap) この関数は、証明書操作中に使用するための新しい Cert 構造体を割り当てます。アプリケーションが構造体自体を割り当てる必要はありません。Cert 構造体もこの関数によって初期化されるため、wc_InitCert() を呼び出す必要がなくなります。アプリケーションが割り当てられた Cert 構造体の使用を終了したら、wc_CertFree() を呼び出す必要があります。
void	wc_CertFree (Cert * cert) この関数は、wc_CertNew() への以前の呼び出しによって cert 構造体に割り当てられたメモリを解放します。
int	wc_MakeCert (Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng) CA 署名付き証明書を作成するために使用されます。subject 情報が入力された後に呼び出されます。この関数は、cert 入力から x509 証明書 v3 RSA または ECC を作成します。次に、この証明書を derBuffer に書き込みます。証明書を生成するために、rsaKey または eccKey のいずれかを受け取ります。このメソッドを呼び出す前に、証明書を wc_InitCert で初期化する必要があります。
int	wc_MakeCertReq (Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey) この関数は、入力証明書を使用して証明書署名要求を作成し、出力を derBuffer に書き込みます。証明書要求を生成するために、rsaKey または eccKey のいずれかを受け取ります。証明書署名要求に署名するには、この関数の後に wc_SignCert() を呼び出す必要があります。この関数の使用例については、wolfCrypt テストアプリケーション (./wolfcrypt/test/test.c) を参照してください。
int	wc_SignCert (int requestSz, int sigType, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng) この関数は buffer に署名し、署名を buffer の末尾に追加します。署名タイプを受け取ります。CA 署名付き証明書を作成する場合は、wc_MakeCert() または wc_MakeCertReq() の後に呼び出す必要があります。
int	wc_MakeSelfCert (Cert * cert, byte * derBuffer, word32 derSz, RsaKey * key, WC_RNG * rng) この関数は、自己署名用の前の 2 つの関数、wc_MakeCert と wc_SignCert の組み合わせです (前の関数は CA 要求に使用できます)。証明書を生成してから署名し、自己署名証明書を生成します。

	Name
int	wc_SetIssuer (Cert * cert, const char * issuerFile) この関数は、証明書の発行者を、提供された pem issuerFile の発行者に設定します。また、証明書の自己署名属性を false に変更します。issuerFile で指定された発行者は、cert 発行者を設定する前に検証されます。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetSubject (Cert * cert, const char * subjectFile) この関数は、証明書の subject を、提供された pem subjectFile の subject に設定します。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetSubjectRaw (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの subject から証明書の生の subject を設定します。このメソッドは、署名前に生の subject フィールドを設定するために使用されます。
int	wc_GetSubjectRaw (byte ** subjectRaw, Cert * cert) この関数は、証明書構造体から生の subject を取得します。
int	wc_SetAltNames (Cert * cert, const char * file) この関数は、証明書の代替名を、提供された pem ファイル内の代替名に設定します。これは、同じ証明書で複数のドメインを保護したい場合に便利です。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetIssuerBuffer (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの発行者から証明書の発行者を設定します。また、証明書の自己署名属性を false に変更します。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetIssuerRaw (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの発行者から証明書の生の発行者を設定します。このメソッドは、署名前に生の発行者フィールドを設定するために使用されます。
int	wc_SetSubjectBuffer (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの subject から証明書の subject を設定します。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetAltNamesBuffer (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの代替名から証明書の代替名を設定します。これは、同じ証明書で複数のドメインを保護したい場合に便利です。このメソッドは、署名前にフィールドを設定するために使用されます。

	Name
int	wc_SetDatesBuffer (Cert * cert, const byte * der, int derSz) この関数は、提供された der バッファの日付範囲から証明書の日付を設定します。このメソッドは、署名前にフィールドを設定するために使用されます。
int	wc_SetAuthKeyIdFromPublicKey (Cert * cert, RsaKey * rsaKey, ecc_key * ekey)RSA または ECC 公開鍵から AKID を設定します。注：rsaKey または ekey のいずれか一方のみを設定し、両方は設定しないでください。
int	wc_SetAuthKeyIdFromCert (Cert * cert, const byte * der, int derSz)DER エンコードされた証明書から AKID を設定します。
int	wc_SetAuthKeyId (Cert * cert, const char * file)PEM 形式の証明書ファイルから AKID を設定します。
int	wc_SetSubjectKeyIdFromPublicKey (Cert * cert, RsaKey * rsaKey, ecc_key * ekey)RSA または ECC 公開鍵から SKID を設定します。
int	wc_SetSubjectKeyId (Cert * cert, const char * file)PEM 形式の公開鍵ファイルから SKID を設定します。両方の引数が必要です。
int	wc_SetKeyUsage (Cert * cert, const char * value) この関数を使用すると、カンマ区切りのトークン文字列を使用してキー使用法を設定できます。受け入れられるトークンは：digitalSignature、nonRepudiation、contentCommitment、keyCertSign、cRLSign、dataEncipherment、keyAgreement、keyEncipherment、encipherOnly、decipherOnly です。例： "digitalSignature,nonRepudiation" nonRepudiation と contentCommitment は同じ用途です。
int	wc_PemPubKeyToDer (const char * fileName, unsigned char * derBuf, int derSz) ファイルから PEM キーを読み込み、DER エンコードされたバッファに変換します。
int	wc_PubKeyPemToDer (const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz)PEM エンコードされた公開鍵を DER に変換します。バッファに書き込まれたバイト数、またはエラーの場合は負の値を返します。
int	wc_PemCertToDer (const char * fileName, unsigned char * derBuf, int derSz) この関数は、pem 証明書を der 証明書に変換し、結果の証明書を提供された derBuf バッファに配置します。

	Name
int	wc_DerToPem (const byte * der, word32 derSz, byte * output, word32 outSz, int type) この関数は、der バッファに含まれる der 形式の入力証明書を、output バッファに含まれる pem 形式の出力証明書に変換します。これはインプレース変換ではなく、pem 形式の出力を格納するために別のバッファを使用する必要があることに注意してください。
int	wc_DerToPemEx (const byte * der, word32 derSz, byte * output, word32 outSz, byte * cipher_info, int type) この関数は、der 形式の入力証明書を変換します、der バッファに含まれる、pem 形式の出力証明書に変換し、output バッファに格納します。これはインプレース変換ではなく、pem 形式の出力を格納するために別のバッファを使用する必要があることに注意してください。暗号情報の設定を許可します。
int	wc_KeyPemToDer (const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, const char * pass) PEM 形式のキーを DER 形式に変換します。
int	wc_CertPemToDer (const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, int type) この関数は、PEM 形式の証明書を DER 形式に変換します。OpenSSL 関数 PemToDer を呼び出します。
int	wc_GetPubKeyDerFromCert (struct DecodedCert * cert, byte * derKey, word32 * derKeySz) この関数は、入力された DecodedCert 構造体から DER 形式の公開鍵を取得します。この API を呼び出す前に、ユーザーは wc_InitDecodedCert() と wc_ParseCert() を呼び出す必要があります。wc_InitDecodedCert() は DER/ASN.1 エンコードされた証明書を受け入れます。PEM 証明書を DER に変換するには、wc_InitDecodedCert() を呼び出す前にまず wc_CertPemToDer() を使用してください。
int	wc_EccPrivateKeyDecode (const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz) この関数は、入力バッファ input から ECC 秘密鍵を読み取り、秘密鍵を解析し、それを使用して ecc_key オブジェクトを生成し、key に格納します。
int	wc_EccKeyToDer (ecc_key * key, byte * output, word32 inLen) この関数は、秘密 ECC キーを der 形式に書き込みます。
int	wc_EccPublicKeyDecode (const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz) 入力バッファから ECC 公開鍵をデコードします。ECC キーを取得するために ASN シーケンスを解析します。

	Name
int	wc_EccPublicKeyToDer (ecc_key * key, byte * output, word32 inLen, int with_AlgCurve) この関数は、ECC 公開鍵を DER 形式に変換します。使用されたバッファのサイズを返します。DER 形式の ECC 公開鍵は出力バッファに格納されます。with_AlgCurve フラグは、アルゴリズムと曲線情報を持つヘッダーを含めます
int	wc_EccPublicKeyToDer_ex (ecc_key * key, byte * output, word32 inLen, int with_AlgCurve, int comp) この関数は、ECC 公開鍵を DER 形式に変換します。使用されたバッファのサイズを返します。DER 形式の ECC 公開鍵は出力バッファに格納されます。with_AlgCurve フラグは、アルゴリズムと曲線情報を持つヘッダーを含めます。comp パラメータは、公開鍵を圧縮形式でエクスポートするかどうかを決定します。
int	wc_Curve25519PrivateKeyDecode (const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz) この関数は、DER エンコードされたバッファから Curve25519 秘密鍵 (のみ) をデコードします
int	wc_Curve25519PublicKeyDecode (const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz) この関数は、DER エンコードされたバッファから Curve25519 公開鍵 (のみ) をデコードします。
int	wc_Curve25519KeyDecode (const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz) この関数は、DER エンコードされたバッファから Curve25519 キーをデコードします。秘密鍵、公開鍵、または両方をデコードできます。
int	wc_Curve25519PrivateKeyToDer (curve25519_key * key, byte * output, word32 inLen) この関数は、Curve25519 秘密鍵を DER 形式にエンコードします。入力キー構造体に公開鍵が含まれている場合、それは無視されます。
int	wc_Curve25519PublicKeyToDer (curve25519_key * key, byte * output, word32 inLen, int withAlg) この関数は、Curve25519 公開鍵を DER 形式にエンコードします。入力キー構造体に秘密鍵が含まれている場合、それは無視されます。
int	wc_Curve25519KeyToDer (curve25519_key * key, byte * output, word32 inLen, int withAlg) この関数は、Curve25519 キーを DER 形式にエンコードします。秘密鍵、公開鍵、または両方をエンコードできます。
word32	wc_EncodeSignature (byte * out, const byte * digest, word32 digSz, int hashOID) この関数は、デジタル署名を出力バッファにエンコードし、作成されたエンコードされた署名のサイズを返します。

	Name
int	wc_GetCTC_HashOID (int type) この関数は、ハッシュタイプに対応するハッシュ OID を返します。例えば、WC_SHA512 タイプが与えられた場合、この関数は SHA512 ハッシュに対応する識別子 SHA512h を返します。
void	wc_SetCert_Free (Cert * cert) この関数は、証明書構造体のデコードされた証明書キャッシュによって使用されるメモリとリソースをクリーンアップします。WOLFSSL_CERT_GEN_CACHE が定義されている場合、デコードされた証明書構造体は証明書構造体にキャッシュされます。これにより、証明書設定関数への後続の呼び出しで、各呼び出しでデコードされた証明書を解析することを回避できます。
int	wc_GetPkcs8TraditionalOffset (byte * input, word32 * inOutIdx, word32 sz) この関数は、PKCS#8 暗号化されていないバッファ内の従来の秘密鍵の先頭を見つけます。
int	wc_CreatePKCS8Key (byte * out, word32 * outSz, byte * key, word32 keySz, int algoID, const byte * curveOID, word32 oidSz) この関数は、DER 秘密鍵を受け取り、PKCS#8 形式に変換します。PKCS#12 縮小キーバッグの作成にも使用されます。RFC 5208 を参照してください。
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) この関数は、暗号化されていない PKCS#8 DER キー（例：wc_CreatePKCS8Key によって作成されたもの）を受け取り、PKCS#8 暗号化形式に変換します。結果の暗号化されたキーは、wc_DecryptPKCS8Key を使用して復号できます。RFC 5208 を参照してください。
int	wc_DecryptPKCS8Key (byte * input, word32 sz, const char * password, int passwordSz) この関数は、暗号化された PKCS#8 DER キーを受け取り、PKCS#8 暗号化されていない DER に復号します。wc_EncryptPKCS8Key によって行われた暗号化を元に戻します。RFC5208 を参照してください。入力バッファは復号されたデータで上書きされます。
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) この関数は、従来の DER キーを受け取り、PKCS#8 形式に変換し、暗号化します。これを行うために wc_CreatePKCS8Key と wc_EncryptPKCS8Key を使用します。

	Name
void	wc_InitDecodedCert (struct DecodedCert * cert, const byte * source, word32 inSz, void * heap) この関数は、“cert” パラメータが指す DecodedCert を初期化します。長さ “inSz” の DER エンコードされた証明書への “source” ポインタを保存します。この証明書は、wc_ParseCert への後続の呼び出しによって解析できます。
int	wc_ParseCert (DecodedCert * cert, int type, int verify, void * cm) この関数は、DecodedCert オブジェクトに保存された DER エンコードされた証明書を解析し、そのオブジェクトのフィールドを設定します。DecodedCert は、wc_InitDecodedCert への事前の呼び出しで初期化されている必要があります。この関数は、CertificateManager オブジェクトへのオプションのポインタを取り、CA が CertificateManager で見つかった場合、DecodedCert の証明機関情報を設定するために使用されます。
void	wc_FreeDecodedCert (struct DecodedCert * cert) この関数は、wc_InitDecodedCert で以前に初期化された DecodedCert を解放します。
int	wc_SetTimeCb (wc_time_cb f) この関数は、wolfSSL が現在時刻を取得する必要があるときに使用される時刻コールバックを登録します。コールバックのプロトタイプは、C 標準ライブラリの “time” 関数と同じである必要があります。
time_t	wc_Time (time_t * t) この関数は、現在時刻を取得します。デフォルトでは、プラットフォーム間で異なる XTIME マクロを使用します。ユーザーは、wc_SetTimeCb 関数を介して任意の関数を使用できます。
int	wc_SetCustomExtension (Cert * cert, int critical, const char * oid, const byte * der, word32 derSz) この関数は、X.509 証明書にカスタム拡張を挿入します。注：ポインタであるパラメータのいずれかが指すアドレスのコンテンツは、証明書が生成されて der 出力が得られるまで変更してはいけません。この関数はコンテンツを別のバッファにコピーしません。
int	wc_SetUnknownExtCallback (DecodedCert * cert, wc_UnknownExtCallback cb) この関数は、wolfSSL が証明書の解析中に証明書内の不明な X.509 拡張に遭遇したときに使用されるコールバックを登録します。コールバックのプロトタイプは次のようにする必要があります：
int	wc_CheckCertSigPubKey (const byte * cert, word32 certSz, void * heap, const byte * pubKey, word32 pubKeySz, int pubKeyOID) この関数は、X.509 証明書の der 形式の署名を公開鍵に対して検証します。公開鍵は、der 形式の完全なサブジェクト公開鍵情報であることが期待されます。

	Name
int	wc_Asn1PrintOptions_Init (Asn1PrintOptions * opts) この関数は、ASN.1 プリントオプションを初期化します。
int	wc_Asn1PrintOptions_Set (Asn1PrintOptions * opts, enum Asn1PrintOpt opt, word32 val) この関数は、ASN.1 プリントオプションオブジェクトにプリントオプションを設定します。
int	wc_Asn1_Init (Asn1 * asn1) この関数は、ASN.1 解析オブジェクトを初期化します。
int	wc_Asn1_SetFile (Asn1 * asn1, XFILE file) この関数は、ASN.1 解析オブジェクトへのプリント時に使用するファイルを設定します。
int	wc_Asn1_PrintAll (Asn1 * asn1, Asn1PrintOptions * opts, unsigned char * data, word32 len) すべての ASN.1 項目をプリントします。

C.4.2 Functions Documentation

C.4.2.1 function wc_InitCert

```
int wc_InitCert(
    Cert * cert
)
```

この関数は、デフォルトのオプションでデフォルトの証明書を初期化します：version = 3 (0x2)、serial = 0、sigType = SHA_WITH_RSA、issuer = 空白、daysValid = 500、selfSigned = 1 (true) 発行者として subject を使用、subject = 空白

Parameters:

- **cert** 初期化する未初期化の cert 構造体へのポインタ

See:

- **wc_MakeCert**
- **wc_MakeCertReq**

Return: none 返回值なし。

Example

```
Cert myCert;
wc_InitCert(&myCert);
```

C.4.2.2 function wc_CertNew

```
Cert * wc_CertNew(
    void * heap
)
```

この関数は、証明書操作中に使用するための新しい Cert 構造体を割り当てます。アプリケーションが構造体自体を割り当てる必要はありません。Cert 構造体もこの関数によって初期化されるため、wc_InitCert() を呼び出す必要がなくなります。アプリケーションが割り当てられた Cert 構造体の使用を終了したら、wc_CertFree() を呼び出す必要があります。

Parameters:

- **A** 動的割り当てに使用されるヒープへのポインタ。NULL でも可。

See:

- `wc_InitCert`
- `wc_MakeCert`
- `wc_CertFree`

Return:

- pointer 成功した場合、呼び出しは新しく割り当てられ初期化された Cert へのポインタを返します。
- NULL メモリ割り当て失敗時。

Example

```
Cert*    myCert;

myCert = wc_CertNew(NULL);
if (myCert == NULL) {
    // Cert 作成失敗
}
```

C.4.2.3 function `wc_CertFree`

```
void wc_CertFree(
    Cert * cert
)
```

この関数は、`wc_CertNew()` への以前の呼び出しによって `cert` 構造体に割り当てられたメモリを解放します。

Parameters:

- **A** 解放する `cert` 構造体へのポインタ。

See:

- `wc_InitCert`
- `wc_MakeCert`
- `wc_CertNew`

Return: None.

Example

```
Cert*    myCert;

myCert = wc_CertNew(NULL);

// 証明書操作を実行。

wc_CertFree(myCert);
```

C.4.2.4 function `wc_MakeCert`

```
int wc_MakeCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
```

```

    WC_RNG * rng
)

```

CA 署名付き証明書を作成するために使用されます。subject 情報が入力された後に呼び出されます。この関数は、cert 入力から x509 証明書 v3 RSA または ECC を作成します。次に、この証明書を derBuffer に書き込みます。証明書を生成するために、rsaKey または eccKey のいずれかを受け取ります。このメソッドを呼び出す前に、証明書を wc_InitCert で初期化する必要があります。

Parameters:

- **cert** 初期化された cert 構造体へのポインタ
- **derBuffer** 生成された証明書を保持するバッファへのポインタ
- **derSz** 証明書を格納するバッファのサイズ
- **rsaKey** 証明書の生成に使用される rsa キーを含む RsaKey 構造体へのポインタ
- **eccKey** 証明書の生成に使用される ecc キーを含む EccKey 構造体へのポインタ
- **rng** 証明書を生成するために使用される乱数生成器へのポインタ

See:

- [wc_InitCert](#)
- [wc_MakeCertReq](#)

Return:

- Success 指定された入力 cert から x509 証明書を正常に作成すると、生成された証明書のサイズを返します。
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 提供された derBuffer が生成された証明書を格納するには小さすぎる場合に返されます
- Others 証明書の生成が成功しない場合、追加のエラーメッセージが返される可能性があります。

Example

```

Cert myCert;
wc_InitCert(&myCert);
WC_RNG rng;
// rng を初期化;
RsaKey key;
// key を初期化;
byte * derCert = malloc(FOURK_BUF);
word32 certSz;
certSz = wc_MakeCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);

```

C.4.2.5 function wc_MakeCertReq

```

int wc_MakeCertReq(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey
)

```

この関数は、入力証明書を使用して証明書署名要求を作成し、出力を derBuffer に書き込みます。証明書要求を生成するために、rsaKey または eccKey のいずれかを受け取ります。証明書署名要求に署名するには、この関数の後に wc_SignCert() を呼び出す必要があります。この関数の使用例については、wolfCrypt テストアプリケーション (./wolfcrypt/test/test.c) を参照してください。

Parameters:

- **cert** 初期化された cert 構造体へのポインタ

- **derBuffer** 生成された証明書要求を保持するバッファへのポインタ
- **derSz** 証明書要求を格納するバッファのサイズ
- **rsaKey** 証明書要求の生成に使用される rsa キーを含む RsaKey 構造体へのポインタ
- **eccKey** 証明書要求の生成に使用される ecc キーを含む EccKey 構造体へのポインタ

See:

- [wc_InitCert](#)
- [wc_MakeCert](#)

Return:

- Success 指定された入力 cert から X.509 証明書要求を正常に作成すると、生成された証明書要求のサイズを返します。
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 提供された derBuffer が生成された証明書を格納するには小さすぎる場合に返されます
- Other 証明書要求の生成が成功しない場合、追加のエラーメッセージが返される可能性があります。

Example

```
Cert myCert;
// myCert を初期化
EccKey key;
// key を初期化;
byte* derCert = (byte*)malloc(FOURK_BUF);

word32 certSz;
certSz = wc_MakeCertReq(&myCert, derCert, FOURK_BUF, NULL, &key);
```

C.4.2.6 function wc_SignCert

```
int wc_SignCert(
    int requestSz,
    int sigType,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

この関数は buffer に署名し、署名を buffer の末尾に追加します。署名タイプを受け取ります。CA 署名付き証明書を作成する場合は、wc_MakeCert() または wc_MakeCertReq() の後に呼び出す必要があります。

Parameters:

- **requestSz** 署名を要求している証明書本文のサイズ
- **sigType** 作成する署名のタイプ。有効なオプションは：CTC_MD5wRSA、CTC_SHAwRSA、CTC_SHAwECDSA、CTC_SHA256wECDSA、および CTC_SHA256wRSA
- **buffer** 署名される証明書を含むバッファへのポインタ。成功時：新しく署名された証明書を保持します
- **buffSz** 新しく署名された証明書を格納するバッファの（合計）サイズ
- **rsaKey** 証明書に署名するために使用される rsa キーを含む RsaKey 構造体へのポインタ
- **eccKey** 証明書に署名するために使用される ecc キーを含む EccKey 構造体へのポインタ
- **rng** 証明書に署名するために使用される乱数生成器へのポインタ

See:

- [wc_InitCert](#)
- [wc_MakeCert](#)

Return:

- Success 証明書の署名に成功すると、証明書の新しいサイズ（署名を含む）を返します。
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 提供されたバッファが生成された証明書を格納するには小さすぎる場合に返されます
- Other 証明書の生成が成功しない場合、追加のエラーメッセージが返される可能性があります。

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// myCert、derCert を初期化
RsaKey key;
// key を初期化;
WC_RNG rng;
// rng を初期化

word32 certSz;
certSz = wc_SignCert(myCert.bodySz, myCert.sigType, derCert, FOURK_BUF,
&key, NULL,
&rng);
```

C.4.2.7 function wc_MakeSelfCert

```
int wc_MakeSelfCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * key,
    WC_RNG * rng
)
```

この関数は、自己署名用の前の 2 つの関数、wc_MakeCert と wc_SignCert の組み合わせです（前の関数は CA 要求に使用できます）。証明書を作成してから署名し、自己署名証明書を生成します。

Parameters:

- **cert** 作成して署名する証明書へのポインタ
- **buffer** 署名された証明書を保持するバッファへのポインタ
- **buffSz** 署名された証明書を格納するバッファのサイズ
- **key** 証明書に署名するために使用される rsa キーを含む RsaKey 構造体へのポインタ
- **rng** 証明書の生成と署名に使用される乱数生成器へのポインタ

See:

- [wc_InitCert](#)
- [wc_MakeCert](#)
- [wc_SignCert](#)

Return:

- Success 証明書の署名に成功すると、証明書の新しいサイズを返します。
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 提供されたバッファが生成された証明書を格納するには小さすぎる場合に返されます
- Other 証明書の生成が成功しない場合、追加のエラーメッセージが返される可能性があります。

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
```

```
// myCert、derCert を初期化
RsaKey key;
// key を初期化;
WC_RNG rng;
// rng を初期化

word32 certSz;
certSz = wc_MakeSelfCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

C.4.2.8 function wc_SetIssuer

```
int wc_SetIssuer(
    Cert * cert,
    const char * issuerFile
)
```

この関数は、証明書の発行者を、提供された pem issuerFile の発行者に設定します。また、証明書の自己署名属性を false に変更します。issuerFile で指定された発行者は、cert 発行者を設定する前に検証されます。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 発行者を設定する証明書へのポインタ
- **issuerFile** pem 形式の証明書を含むファイルのパス

See:

- [wc_InitCert](#)
- [wc_SetSubject](#)
- [wc_SetIssuerBuffer](#)

Return:

- 0 証明書の発行者を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
if(wc_SetIssuer(&myCert, "../path/to/ca-cert.pem") != 0) {
```

```
// 発行者設定エラー
}
```

C.4.2.9 function wc_SetSubject

```
int wc_SetSubject(
    Cert * cert,
    const char * subjectFile
)
```

この関数は、証明書の subject を、提供された pem subjectFile の subject に設定します。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 発行者を設定する証明書へのポインタ
- **subjectFile** pem 形式の証明書を含むファイルのパス

See:

- [wc_InitCert](#)
- [wc_SetIssuer](#)

Return:

- 0 証明書の発行者を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
    // subject 設定エラー
}
```

C.4.2.10 function wc_SetSubjectRaw

```
int wc_SetSubjectRaw(
    Cert * cert,
    const byte * der,
```

```
int derSz
)
```

この関数は、提供された der バッファの subject から証明書の生の subject を設定します。このメソッドは、署名前に生の subject フィールドを設定するために使用されます。

Parameters:

- **cert** 生の subject を設定する証明書へのポインタ
- **der** subject を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** subject を取得する der 形式の証明書を含むバッファのサイズ

See:

- [wc_InitCert](#)
- [wc_SetSubject](#)

Return:

- 0 証明書の subject を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);
// der を初期化
if(wc_SetSubjectRaw(&myCert, der, FOURK_BUF) != 0) {
    // subject 設定エラー
}
```

C.4.2.11 function wc_GetSubjectRaw

```
int wc_GetSubjectRaw(
    byte ** subjectRaw,
    Cert * cert
)
```

この関数は、証明書構造体から生の subject を取得します。

Parameters:

- **subjectRaw** 正常に返された場合の生の subject へのポインタのポインタ
- **cert** 生の subject を取得する証明書へのポインタ

See:

- [wc_InitCert](#)
- [wc_SetSubjectRaw](#)

Return:

- 0 証明書から subject を正常に取得した場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます

Example

```
Cert myCert;
byte *subjRaw;
// myCert を初期化

if(wc_GetSubjectRaw(&subjRaw, &myCert) != 0) {
    // subject 設定エラー
}
```

C.4.2.12 function wc_SetAltNames

```
int wc_SetAltNames(
    Cert * cert,
    const char * file
)
```

この関数は、証明書の代替名を、提供された pem ファイル内の代替名に設定します。これは、同じ証明書で複数のドメインを保護したい場合に便利です。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 代替名を設定する証明書へのポインタ
- **file** pem 形式の証明書を含むファイルのパス

See:

- [wc_InitCert](#)
- [wc_SetIssuer](#)

Return:

- 0 証明書の代替名を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます

- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
if(wc_SetSubject(&myCert, ".\\path\\to\\ca-cert.pem") != 0) {
    // 代替名設定エラー
}
```

C.4.2.13 function wc_SetIssuerBuffer

```
int wc_SetIssuerBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は、提供された der バッファの発行者から証明書の発行者を設定します。また、証明書の自己署名属性を false に変更します。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 発行者を設定する証明書へのポインタ
- **der** 発行者を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** 発行者を取得する der 形式の証明書を含むバッファのサイズ

See:

- [wc_InitCert](#)
- [wc_SetIssuer](#)

Return:

- 0 証明書の発行者を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```

Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);
// der を初期化
if(wc_SetIssuerBuffer(&myCert, der, FOURK_BUF) != 0) {
    // issuer 設定エラー
}

```

C.4.2.14 function wc_SetIssuerRaw

```

int wc_SetIssuerRaw(
    Cert * cert,
    const byte * der,
    int derSz
)

```

この関数は、提供された der バッファの発行者から証明書の生の発行者を設定します。このメソッドは、署名前に生の発行者フィールドを設定するために使用されます。

Parameters:

- **cert** 生の発行者を設定する証明書へのポインタ
- **der** subject を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** subject を取得する der 形式の証明書を含むバッファのサイズ

See:

- [wc_InitCert](#)
- [wc_SetIssuer](#)

Return:

- 0 証明書の発行者を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```

Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);

```



```
// der を初期化
if(wc_SetIssuerRaw(&myCert, der, FOURK_BUF) != 0) {
    // subject 設定エラー
}
```

C.4.2.15 function wc_SetSubjectBuffer

```
int wc_SetSubjectBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は、提供された der バッファの subject から証明書の subject を設定します。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** subject を設定する証明書へのポインタ
- **der** subject を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** subject を取得する der 形式の証明書を含むバッファのサイズ

See:

- [wc_InitCert](#)
- [wc_SetSubject](#)

Return:

- 0 証明書の subject を正常に設定した場合に返されます
- MEMORY_XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキョブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);
// der を初期化
if(wc_SetSubjectBuffer(&myCert, der, FOURK_BUF) != 0) {
    // subject 設定エラー
}
```

C.4.2.16 function wc_SetAltNamesBuffer

```
int wc_SetAltNamesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は、提供された der バッファの代替名から証明書の代替名を設定します。これは、同じ証明書で複数のドメインを保護したい場合に便利です。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 代替名を設定する証明書へのポインタ
- **der** 代替名を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** 代替名を取得する der 形式の証明書を含むバッファのサイズ

See:

- [wc_InitCert](#)
- [wc_SetAltNames](#)

Return:

- 0 証明書の代替名を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);
// der を初期化
if(wc_SetAltNamesBuffer(&myCert, der, FOURK_BUF) != 0) {
    // subject 設定エラー
}
```

C.4.2.17 function wc_SetDatesBuffer

```
int wc_SetDatesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は、提供された der バッファの日付範囲から証明書の日付を設定します。このメソッドは、署名前にフィールドを設定するために使用されます。

Parameters:

- **cert** 日付を設定する証明書へのポインタ
- **der** 日付範囲を取得する der 形式の証明書を含むバッファへのポインタ
- **derSz** 日付範囲を取得する der 形式の証明書を含むバッファのサイズ

See: [wc_InitCert](#)

Return:

- 0 証明書の日付を正常に設定した場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_PARSE_E 証明書ヘッダーファイルの解析エラーがある場合に返されます
- ASN_OBJECT_ID_E 証明書から暗号化タイプを解析する際にエラーがある場合に返されます
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます
- ASN_BEFORE_DATE_E 日付が証明書の開始日より前の場合に返されます
- ASN_AFTER_DATE_E 日付が証明書の有効期限より後の場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析する際にエラーがある場合に返されます
- ECC_CURVE_OID_E 証明書から ECC キーを解析する際にエラーがある場合に返されます
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます
- ASN_CRIT_EXT_E 証明書の処理中に見慣れない重要な拡張に遭遇した場合に返されます
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます
- ASN_SIG_CONFIRM_E 証明書署名の確認が失敗した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約で許可されていない場合に返されます
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます

Example

```
Cert myCert;
// myCert を初期化
byte* der;
der = (byte*)malloc(FOURK_BUF);
// der を初期化
if(wc_SetDatesBuffer(&myCert, der, FOURK_BUF) != 0) {
    // subject 設定エラー
}
```

C.4.2.18 function wc_SetAuthKeyIdFromPublicKey

```
int wc_SetAuthKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsaKey,
    ecc_key * eckey
)
```

RSA または ECC 公開鍵から AKID を設定します。注：rsakey または eckey のいずれか一方のみを設定し、両方は設定しないでください。

Parameters:

- **cert** SKID を設定する証明書へのポインタ。
- **rsakey** 読み取り元の RsaKey 構造体へのポインタ。
- **eckey** 読み取り元の ecc_key へのポインタ。

See:

- [wc_SetSubjectKeyId](#)
- [wc_SetAuthKeyId](#)
- [wc_SetAuthKeyIdFromCert](#)

Return:

- 0 成功
- BAD_FUNC_ARG cert が null または rsakey と eckey の両方が null の場合。
- MEMORY_E メモリの割り当てエラー。
- PUBLIC_KEY_E キーへの書き込みエラー。

Example

```
Cert myCert;
RsaKey keypub;

wc_InitRsaKey(&keypub, 0);

if (wc_SetAuthKeyIdFromPublicKey(&myCert, &keypub, NULL) != 0)
{
    // エラーを処理
}
```

C.4.2.19 function wc_SetAuthKeyIdFromCert

```
int wc_SetAuthKeyIdFromCert(
    Cert * cert,
    const byte * der,
    int derSz
)
```

DER エンコードされた証明書から AKID を設定します。

Parameters:

- **cert** 書き込み先の Cert 構造体。
- **der** DER エンコードされた証明書バッファ。
- **derSz** der のサイズ (バイト単位)。

See:

- [wc_SetAuthKeyIdFromPublicKey](#)
- [wc_SetAuthKeyId](#)

Return:

- 0 成功
- BAD_FUNC_ARG いずれかの引数が null または derSz が 0 未満の場合のエラー。
- MEMORY_E メモリの割り当てに問題がある場合のエラー。
- ASN_NO_SKID サブジェクトキー ID が見つかりません。

Example

```

Cert some_cert;
byte some_der[] = { // DER バッファを初期化 };
wc_InitCert(&some_cert);
if(wc_SetAuthKeyIdFromCert(&some_cert, some_der, sizeof(some_der) != 0)
{
    // エラーを処理
}

```

C.4.2.20 function wc_SetAuthKeyId

```

int wc_SetAuthKeyId(
    Cert * cert,
    const char * file
)

```

PEM 形式の証明書ファイルから AKID を設定します。

Parameters:

- **cert** AKID を設定したい Cert 構造体。
- **file** PEM 証明書ファイルを含むバッファ。

See:

- [wc_SetAuthKeyIdFromPublicKey](#)
- [wc_SetAuthKeyIdFromCert](#)

Return:

- 0 成功
- BAD_FUNC_ARG cert または file が null の場合のエラー。
- MEMORY_E メモリの割り当てに問題がある場合のエラー。

Example

```

char* file_name = "/path/to/file";
cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetAuthKeyId(&some_cert, file_name) != 0)
{
    // エラーを処理
}

```

C.4.2.21 function wc_SetSubjectKeyIdFromPublicKey

```

int wc_SetSubjectKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsa_key,
    ecc_key * eckey
)

```

RSA または ECC 公開鍵から SKID を設定します。

Parameters:

- **cert** 使用する Cert 構造体へのポインタ。
- **rsa_key** RsaKey 構造体へのポインタ
- **eckey** ecc_key 構造体へのポインタ

See: [wc_SetSubjectKeyId](#)

Return:

- 0 成功
- BAD_FUNC_ARG cert または rsakey と eckey が null の場合に返されます。
- MEMORY_E メモリの割り当てエラーがある場合に返されます。
- PUBLIC_KEY_E 公開鍵の取得エラーがある場合に返されます。

Example

```
Cert some_cert;
RsaKey some_key;
wc_InitCert(&some_cert);
wc_InitRsaKey(&some_key);

if(wc_SetSubjectKeyIdFromPublicKey(&some_cert,&some_key, NULL) != 0)
{
    // エラーを処理
}
```

C.4.2.22 function wc_SetSubjectKeyId

```
int wc_SetSubjectKeyId(
    Cert * cert,
    const char * file
)
```

PEM 形式の公開鍵ファイルから SKID を設定します。両方の引数が必要です。

Parameters:

- **cert** SKID を設定する Cert 構造体。
- **file** PEM エンコードされたファイルを含む。

See: [wc_SetSubjectKeyIdFromPublicKey](#)

Return:

- 0 成功
- BAD_FUNC_ARG cert または file が null の場合に返されます。
- MEMORY_E キー用のメモリ割り当てに問題がある場合に返されます。
- PUBLIC_KEY_E 公開鍵のデコードエラーがある場合に返されます。

Example

```
const char* file_name = "path/to/file";
Cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetSubjectKeyId(&some_cert, file_name) != 0)
{
    // エラーを処理
}
```

C.4.2.23 function wc_SetKeyUsage

```
int wc_SetKeyUsage(
    Cert * cert,
```

```
    const char * value
)
```

この関数を使用すると、カンマ区切りのトークン文字列を使用してキー使用法を設定できます。受け入れられるトークンは：digitalSignature、nonRepudiation、contentCommitment、keyCertSign、cRLSign、dataEncipherment、keyAgreement、keyEncipherment、encipherOnly、decipherOnly です。例：“digitalSignature,nonRepudiation” nonRepudiation と contentCommitment は同じ用途です。

Parameters:

- **cert** 初期化された Cert 構造体へのポインタ。
- **value** 使用法を設定するトークンのカンマ区切り文字列。

See:

- [wc_InitCert](#)
- [wc_MakeRsaKey](#)

Return:

- 0 成功
- BAD_FUNC_ARG いずれかの引数が null の場合に返されます。
- MEMORY_E メモリの割り当てエラーがある場合に返されます。
- KEYUSAGE_E 認識されないトークンが入力された場合に返されます。

Example

```
Cert cert;
wc_InitCert(&cert);

if(wc_SetKeyUsage(&cert, "cRLSign,keyCertSign") != 0)
{
    // エラーを処理
}
```

C.4.2.24 function wc_PemPubKeyToDer

```
int wc_PemPubKeyToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

ファイルから PEM キーを読み込み、DER エンコードされたバッファに変換します。

Parameters:

- **fileName** ロードするファイルの名前。
- **derBuf** DER エンコードされたキー用のバッファ。
- **derSz** DER バッファのサイズ。

See: [wc_PubKeyPemToDer](#)

Return:

- 0 成功
- <0 エラー
- SSL_BAD_FILE ファイルを開く際に問題があります。
- MEMORY_E ファイルバッファ用のメモリ割り当てエラーがあります。
- BUFFER_E derBuf が変換されたキーを保持するのに十分な大きさではありません。

Example

```
char* some_file = "filename";
unsigned char der[];

if(wc_PemPubKeyToDer(some_file, der, sizeof(der)) != 0)
{
    // エラーを処理
}
```

C.4.2.25 function wc_PubKeyPemToDer

```
int wc_PubKeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz
)
```

PEM エンコードされた公開鍵を DER に変換します。バッファに書き込まれたバイト数、またはエラーの場合は負の値を返します。

Parameters:

- **pem** PEM エンコードされたキー
- **pemSz** pem のサイズ
- **buff** 出力用バッファへのポインタ。
- **buffSz** バッファのサイズ。

See: [wc_PemPubKeyToDer](#)

Return:

- 0 成功、書き込まれたバイト数。
- BAD_FUNC_ARG pem、buff、または buffSz が null の場合に返されます
- <0 関数内でエラーが発生しました。

Example

```
byte some_pem[] = { PEM キーで初期化 }
unsigned char out_buffer[1024]; // バッファが DER を格納するのに十分な大きさであること
    ⇨ を確認
```

```
if(wc_PubKeyPemToDer(some_pem, sizeof(some_pem), out_buffer,
    sizeof(out_buffer)) < 0)
{
    // エラーを処理
}
```

C.4.2.26 function wc_PemCertToDer

```
int wc_PemCertToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

この関数は、pem 証明書を der 証明書に変換し、結果の証明書を提供された derBuf バッファに配置します。

Parameters:

- **fileName** der 証明書に変換する pem 証明書を含むファイルへのパス
- **derBuf** 変換された証明書を格納する char バッファへのポインタ
- **derSz** 変換された証明書を格納する char バッファのサイズ

See: なし**Return:**

- Success 成功時に生成された derBuf のサイズを返します
- BUFFER_E derBuf のサイズが生成された証明書を保持するには小さすぎる場合に返されます
- MEMORY_E XMALLOC の呼び出しが失敗した場合に返されます

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 エラー
}
```

C.4.2.27 function wc_DerToPem

```
int wc_DerToPem(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outSz,
    int type
)
```

この関数は、der バッファに含まれる der 形式の入力証明書を、output バッファに含まれる pem 形式の出力証明書に変換します。これはインプレース変換ではなく、pem 形式の出力を格納するために別のバッファを使用する必要があることに注意してください。

Parameters:

- **der** 変換する証明書のバッファへのポインタ
- **derSz** 変換する証明書のサイズ
- **output** pem 形式の証明書を格納するバッファへのポインタ
- **outSz** pem 形式の証明書を格納するバッファのサイズ
- **type** 生成する証明書のタイプ。有効なタイプは：CERT_TYPE、PRIVATEKEY_TYPE、ECC_PRIVATEKEY_TYPE、および CERTREQ_TYPE。

See: [wc_PemCertToDer](#)**Return:**

- Success 入力 der 証明書から正常に pem 証明書を作成すると、生成された pem 証明書のサイズを返します。
- BAD_FUNC_ARG der ファイルを解析して pem ファイルとして格納する際にエラーがある場合に返されます
- MEMORY_E XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_INPUT_E base64 エンコードエラーの場合に返されます
- BUFFER_E 出力バッファが pem 形式の証明書を格納するには小さすぎる場合に返される可能性があります

Example

```
byte* der;
// 証明書で der を初期化
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
pemSz = wc_DerToPem(der, derSz, pemFormatted, FOURK_BUF, CERT_TYPE);
```

C.4.2.28 function wc_DerToPemEx

```
int wc_DerToPemEx(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outSz,
    byte * cipher_info,
    int type
)
```

この関数は、der 形式の入力証明書を変換します、der バッファに含まれる、pem 形式の出力証明書に変換し、output バッファに格納します。これはインプレース変換ではなく、pem 形式の出力を格納するために別のバッファを使用する必要があることに注意してください。暗号情報の設定を許可します。

Parameters:

- **der** 変換する証明書のバッファへのポインタ
- **derSz** 変換する証明書のサイズ
- **output** pem 形式の証明書を格納するバッファへのポインタ
- **outSz** pem 形式の証明書を格納するバッファのサイズ
- **cipher_info** 追加の暗号情報。
- **type** 生成する証明書のタイプ。有効なタイプは：CERT_TYPE、PRIVATEKEY_TYPE、ECC_PRIVATEKEY_TYPE、および CERTREQ_TYPE。

See: [wc_PemCertToDer](#)

Return:

- Success 入力 der 証明書から正常に pem 証明書を作成すると、生成された pem 証明書のサイズを返します。
- BAD_FUNC_ARG der ファイルを解析して pem ファイルとして格納する際にエラーがある場合に返されます
- MEMORY_E XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- ASN_INPUT_E base64 エンコードエラーの場合に返されます
- BUFFER_E 出力バッファが pem 形式の証明書を格納するには小さすぎる場合に返される可能性があります

Example

```
byte* der;
// 証明書で der を初期化
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
byte* cipher_info[] { 追加の暗号情報。 }
pemSz = wc_DerToPemEx(der, derSz, pemFormatted, FOURK_BUF, cipher_info,
    ↪ CERT_TYPE);
```

C.4.2.29 function wc_KeyPemToDer

```
int wc_KeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    const char * pass
)
```

PEM 形式のキーを DER 形式に変換します。

Parameters:

- **pem** PEM エンコードされた証明書へのポインタ。
- **pemSz** PEM バッファ (pem) のサイズ
- **buff** DerBuffer 構造体の buffer メンバーのコピーへのポインタ。
- **buffSz** DerBuffer 構造体に割り当てられたバッファスペースのサイズ。
- **pass** 関数に渡されるパスワード。

See: wc_PemToDer

Return:

- int 関数は正常実行時にバッファに書き込まれたバイト数を返します。
- int エラーを示す負の整数が返されます。

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){
    // バイトがバッファに書き込まれました。
}
```

C.4.2.30 function wc_CertPemToDer

```
int wc_CertPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    int type
)
```

この関数は、PEM 形式の証明書を DER 形式に変換します。OpenSSL 関数 PemToDer を呼び出します。

Parameters:

- **pem** PEM 形式の証明書へのポインタ。
- **pemSz** 証明書のサイズ。
- **buff** DER 形式にコピーされるバッファ。

- **buffSz** バッファのサイズ。
- **type** [asn_public.h](#) enum CertType にある証明書ファイルタイプ。

See: [wc_PemToDer](#)

Return: buffer バッファに書き込まれたバイトを返します。

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) {
    // バッファにバイトが書き込まれました
}
```

C.4.2.31 function [wc_GetPubKeyDerFromCert](#)

```
int wc_GetPubKeyDerFromCert(
    struct DecodedCert * cert,
    byte * derKey,
    word32 * derKeySz
)
```

この関数は、入力された DecodedCert 構造体から DER 形式の公開鍵を取得します。この API を呼び出す前に、ユーザーは [wc_InitDecodedCert\(\)](#) と [wc_ParseCert\(\)](#) を呼び出す必要があります。[wc_InitDecodedCert\(\)](#) は DER/ASN.1 エンコードされた証明書を受け入れます。PEM 証明書を DER に変換するには、[wc_InitDecodedCert\(\)](#) を呼び出す前にまず [wc_CertPemToDer\(\)](#) を使用してください。

Parameters:

- **cert** X.509 証明書を保持する入力された DecodedCert 構造体
- **derKey** DER エンコードされた公開鍵を配置する出力バッファ
- **derKeySz** [IN/OUT] 入力時の derKey バッファのサイズ、返却時の公開鍵のサイズ。derKey が NULL として渡された場合、derKeySz は公開鍵に必要なバッファサイズに設定され、関数から LENGTH_ONLY_E が返されます。

See: [wc_GetPubKeyDerFromCert](#)

Return: 0 成功時、エラー時は負の値。derKey が NULL で長さのみを返す場合は LENGTH_ONLY_E。

C.4.2.32 function [wc_EccPrivateKeyDecode](#)

```
int wc_EccPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

この関数は、入力バッファ input から ECC 秘密鍵を読み取り、秘密鍵を解析し、それを使用して ecc_key オブジェクトを生成し、key に格納します。

Parameters:

- **input** 入力秘密鍵を含むバッファへのポインタ
- **inOutIdx** バッファ内で開始するインデックスを含む word32 オブジェクトへのポインタ

- **key** デコードされた秘密鍵を格納する初期化された ecc オブジェクトへのポインタ
- **inSz** 秘密鍵を含む入力バッファのサイズ

See: wc_RSA_PrivateKeyDecode

Return:

- 0 秘密鍵のデコードに成功し、結果を ecc_key 構造体に格納した場合
- ASN_PARSE_E der ファイルを解析して pem ファイルとして格納する際にエラーがある場合に返されます
- MEMORY_E XMMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 変換する証明書が指定された最大証明書サイズより大きい場合に返されます
- ASN_OBJECT_ID_E 証明書エンコーディングに無効なオブジェクト ID がある場合に返されます
- ECC_CURVE_OID_E 提供されたキーの ECC 曲線がサポートされていない場合に返されます
- ECC_BAD_ARG_E ECC キー形式にエラーがある場合に返されます
- NOT_COMPILED_IN 秘密鍵が圧縮されており、圧縮キーが提供されていない場合に返されます
- MP_MEM 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます
- MP_VAL 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます
- MP_RANGE 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます

Example

```
int ret, idx=0;
ecc_key key; // キーを格納

byte* tmp; // キーを読み取るための一時バッファ
tmp = (byte*) malloc(FOURK_BUF);

int inSz;
inSz = fread(tmp, 1, FOURK_BUF, privateKeyFile);
// tmp バッファにキーを読み取る

wc_ecc_init(&key); // キーを初期化
ret = wc_EccPrivateKeyDecode(tmp, &idx, &key, (word32)inSz);
if(ret < 0) {
    // ecc キーのデコードエラー
}
```

C.4.2.33 function wc_EccKeyToDer

```
int wc_EccKeyToDer(
    ecc_key * key,
    byte * output,
    word32 inLen
)
```

この関数は、秘密 ECC キーを der 形式に書き込みます。

Parameters:

- **key** 入力 ecc キーを含むバッファへのポインタ
- **output** der 形式のキーを格納するバッファへのポインタ
- **inLen** der 形式のキーを格納するバッファの長さ

See: wc_RsaKeyToDer

Return:

- Success ECC キーの der 形式への書き込みに成功すると、バッファに書き込まれた長さを返します
- BAD_FUNC_ARG key または output が null、または inLen がゼロの場合に返されます

- MEMORY_E XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- BUFFER_E 変換された証明書が出力バッファに格納するには大きすぎる場合に返されます
- ASN_UNKNOWN_OID_E 使用されている ECC キーが不明なタイプの場合に返されます
- MP_MEM 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます
- MP_VAL 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます
- MP_RANGE 秘密鍵の解析中に使用される数学ライブラリにエラーがある場合に返されます

Example

```
int derSz;
ecc_key key;
// キーを初期化して作成
byte der[FOURK_BUF];
// ここに der 形式のキーを格納

derSz = wc_EccKeyToDer(&key, der, FOURK_BUF);
if(derSz < 0) {
    // ecc キーを der バッファに変換する際のエラー
}
```

C.4.2.34 function wc_EccPublicKeyDecode

```
int wc_EccPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

入力バッファから ECC 公開鍵をデコードします。ECC キーを取得するために ASN シーケンスを解析します。

Parameters:

- **input** デコードする DER エンコードされたキーを含むバッファ。
- **inOutIdx** 入力バッファの読み取り開始位置のインデックス。出力時、インデックスは入力バッファの最後に解析された位置に設定されます。
- **key** 公開鍵を格納する ecc_key 構造体へのポインタ。
- **inSz** 入力バッファのサイズ。

See: [wc_ecc_import_x963](#)

Return:

- 0 成功
- BAD_FUNC_ARG いずれかの引数が null の場合に返されます。
- ASN_PARSE_E 解析エラーがある場合に返されます
- ASN_ECC_KEY_E キーのインポートエラーがある場合に返されます。考えられる理由については [wc_ecc_import_x963](#) を参照してください。

Example

```
int ret;
word32 idx = 0;
byte buff[] = { // キーで初期化 };
ecc_key pubKey;
wc_ecc_init(&pubKey);
if ( wc_EccPublicKeyDecode(buff, &idx, &pubKey, sizeof(buff)) != 0) {
```

```

        // キーのデコードエラー
    }

```

C.4.2.35 function wc_EccPublicKeyToDer

```

int wc_EccPublicKeyToDer(
    ecc_key * key,
    byte * output,
    word32 inLen,
    int with_AlgCurve
)

```

この関数は、ECC 公開鍵を DER 形式に変換します。使用されたバッファのサイズを返します。DER 形式の ECC 公開鍵は出力バッファに格納されます。with_AlgCurve フラグは、アルゴリズムと曲線情報を持つヘッダーを含めます

Parameters:

- **key** ECC キーへのポインタ
- **output** 書き込み先の出力バッファへのポインタ。
- **inLen** バッファのサイズ。
- **with_AlgCurve** アルゴリズムと曲線情報を持つヘッダーを含めるタイミングのフラグ。

See:

- [wc_EccKeyToDer](#)
- [wc_EccPrivateKeyDecode](#)

Return:

- 0 成功、使用されたバッファのサイズ
- BAD_FUNC_ARG output または key が null の場合に返されます。
- LENGTH_ONLY_E ECC 公開鍵のサイズ取得エラー。
- BUFFER_E 出力バッファが小さすぎる場合に返されます。

Example

```

ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // der 用の適切なサイズ;
byte der[derSz];

if(wc_EccPublicKeyToDer(&key, der, derSz, 1) < 0)
{
    // ECC 公開鍵の der 変換エラー
}

```

C.4.2.36 function wc_EccPublicKeyToDer_ex

```

int wc_EccPublicKeyToDer_ex(
    ecc_key * key,
    byte * output,
    word32 inLen,
    int with_AlgCurve,

```

```
    int comp
)
```

この関数は、ECC 公開鍵を DER 形式に変換します。使用されたバッファのサイズを返します。DER 形式の ECC 公開鍵は出力バッファに格納されます。with_AlgCurve フラグは、アルゴリズムと曲線情報を持つヘッダーを含めます。comp パラメータは、公開鍵を圧縮形式でエクスポートするかどうかを決定します。

Parameters:

- **key** ECC キーへのポインタ
- **output** 書き込み先の出力バッファへのポインタ。
- **inLen** バッファのサイズ。
- **with_AlgCurve** アルゴリズムと曲線情報を持つヘッダーを含めるタイミングのフラグ。
- **comp** 1 (ゼロ以外) の場合、ECC 公開鍵は圧縮形式で書き込まれます。0 の場合、非圧縮形式で書き込まれます。

See:

- [wc_EccKeyToDer](#)
- [wc_EccPublicKeyDecode](#)

Return:

- 0 成功、使用されたバッファのサイズ
- BAD_FUNC_ARG output または key が null の場合に返されます。
- LENGTH_ONLY_E ECC 公開鍵のサイズ取得エラー。
- BUFFER_E 出力バッファが小さすぎる場合に返されます。

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // der 用の適切なサイズ;
byte der[derSz];

// 圧縮された ECC キーを書き出す
if(wc_EccPublicKeyToDer_ex(&key, der, derSz, 1, 1) < 0)
{
    // ECC 公開鍵の der 変換エラー
}
```

C.4.2.37 function wc_Curve25519PrivateKeyDecode

```
int wc_Curve25519PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

この関数は、DER エンコードされたバッファから Curve25519 秘密鍵（のみ）をデコードします

Parameters:

- **input** DER エンコードされた秘密鍵を含むバッファへのポインタ

- **inOutIdx** 入力バッファの読み取り開始位置のインデックス。出力時、インデックスは入力バッファの最後に解析された位置に設定されます。
- **key** デコードされたキーを格納する curve25519_key 構造体へのポインタ
- **inSz** 入力 DER バッファのサイズ

See:

- [wc_Curve25519KeyDecode](#)
- [wc_Curve25519PublicKeyDecode](#)

Return:

- 0 成功
- BAD_FUNC_ARG input、inOutIdx または key が null の場合に返されます
- ASN_PARSE_E DER エンコードされたデータの解析エラーがある場合に返されます
- ECC_BAD_ARG_E キー長が CURVE25519_KEYSIZE でない場合、または DER キーが適切にフォーマットされているにもかかわらず他の問題を含む場合に返されます。
- BUFFER_E 入力バッファが有効な DER エンコードされたキーを含むには小さすぎる場合に返されます。

Example

```
byte der[] = { // DER エンコードされたキー };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);

if (wc_Curve25519PrivateKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // 秘密鍵のデコードエラー
}
```

C.4.2.38 function wc_Curve25519PublicKeyDecode

```
int wc_Curve25519PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

この関数は、DER エンコードされたバッファから Curve25519 公開鍵（のみ）をデコードします。

Parameters:

- **input** DER エンコードされた公開鍵を含むバッファへのポインタ
- **inOutIdx** 入力バッファの読み取り開始位置のインデックス。出力時、インデックスは入力バッファの最後に解析された位置に設定されます。
- **key** デコードされたキーを格納する curve25519_key 構造体へのポインタ
- **inSz** 入力 DER バッファのサイズ

See:

- [wc_Curve25519KeyDecode](#)
- [wc_Curve25519PrivateKeyDecode](#)

Return:

- 0 成功
- BAD_FUNC_ARG input、inOutIdx または key が null の場合に返されます
- ASN_PARSE_E DER エンコードされたデータの解析エラーがある場合に返されます

- ECC_BAD_ARG_E キー長が CURVE25519_KEYSIZE でない場合、または DER キーが適切にフォーマットされているにもかかわらず他の問題を含む場合に返されます。
- BUFFER_E 入力バッファが有効な DER エンコードされたキーを含むには小さすぎる場合に返されます。

Example

```
byte der[] = { // DER エンコードされたキー };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);
if (wc_Curve25519PublicKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // 公開鍵のデコードエラー
}
```

C.4.2.39 function wc_Curve25519KeyDecode

```
int wc_Curve25519KeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

この関数は、DER エンコードされたバッファから Curve25519 キーをデコードします。秘密鍵、公開鍵、または両方をデコードできます。

Parameters:

- **input** DER エンコードされたキーを含むバッファへのポインタ
- **inOutIdx** 入力バッファの読み取り開始位置のインデックス。出力時、インデックスは入力バッファの最後に解析された位置に設定されます。
- **key** デコードされたキーを格納する curve25519_key 構造体へのポインタ
- **inSz** 入力 DER バッファのサイズ

See:

- [wc_Curve25519PrivateKeyDecode](#)
- [wc_Curve25519PublicKeyDecode](#)

Return:

- 0 成功
- BAD_FUNC_ARG input、inOutIdx または key が null の場合に返されます
- ASN_PARSE_E DER エンコードされたデータの解析エラーがある場合に返されます
- ECC_BAD_ARG_E キー長が CURVE25519_KEYSIZE でない場合、または DER キーが適切にフォーマットされているにもかかわらず他の問題を含む場合に返されます。
- BUFFER_E 入力バッファが有効な DER エンコードされたキーを含むには小さすぎる場合に返されます。

Example

```
byte der[] = { // DER エンコードされたキー };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);
if (wc_Curve25519KeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // キーのデコードエラー
}
```

C.4.2.40 function wc_Curve25519PrivateKeyToDer

```
int wc_Curve25519PrivateKeyToDer(  
    curve25519_key * key,  
    byte * output,  
    word32 inLen  
)
```

この関数は、Curve25519 秘密鍵を DER 形式にエンコードします。入力キー構造体に公開鍵が含まれている場合、それは無視されます。

Parameters:

- **key** エンコードする秘密鍵を含む curve25519_key 構造体へのポインタ
- **output** DER エンコーディングを保持するバッファ
- **inLen** 出力バッファのサイズ

See:

- [wc_Curve25519KeyToDer](#)
- [wc_Curve25519PublicKeyToDer](#)

Return:

- 0 成功、DER エンコーディングの長さ
- BAD_FUNC_ARG key または output が null の場合に返されます
- MEMORY_E 割り当て失敗がある場合に返されます
- BUFFER_E 出力バッファが小さすぎる場合に返されます

Example

```
curve25519_key key;  
wc_curve25519_init(&key);  
...  
int derSz = 128; // 出力 DER 用の適切なサイズ  
byte der[derSz];  
wc_Curve25519PrivateKeyToDer(&key, der, derSz);
```

C.4.2.41 function wc_Curve25519PublicKeyToDer

```
int wc_Curve25519PublicKeyToDer(  
    curve25519_key * key,  
    byte * output,  
    word32 inLen,  
    int withAlg  
)
```

この関数は、Curve25519 公開鍵を DER 形式にエンコードします。入力キー構造体に秘密鍵が含まれている場合、それは無視されます。

Parameters:

- **key** エンコードする公開鍵を含む curve25519_key 構造体へのポインタ
- **output** DER エンコーディングを保持するバッファ
- **inLen** 出力バッファのサイズ
- **withAlg** DER エンコーディングにアルゴリズム識別子を含めるかどうか

See:

- [wc_Curve25519KeyToDer](#)

- [wc_Curve25519PrivateKeyToDer](#)

Return:

- 0 成功、DER エンコーディングの長さ
- BAD_FUNC_ARG key または output が null の場合に返されます
- MEMORY_E 割り当て失敗がある場合に返されます
- BUFFER_E 出力バッファが小さすぎる場合に返されます

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // 出力 DER 用の適切なサイズ
byte der[derSz];
wc_Curve25519PublicKeyToDer(&key, der, derSz, 1);
```

C.4.2.42 function wc_Curve25519KeyToDer

```
int wc_Curve25519KeyToDer(
    curve25519_key * key,
    byte * output,
    word32 inLen,
    int withAlg
)
```

この関数は、Curve25519 キーを DER 形式にエンコードします。秘密鍵、公開鍵、または両方をエンコードできます。

Parameters:

- **key** エンコードするキーを含む curve25519_key 構造体へのポインタ
- **output** DER エンコーディングを保持するバッファ
- **inLen** 出力バッファのサイズ
- **withAlg** DER エンコーディングにアルゴリズム識別子を含めるかどうか

See:

- [wc_Curve25519PrivateKeyToDer](#)
- [wc_Curve25519PublicKeyToDer](#)

Return:

- 0 成功、DER エンコーディングの長さ
- BAD_FUNC_ARG key または output が null の場合に返されます
- MEMORY_E 割り当て失敗がある場合に返されます
- BUFFER_E 出力バッファが小さすぎる場合に返されます

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // 出力 DER 用の適切なサイズ
byte der[derSz];
wc_Curve25519KeyToDer(&key, der, derSz, 1);
```

C.4.2.43 function wc_EncodeSignature

```
word32 wc_EncodeSignature(
    byte * out,
    const byte * digest,
    word32 digSz,
    int hashOID
)
```

この関数は、デジタル署名を出力バッファにエンコードし、作成されたエンコードされた署名のサイズを返します。

Parameters:

- **out** エンコードされた署名が書き込まれるバッファへのポインタ
- **digest** 署名のエンコードに使用するダイジェストへのポインタ
- **digSz** ダイジェストを含むバッファの長さ
- **hashOID** 署名の生成に使用されるハッシュタイプを識別する OID。ビルド構成に応じた有効なオプションは：SHAh、SHA256h、SHA384h、SHA512h、MD2h、MD5h、DESh、DES3b、CTC_MD5wRSA、CTC_SHAwRSA、CTC_SHA256wRSA、CTC_SHA384wRSA、CTC_SHA512wRSA、CTC_SHAwECDSA、CTC_SHA256wECDSA、CTC_SHA384wECDSA、および CTC_SHA512wECDSA。

See: なし

Return: Success エンコードされた署名を output に正常に書き込むと、バッファに書き込まれた長さを返します

```
int signSz;
byte encodedSig[MAX_ENCODED_SIG_SZ];
Sha256 sha256;
// ハッシュ化のために sha256 を初期化

byte* dig = (byte*)malloc(WC_SHA256_DIGEST_SIZE);
// ハッシュ化とハッシュ更新を実行して dig に SHA-256 ハッシュを格納
// (wc_InitSha256、wc_Sha256Update、wc_Sha256Final を参照)
signSz = wc_EncodeSignature(encodedSig, dig, WC_SHA256_DIGEST_SIZE, SHA256h);
```

C.4.2.44 function wc_GetCTC_HashOID

```
int wc_GetCTC_HashOID(
    int type
)
```

この関数は、ハッシュタイプに対応するハッシュ OID を返します。例えば、WC_SHA512 タイプが与えられた場合、この関数は SHA512 ハッシュに対応する識別子 SHA512h を返します。

Parameters:

- **type** OID を見つけるハッシュタイプ。ビルド構成に応じた有効なオプションには：WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512

See: なし

Return:

- Success 成功時に、その暗号化タイプで使用する適切なハッシュに対応する OID を返します。
- 0 認識されないハッシュタイプが引数として渡された場合に返されます。

Example

```
int hashOID;

hashOID = wc_GetCTC_HashOID(WC_SHA512);
if (hashOID == 0) {
    // WOLFSSL_SHA512 が定義されていません
}
```

C.4.2.45 function wc_SetCert_Free

```
void wc_SetCert_Free(
    Cert * cert
)
```

この関数は、証明書構造体のデコードされた証明書キャッシュによって使用されるメモリとリソースをクリーンアップします。WOLFSSL_CERT_GEN_CACHE が定義されている場合、デコードされた証明書構造体は証明書構造体にキャッシュされます。これにより、証明書設定関数への後続の呼び出しで、各呼び出しでデコードされた証明書を解析することを回避できます。

Parameters:

- **cert** 初期化されていない証明書情報構造体へのポインタ。

See:

- [wc_SetAuthKeyIdFromCert](#)
- [wc_SetIssuerBuffer](#)
- [wc_SetSubjectBuffer](#)
- [wc_SetSubjectRaw](#)
- [wc_SetIssuerRaw](#)
- [wc_SetAltNamesBuffer](#)
- [wc_SetDatesBuffer](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG 無効なポインタが引数として渡された場合に返されます。

Example

```
Cert cert; // 初期化された証明書構造体

wc_SetCert_Free(&cert);
```

C.4.2.46 function wc_GetPkcs8TraditionalOffset

```
int wc_GetPkcs8TraditionalOffset(
    byte * input,
    word32 * inOutIdx,
    word32 sz
)
```

この関数は、PKCS#8 暗号化されていないバッファ内の従来の秘密鍵の先頭を見つけます。

Parameters:

- **input** 暗号化されていない PKCS#8 秘密鍵を含むバッファ。
- **inOutIdx** 入力バッファへのインデックス。入力時、PKCS#8 バッファの先頭へのバイトオフセットである必要があります。出力時、入力バッファ内の従来の秘密鍵へのバイトオフセットになります。
- **sz** 入力バッファのバイト数。

See:

- `wc_CreatePKCS8Key`
- `wc_EncryptPKCS8Key`
- `wc_DecryptPKCS8Key`
- `wc_CreateEncryptedPKCS8Key`

Return:

- Length 成功時に従来の秘密鍵の長さ。
- Negative 失敗時に負の値。

Example

```
byte* pkcs8Buf; // PKCS#8 キーを含むバッファ。
word32 idx = 0;
word32 sz; // pkcs8Buf のサイズ。
...
ret = wc_GetPkcs8TraditionalOffset(pkcs8Buf, &idx, sz);
// pkcs8Buf + idx は従来の秘密鍵バイトの先頭になります。
```

C.4.2.47 function wc_CreatePKCS8Key

```
int wc_CreatePKCS8Key(
    byte * out,
    word32 * outSz,
    byte * key,
    word32 keySz,
    int algoID,
    const byte * curveOID,
    word32 oidSz
)
```

この関数は、DER 秘密鍵を受け取り、PKCS#8 形式に変換します。PKCS#12 縮小キーバッグの作成にも使用されます。RFC 5208 を参照してください。

Parameters:

- **out** 結果を配置するバッファ。NULL の場合、outSz に必要な出力バッファサイズが返されます。
- **outSz** out バッファのサイズ。
- **key** 従来の DER キーを持つバッファ。
- **keySz** key バッファのサイズ。
- **algoID** アルゴリズム ID (例: RSAK)。
- **curveOID** 使用される場合の ECC 曲線 OID。RSA キーの場合は NULL である必要があります。
- **oidSz** 曲線 OID のサイズ。curveOID が NULL の場合は 0 に設定されます。

See:

- `wc_GetPkcs8TraditionalOffset`
- `wc_EncryptPKCS8Key`
- `wc_DecryptPKCS8Key`
- `wc_CreateEncryptedPKCS8Key`

Return:

- The 成功時に out に配置された PKCS#8 キーのサイズ。
- LENGTH_ONLY_E out が NULL の場合、outSz に必要な出力バッファサイズが返されます。
- Other 失敗時に負の値。

Example

```
ecc_key eccKey; // wolfSSL ECC キーオブジェクト。
byte* der; // DER エンコードされた ECC キー。
```

```

word32 derSize;           // der のサイズ。
const byte* curveOid = NULL; // eccKey で使用される曲線の OID。
word32 curveOidSz = 0;    // 曲線 OID のサイズ。
byte* pkcs8;             // PKCS#8 キー用の出力バッファ。
word32 pkcs8Sz;          // 出力バッファのサイズ。

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); // pkcs8Sz に必要なサイズを取得。
...
ret = wc_CreatePKCS8Key(pkcs8, &pkcs8Sz, der,
    derSize, ECDSAk, curveOid, curveOidSz);

```

C.4.2.48 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
)

```

この関数は、暗号化されていない PKCS#8 DER キー（例：wc_CreatePKCS8Key によって作成されたもの）を受け取り、PKCS#8 暗号化形式に変換します。結果の暗号化されたキーは、wc_DecryptPKCS8Key を使用して復号できます。RFC 5208 を参照してください。

Parameters:

- **key** 従来の DER キーを持つバッファ。
- **keySz** key バッファのサイズ。
- **out** 結果を配置するバッファ。NULL の場合、outSz に必要な出力バッファサイズが返されます。
- **outSz** out バッファのサイズ。
- **password** パスワードベースの暗号化アルゴリズムに使用するパスワード。
- **passwordSz** パスワードの長さ（NULL ターミネータを含まない）。
- **vPKCS** 使用する PKCS バージョン。PKCS12 または PKCS5 の場合は 1。
- **pbeOid** 使用する PBE スキームの OID（例：PBES2 または RFC 2898 A.3 の PBES1 の OID の 1 つ）。
- **encAlgId** 使用する暗号化アルゴリズム ID（例：AES256CBCb）。
- **salt** 使用するソルトバッファ。NULL の場合、ランダムなソルトが使用されます。
- **saltSz** ソルトバッファの長さ。salt に NULL を渡す場合は 0。
- **itt** KDF に使用する反復回数。
- **rng** 初期化された WC_RNG オブジェクトへのポインタ。

- **heap** 動的割り当てに使用されるヒープへのポインタ。NULL でも可。

See:

- [wc_GetPkcs8TraditionalOffset](#)
- [wc_CreatePKCS8Key](#)
- [wc_DecryptPKCS8Key](#)
- [wc_CreateEncryptedPKCS8Key](#)

Return:

- The 成功時に out に配置された暗号化されたキーのサイズ。
- LENGTH_ONLY_E out が NULL の場合、outSz に必要な出力バッファサイズが返されます。
- Other 失敗時に負の値。

Example

```
byte* pkcs8;           // 暗号化されていない PKCS#8 キー。
word32 pkcs8Sz;        // pkcs8 のサイズ。
byte* pkcs8Enc;        // 暗号化された PKCS#8 キー。
word32 pkcs8EncSz;      // pkcs8Enc のサイズ。
const char* password;  // 暗号化に使用するパスワード。
int passwordSz;         // パスワードの長さ (NULL ターミネータを含まない)。
WC_RNG rng;

// 以下は、pkcs8Enc に pkcs8 の暗号化されたバージョンを生成します。暗号化は
// PKCS#5 のパスワードベース暗号化スキーム 2 (PBE2) と CBC モードの 256 ビット
// キーを持つ AES 暗号を使用します。PKCS#5 の詳細については RFC 8018 を参照してください。
ret = wc_EncryptPKCS8Key(pkcs8, pkcs8Sz, pkcs8Enc, &pkcs8EncSz, password,
    passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
    WC_PKCS12_ITT_DEFAULT, &rng, NULL);
```

C.4.2.49 function wc_DecryptPKCS8Key

```
int wc_DecryptPKCS8Key(
    byte * input,
    word32 sz,
    const char * password,
    int passwordSz
)
```

この関数は、暗号化された PKCS#8 DER キーを受け取り、PKCS#8 暗号化されていない DER に復号します。wc_EncryptPKCS8Key によって行われた暗号化を元に戻します。RFC5208 を参照してください。入力バッファは復号されたデータで上書きされます。

Parameters:

- **input** 入力時、暗号化された PKCS#8 キーを含むバッファ。出力が成功すると、復号されたキーを含みます。
- **sz** 入力バッファのサイズ。
- **password** キーの暗号化に使用されたパスワード。
- **passwordSz** パスワードの長さ (NULL ターミネータを含まない)。

See:

- [wc_GetPkcs8TraditionalOffset](#)
- [wc_CreatePKCS8Key](#)
- [wc_EncryptPKCS8Key](#)
- [wc_CreateEncryptedPKCS8Key](#)

Return:

- The 成功時に復号されたバッファの長さ。
- Negative 失敗時に負の値。

Example

```
byte* pkcs8Enc;           // wc_EncryptPKCS8Key で作成された暗号化された PKCS#8 キー。
word32 pkcs8EncSz;        // pkcs8Enc のサイズ。
const char* password;     // 復号に使用するパスワード。
int passwordSz;           // パスワードの長さ (NULL ターミネータを含まない)。

ret = wc_DecryptPKCS8Key(pkcs8Enc, pkcs8EncSz, password, passwordSz);
```

C.4.2.50 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
)
```

この関数は、従来の DER キーを受け取り、PKCS#8 形式に変換し、暗号化します。これを行うために wc_CreatePKCS8Key と wc_EncryptPKCS8Key を使用します。

Parameters:

- **key** 従来の DER キーを持つバッファ。
- **keySz** key バッファのサイズ。
- **out** 結果を配置するバッファ。NULL の場合、outSz に必要な出力バッファサイズが返されます。
- **outSz** out バッファのサイズ。
- **password** パスワードベースの暗号化アルゴリズムに使用するパスワード。
- **passwordSz** パスワードの長さ (NULL ターミネータを含まない)。
- **vPKCS** 使用する PKCS バージョン。PKCS12 または PKCS5 の場合は 1。
- **pbeOid** 使用する PBE スキームの OID (例: PBES2 または RFC 2898 A.3 の PBES1 の OID の 1 つ)。
- **encAlgId** 使用する暗号化アルゴリズム ID (例: AES256CBCb)。
- **salt** 使用するソルトバッファ。NULL の場合、ランダムなソルトが使用されます。
- **saltSz** ソルトバッファの長さ。salt に NULL を渡す場合は 0。
- **itt** KDF に使用する反復回数。
- **rng** 初期化された WC_RNG オブジェクトへのポインタ。
- **heap** 動的割り当てに使用されるヒープへのポインタ。NULL でも可。

See:

- [wc_GetPkcs8TraditionalOffset](#)
- [wc_CreatePKCS8Key](#)
- [wc_EncryptPKCS8Key](#)

- `wc_DecryptPKCS8Key`

Return:

- The 成功時に out に配置された暗号化されたキーのサイズ。
- LENGTH_ONLY_E out が NULL の場合、outSz に必要な出力バッファサイズが返されます。
- Other 失敗時に負の値。

Example

```
byte* key;           // 従来の秘密鍵 (DER 形式)。
word32 keySz;        // key のサイズ。
byte* pkcs8Enc;      // 暗号化された PKCS#8 キー。
word32 pkcs8EncSz;   // pkcs8Enc のサイズ。
const char* password; // 暗号化に使用するパスワード。
int passwordSz;      // パスワードの長さ (NULL ターミネータを含まない)。
WC_RNG rng;

// 以下は、pkcs8Enc に key の暗号化された PKCS#8 バージョンを生成します。
// 暗号化は PKCS#5 のパスワードベース暗号化スキーム 2 (PBE2) と CBC モードの
// 256 ビットキーを持つ AES 暗号を使用します。PKCS#5 の詳細については
// RFC 8018 を参照してください。
ret = wc_CreateEncryptedPKCS8Key(key, keySz, pkcs8Enc, &pkcs8EncSz,
    password, passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
    WC_PKCS12_ITT_DEFAULT, &rng, NULL);
```

C.4.2.51 function wc_InitDecodedCert

```
void wc_InitDecodedCert(
    struct DecodedCert * cert,
    const byte * source,
    word32 inSz,
    void * heap
)
```

この関数は、“cert” パラメータが指す DecodedCert を初期化します。長さ “inSz” の DER エンコードされた証明書への “source” ポインタを保存します。この証明書は、wc_ParseCert への後続の呼び出しによって解析できます。

Parameters:

- **cert** 割り当てられた DecodedCert オブジェクトへのポインタ。
- **source** DER エンコードされた証明書へのポインタ。
- **inSz** DER エンコードされた証明書の長さ (バイト単位)。
- **heap** 動的割り当てに使用されるヒープへのポインタ。NULL でも可。

See:

- `wc_ParseCert`
- `wc_FreeDecodedCert`

Example

```
DecodedCert decodedCert; // デコードされた証明書オブジェクト。
byte* certBuf;           // DER エンコードされた証明書バッファ。
word32 certBufSz;        // certBuf のサイズ (バイト単位)。

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
```

C.4.2.52 function wc_ParseCert

```
int wc_ParseCert(
    DecodedCert * cert,
    int type,
    int verify,
    void * cm
)
```

この関数は、DecodedCert オブジェクトに保存された DER エンコードされた証明書を解析し、そのオブジェクトのフィールドを設定します。DecodedCert は、wc_InitDecodedCert への事前の呼び出しで初期化されている必要があります。この関数は、CertificateManager オブジェクトへのオプションのポインタを取り、CA が CertificateManager で見つかった場合、DecodedCert の証明機関情報を設定するために使用されます。

Parameters:

- **cert** 初期化された DecodedCert オブジェクトへのポインタ。
- **type** 証明書のタイプ。asn_public.h の CertType 列挙型を参照してください。
- **verify** 設定されている場合、ユーザーが証明書の有効性を検証したいことを示すフラグ。
- **cm** CertificateManager へのオプションのポインタ。NULL でも可。

See:

- [wc_InitDecodedCert](#)
- [wc_FreeDecodedCert](#)

Return:

- 0 成功時。
- Other 失敗時に負の値。

Example

```
int ret;
DecodedCert decodedCert; // デコードされた証明書オブジェクト。
byte* certBuf;           // DER エンコードされた証明書バッファ。
word32 certBufSz;        // certBuf のサイズ (バイト単位)。

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.4.2.53 function wc_FreeDecodedCert

```
void wc_FreeDecodedCert(
    struct DecodedCert * cert
)
```

この関数は、wc_InitDecodedCert で以前に初期化された DecodedCert を解放します。

Parameters:

- **cert** 初期化された DecodedCert オブジェクトへのポインタ。

See:

- [wc_InitDecodedCert](#)
- [wc_ParseCert](#)

Example

```

int ret;
DecodedCert decodedCert; // デコードされた証明書オブジェクト。
byte* certBuf;           // DER エンコードされた証明書バッファ。
word32 certBufSz;        // certBuf のサイズ (バイト単位)。

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.4.2.54 function wc_SetTimeCb

```

int wc_SetTimeCb(
    wc_time_cb f
)

```

この関数は、wolfSSL が現在時刻を取得する必要があるときに使用される時刻コールバックを登録します。コールバックのプロトタイプは、C 標準ライブラリの “time” 関数と同じである必要があります。

Parameters:

- **f** 時刻コールバックとして登録する関数。

See: `wc_Time`

Return: 0 成功時に返されます。

Example

```

int ret = 0;
// 時刻コールバックのプロトタイプ
time_t my_time_cb(time_t* t);
// 登録する
ret = wc_SetTimeCb(my_time_cb);
if (ret != 0) {
    // 時刻コールバックの設定失敗
}
time_t my_time_cb(time_t* t)
{
    // カスタム時刻関数
}

```

C.4.2.55 function wc_Time

```

time_t wc_Time(
    time_t * t
)

```

この関数は、現在時刻を取得します。デフォルトでは、プラットフォーム間で異なる XTIME マクロを使用します。ユーザーは、wc_SetTimeCb 関数を介して任意の関数を使用できます。

Parameters:

- **t** 現在時刻を設定するオプションの time_t ポインタ。

See: [wc_SetTimeCb](#)

Return: Time 成功時に返される現在時刻。

Example

```
time_t currentTime = 0;
currentTime = wc_Time(NULL);
wc_Time(&currentTime);
```

C.4.2.56 function wc_SetCustomExtension

```
int wc_SetCustomExtension(
    Cert * cert,
    int critical,
    const char * oid,
    const byte * der,
    word32 derSz
)
```

この関数は、X.509 証明書にカスタム拡張を挿入します。注：ポインタであるパラメータのいずれかが指すアドレスのコンテンツは、証明書が生成されて der 出力が得られるまで変更してはいけません。この関数はコンテンツを別のバッファにコピーしません。

Parameters:

- **cert** 初期化された DecodedCert オブジェクトへのポインタ。
- **critical** 0 の場合、拡張は重要としてマークされません。それ以外の場合は重要としてマークされます。
- **oid** ドット区切りの oid を文字列として。例：“1.2.840.10045.3.1.7”
- **der** 拡張のコンテンツの der エンコーディング。
- **derSz** der エンコーディングのサイズ（バイト単位）。

See:

- [wc_InitCert](#)
- [wc_SetUnknownExtCallback](#)

Return:

- 0 成功時に返されます。
- Other 失敗時に負の値。

Example

```
int ret = 0;
Cert newCert;
wc_InitCert(&newCert);

// subject、公開鍵、発行者、その他のものを設定するコードがここに入ります。

ret = wc_SetCustomExtension(&newCert, 1, "1.2.3.4.5",
    (const byte *)"This is a critical extension", 28);
if (ret < 0) {
    // 拡張の設定失敗。
}

ret = wc_SetCustomExtension(&newCert, 0, "1.2.3.4.6",
    (const byte *)"This is NOT a critical extension", 32)
if (ret < 0) {
```

```

    // 拡張の設定失敗。
}

// 証明書に署名してから書き出すコードがここに入ります。

```

C.4.2.57 function wc_SetUnknownExtCallback

```

int wc_SetUnknownExtCallback(
    DecodedCert * cert,
    wc_UnknownExtCallback cb
)

```

この関数は、wolfSSL が証明書の解析中に証明書内の不明な X.509 拡張に遭遇したときに使用されるコールバックを登録します。コールバックのプロトタイプは次のようにする必要があります：

Parameters:

- **cert** このコールバックに関連付けられる DecodedCert 構造体。
- **cb** 時刻コールバックとして登録する関数。

See:

- ParseCert
- wc_SetCustomExtension

Return:

- 0 成功時に返されます。
- Other 失敗時に負の値。

Example

```

int ret = 0;
// 不明な拡張コールバックのプロトタイプ
int myUnknownExtCallback(const word16* oid, word32 oidSz, int crit,
                        const unsigned char* der, word32 derSz);

// 登録する
ret = wc_SetUnknownExtCallback(cert, myUnknownExtCallback);
if (ret != 0) {
    // コールバックの設定失敗
}

// oid: oid のドット区切り値である整数の配列。
// oidSz: oid 内の値の数。
// crit: 拡張が重要としてマークされたかどうか。
// der: 拡張のコンテンツの der エンコーディング。
// derSz: der エンコーディングのサイズ (バイト単位)。
int myCustomExtCallback(const word16* oid, word32 oidSz, int crit,
                        const unsigned char* der, word32 derSz) {

    // 拡張を解析するロジックがここに入ります。

    // 注：0 を返すことで、この拡張を受け入れ、wolfSSL に
    // それが許容可能であることを通知しています。許容できない拡張が
    // 見つかった場合は、エラーを返す必要があります。重要フラグが
    // 設定された不明な拡張に遭遇した場合の標準的な動作は、
    // ASN_CRIT_EXT_E を返すことです。簡潔にするため、この例では
    // 常にすべての拡張を受け入れています。異なるロジックを

```

```
    // 使用する必要があります。  
    return 0;  
}
```

C.4.2.58 function wc_CheckCertSigPubKey

```
int wc_CheckCertSigPubKey(  
    const byte * cert,  
    word32 certSz,  
    void * heap,  
    const byte * pubKey,  
    word32 pubKeySz,  
    int pubKeyOID  
)
```

この関数は、X.509 証明書の der 形式の署名を公開鍵に対して検証します。公開鍵は、der 形式の完全なサブジェクト公開鍵情報であることが期待されます。

Parameters:

- **cert** X.509 証明書の der エンコーディング。
- **certSz** cert のサイズ (バイト単位)。
- **heap** 動的割り当てに使用されるヒープへのポインタ。NULL でも可。
- **pubKey** 公開鍵の der エンコーディング。
- **pubKeySz** pubKey のサイズ (バイト単位)。
- **pubKeyOID** 公開鍵のアルゴリズムを識別する OID。(例: ECDSAk、DSAk または RSAk)

Return:

- 0 成功時に返されます。
- Other 失敗時に負の値。

C.4.2.59 function wc_Asn1PrintOptions_Init

```
int wc_Asn1PrintOptions_Init(  
    Asn1PrintOptions * opts  
)
```

この関数は、ASN.1 プリントオプションを初期化します。

Parameters:

- **opts** プリント用の ASN.1 オプション。

See:

- [wc_Asn1PrintOptions_Set](#)
- [wc_Asn1_PrintAll](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG asn1 が NULL の場合。

Example

```
Asn1PrintOptions opt;
```

```
// 使用前に ASN.1 プリントオプションを初期化します。  
wc_Asn1PrintOptions_Init(&opt);
```


C.4.2.60 function wc_Asn1PrintOptions_Set

```
int wc_Asn1PrintOptions_Set(  
    Asn1PrintOptions * opts,  
    enum Asn1PrintOpt opt,  
    word32 val  
)
```

この関数は、ASN.1 プリントオプションオブジェクトにプリントオプションを設定します。

Parameters:

- **opts** プリント用の ASN.1 オプション。
- **opt** 値を設定するオプション。
- **val** 設定する値。

See:

- [wc_Asn1PrintOptions_Init](#)
- [wc_Asn1_PrintAll](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG asn1 が NULL の場合。
- BAD_FUNC_ARG val が option の範囲外の場合。

Example

```
Asn1PrintOptions opt;
```

```
// 使用前に ASN.1 プリントオプションを初期化します。  
wc_Asn1PrintOptions_Init(&opt);  
// タグ名をプリントする際のインデント数を 1 に設定します。  
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);
```

C.4.2.61 function wc_Asn1_Init

```
int wc_Asn1_Init(  
    Asn1 * asn1  
)
```

この関数は、ASN.1 解析オブジェクトを初期化します。

Parameters:

- **asn1** ASN.1 解析オブジェクト。

See:

- [wc_Asn1_SetFile](#)
- [wc_Asn1_PrintAll](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG asn1 が NULL の場合。

Example

```
Asn1 asn1;
```

```
// 使用前に ASN.1 解析オブジェクトを初期化します。  
wc_Asn1_Init(&asn1);
```

C.4.2.62 function wc_Asn1_SetFile

```
int wc_Asn1_SetFile(  
    Asn1 * asn1,  
    XFILE file  
)
```

この関数は、ASN.1 解析オブジェクトへのプリント時に使用するファイルを設定します。

Parameters:

- **asn1** ASN.1 解析オブジェクト。
- **file** プリント先のファイル。

See:

- [wc_Asn1_Init](#)
- [wc_Asn1_PrintAll](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG asn1 が NULL の場合。
- BAD_FUNC_ARG file が XBADFILE の場合。

Example

```
Asn1 asn1;  
  
// 使用前に ASN.1 解析オブジェクトを初期化します。  
wc_Asn1_Init(&asn1);  
// 標準出力を書き込み先のファイル記述子として設定します。  
wc_Asn1_SetFile(&asn1, stdout);
```

C.4.2.63 function wc_Asn1_PrintAll

```
int wc_Asn1_PrintAll(  
    Asn1 * asn1,  
    Asn1PrintOptions * opts,  
    unsigned char * data,  
    word32 len  
)
```

すべての ASN.1 項目をプリントします。

Parameters:

- **asn1** ASN.1 解析オブジェクト。
- **opts** ASN.1 プリントオプション。
- **data** プリントする BER/DER データを含むバッファ。
- **len** プリントするデータの長さ (バイト単位)。

See:

- [wc_Asn1_Init](#)
- [wc_Asn1_SetFile](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG asn1 または opts が NULL の場合。
- ASN_LEN_E ASN.1 項目の長さが長すぎる場合。
- ASN_DEPTH_E 終了オフセットが無効な場合。

- ASN_PARSE_E ASN.1 項目のすべてが解析されなかった場合。

```

Asn1PrintOptions opts;
Asn1 asn1;
unsigned char data[] = { DER/BER データで初期化 };
word32 len = sizeof(data);

// 使用前に ASN.1 プリントオプションを初期化します。
wc_Asn1PrintOptions_Init(&opt);
// タグ名をプリントする際のインデント数を 1 に設定します。
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);

// 使用前に ASN.1 解析オブジェクトを初期化します。
wc_Asn1_Init(&asn1);
// 標準出力を書き込み先のファイル記述子として設定します。
wc_Asn1_SetFile(&asn1, stdout);
// 指定されたプリントオプションでバッファ内のすべての ASN.1 項目をプリントします。
wc_Asn1_PrintAll(&asn1, &opts, data, len);

```

C.4.3 Source code

```

int wc_InitCert(Cert* cert);

Cert* wc_CertNew(void* heap);

void wc_CertFree(Cert* cert);

int wc_MakeCert(Cert* cert, byte* derBuffer, word32 derSz, RsaKey* rsaKey,
                ecc_key* eccKey, WC_RNG* rng);

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_SetDatesBuffer(Cert* cert, const byte* der, int derSz);

int wc_SetAuthKeyIdFromPublicKey(Cert *cert, RsaKey *rsaKey,
                                  ecc_key *eckey);

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_SetSubjectKeyId(Cert *cert, const char* file);

int wc_SetKeyUsage(Cert *cert, const char *value);

int wc_PemPubKeyToDer(const char* fileName,
                     unsigned char* derBuf, int derSz);

int wc_PubKeyPemToDer(const unsigned char* pem, int pemSz,
                     unsigned char* buff, int buffSz);

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_GetPubKeyDerFromCert(struct DecodedCert* cert,
                          byte* derKey, word32* derKeySz);

int wc_EccPrivateKeyDecode(const byte* input, word32* inOutIdx,
                          ecc_key* key, word32 inSz);

int wc_EccKeyToDer(ecc_key* key, byte* output, word32 inLen);

int wc_EccPublicKeyDecode(const byte* input, word32* inOutIdx,
                          ecc_key* key, word32 inSz);

int wc_EccPublicKeyToDer(ecc_key* key, byte* output,
                        word32 inLen, int with_AlgCurve);
```

```
int wc_EccPublicKeyToDer_ex(ecc_key* key, byte* output,
                           word32 inLen, 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 inLen);

int wc_Curve25519PublicKeyToDer(curve25519_key* key, byte* output, word32
    ↪ inLen,
                                int withAlg);

int wc_Curve25519KeyToDer(curve25519_key* key, byte* output, word32 inLen,
                           int withAlg);

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_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, void* 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);

```

C.5 dox_comments/header_files-ja/blake2.h

C.5.1 Functions

	Name
int	wc_InitBlake2b (Blake2b * b2b, word32 digestSz) この関数は、Blake2 ハッシュ関数で使用するための Blake2b 構造体を初期化します。
int	wc_Blake2bUpdate (Blake2b * b2b, const byte * data, word32 sz) この関数は、指定された入力データで Blake2b ハッシュを更新します。この関数は wc_InitBlake2b の後に呼び出され、最終ハッシュ (wc_Blake2bFinal) の準備ができるまで繰り返されます。
int	wc_Blake2bFinal (Blake2b * b2b, byte * final, word32 requestSz) この関数は、以前に提供された入力データの Blake2b ハッシュを計算します。出力ハッシュの長さは requestSz になります。requestSz==0 の場合は、b2b 構造体の digestSz が使用されます。この関数は、wc_InitBlake2b の後、および必要な各入力データに対して wc_Blake2bUpdate が処理された後に呼び出す必要があります。

C.5.2 Functions Documentation

C.5.2.1 function wc_InitBlake2b

```
int wc_InitBlake2b(
    Blake2b * b2b,
    word32 digestSz
)
```

この関数は、Blake2 ハッシュ関数で使用するための Blake2b 構造体を初期化します。

Parameters:

- **b2b** 初期化する Blake2b 構造体へのポインタ
- **digestSz** 実装する blake 2 ダイジェストの長さ

See: [wc_Blake2bUpdate](#)

Return: 0 Blake2b 構造体の初期化とダイジェストサイズの設定に成功した場合に返されます。

Example

```
Blake2b b2b;
// 64 バイトダイジェストで Blake2b 構造体を初期化
wc_InitBlake2b(&b2b, 64);
```

C.5.2.2 function wc_Blake2bUpdate

```
int wc_Blake2bUpdate(
    Blake2b * b2b,
    const byte * data,
    word32 sz
)
```

この関数は、指定された入力データで Blake2b ハッシュを更新します。この関数は wc_InitBlake2b の後に呼び出され、最終ハッシュ (wc_Blake2bFinal) の準備ができるまで繰り返されます。

Parameters:

- **b2b** 更新する Blake2b 構造体へのポインタ
- **data** 追加するデータを含むバッファへのポインタ
- **sz** 追加する入力データの長さ

See:

- [wc_InitBlake2b](#)
- [wc_Blake2bFinal](#)

Return:

- 0 指定されたデータで Blake2b 構造体の更新に成功した場合に返されます
- -1 入力データの圧縮中に失敗した場合に返されます

Example

```
int ret;
Blake2b b2b;
// 64 バイトダイジェストで Blake2b 構造体を初期化
wc_InitBlake2b(&b2b, 64);

byte plain[] = { // 入力を初期化 };

ret = wc_Blake2bUpdate(&b2b, plain, sizeof(plain));
```

```

if( ret != 0) {
    // blake2b の更新エラー
}

```

C.5.2.3 function wc_Blake2bFinal

```

int wc_Blake2bFinal(
    Blake2b * b2b,
    byte * final,
    word32 requestSz
)

```

この関数は、以前に提供された入力データの Blake2b ハッシュを計算します。出力ハッシュの長さは requestSz になります。requestSz==0 の場合は、b2b 構造体の digestSz が使用されます。この関数は、wc_InitBlake2b の後、および必要な各入力データに対して wc_Blake2bUpdate が処理された後に呼び出す必要があります。

Parameters:

- **b2b** 更新する Blake2b 構造体へのポインタ
- **final** blake2b ハッシュを格納するバッファへのポインタ。requestSz の長さである必要があります
- **requestSz** 計算するダイジェストの長さ。これがゼロの場合、代わりに b2b->digestSz が使用されます

See:

- [wc_InitBlake2b](#)
- [wc_Blake2bUpdate](#)

Return:

- 0 Blake2b ハッシュの計算に成功した場合に返されます
- -1 Blake2b ハッシュの解析中に失敗した場合に返されます

Example

```

int ret;
Blake2b b2b;
byte hash[64];
// 64 バイトダイジェストで Blake2b 構造体を初期化
wc_InitBlake2b(&b2b, 64);
... // wc_Blake2bUpdate を呼び出してハッシュにデータを追加

ret = wc_Blake2bFinal(&b2b, hash, 64);
if( ret != 0) {
    // blake2b ハッシュの生成エラー
}

```

C.5.3 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);

```


C.6 dox_comments/header_files-ja/bn.h

C.6.1 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) この関数は次の数学演算を実行します “ $r = (a^p) \% m$ ”。

C.6.2 Functions Documentation

C.6.2.1 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
)
```

この関数は次の数学演算を実行します “ $r = (a^p) \% m$ ”。

Parameters:

- **r** 結果を保持する構造体。
- **a** 累乗される値。
- **p** a を累乗する指数。
- **m** 使用する剰余。
- **ctx** 現在 wolfSSL では使用されていないため、NULL にできます。

See:

- wolfSSL_BN_new
- wolfSSL_BN_free

Return:

- SSL_SUCCESS 数学演算が正常に実行された場合。
- SSL_FAILURE エラーケースが発生した場合。

Example

```
WOLFSSL_BIGNUM r,a,p,m;
int ret;
// big number 値を設定
ret = wolfSSL_BN_mod_exp(r, a, p, m, NULL);
// ret 値を確認
```

C.6.3 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.7 dox_comments/header_files-ja/camellia.h

C.7.1 Functions

	Name
int	wc_CamelliaSetKey (wc_Camellia * cam, const byte * key, word32 len, const byte * iv) この関数は、camellia オブジェクトのキーと初期化ベクトルを設定し、暗号として使用するために初期化します。
int	wc_CamelliaSetIV (wc_Camellia * cam, const byte * iv) この関数は、camellia オブジェクトの初期化ベクトルを設定します。
int	wc_CamelliaEncryptDirect (wc_Camellia * cam, byte * out, const byte * in) この関数は、提供された camellia オブジェクトを使用して 1 ブロックの暗号化を行います。バッファ in から最初の 16 バイトブロックを解析し、暗号化された結果をバッファ out に格納します。この関数を使用する前に、wc_CamelliaSetKey を使用して camellia オブジェクトを初期化する必要があります。
int	wc_CamelliaDecryptDirect (wc_Camellia * cam, byte * out, const byte * in) この関数は、提供された camellia オブジェクトを使用して 1 ブロックの復号を行います。バッファ in から最初の 16 バイトブロックを解析し、復号して、結果をバッファ out に格納します。この関数を使用する前に、wc_CamelliaSetKey を使用して camellia オブジェクトを初期化する必要があります。
int	wc_CamelliaCbcEncrypt (wc_Camellia * cam, byte * out, const byte * in, word32 sz) この関数は、バッファ in から平文を暗号化し、出力をバッファ out に格納します。暗号ブロック連鎖 (CBC) モードの Camellia を使用してこの暗号化を実行します。
int	wc_CamelliaCbcDecrypt (wc_Camellia * cam, byte * out, const byte * in, word32 sz) この関数は、バッファ in から暗号文を復号し、出力をバッファ out に格納します。暗号ブロック連鎖 (CBC) モードの Camellia を使用してこの復号を実行します。

C.7.2 Functions Documentation

C.7.2.1 function wc_CamelliaSetKey

```
int wc_CamelliaSetKey(
    wc_Camellia * cam,
    const byte * key,
    word32 len,
    const byte * iv
)
```

この関数は、camellia オブジェクトのキーと初期化ベクトルを設定し、暗号として使用するために初期化します。

Parameters:

- **cam** キーと iv を設定する camellia 構造体へのポインタ
- **key** 暗号化と復号に使用する 16、24、または 32 バイトのキーを含むバッファへのポインタ
- **len** 渡されるキーの長さ
- **iv** この camellia 構造体で使用する 16 バイトの初期化ベクトルを含むバッファへのポインタ

See:

- [wc_CamelliaEncryptDirect](#)
- [wc_CamelliaDecryptDirect](#)
- [wc_CamelliaCbcEncrypt](#)
- [wc_CamelliaCbcDecrypt](#)

Return:

- 0 キーと初期化ベクトルの設定に成功した場合に返されます
- BAD_FUNC_ARG 入力引数の 1 つの処理中にエラーが発生した場合に返されます
- MEMORY_E_XMALLOC でメモリ割り当て中にエラーが発生した場合に返されます

Example

```
Camellia cam;
byte key[32];
// キーを初期化
byte iv[16];
// iv を初期化
if( wc_CamelliaSetKey(&cam, key, sizeof(key), iv) != 0) {
    // camellia 構造体の初期化エラー
}
```

C.7.2.2 function wc_CamelliaSetIV

```
int wc_CamelliaSetIV(
    wc_Camellia * cam,
    const byte * iv
)
```

この関数は、camellia オブジェクトの初期化ベクトルを設定します。

Parameters:

- **cam** iv を設定する camellia 構造体へのポインタ
- **iv** この camellia 構造体で使用する 16 バイトの初期化ベクトルを含むバッファへのポインタ

See: [wc_CamelliaSetKey](#)

Return:

- 0 キーと初期化ベクトルの設定に成功した場合に返されます
- BAD_FUNC_ARG 入力引数の 1 つの処理中にエラーが発生した場合に返されます

Example

```
Camellia cam;
byte iv[16];
// iv を初期化
if( wc_CamelliaSetIV(&cam, iv) != 0) {
```

```
// camellia 構造体の初期化エラー
}
```

C.7.2.3 function wc_CamelliaEncryptDirect

```
int wc_CamelliaEncryptDirect(
    wc_Camellia * cam,
    byte * out,
    const byte * in
)
```

この関数は、提供された camellia オブジェクトを使用して 1 ブロックの暗号化を行います。バッファ in から最初の 16 バイトブロックを解析し、暗号化された結果をバッファ out に格納します。この関数を使用する前に、wc_CamelliaSetKey を使用して camellia オブジェクトを初期化する必要があります。

Parameters:

- **cam** 暗号化に使用する camellia 構造体へのポインタ
- **out** 暗号化されたブロックを格納するバッファへのポインタ
- **in** 暗号化する平文ブロックを含むバッファへのポインタ

See: [wc_CamelliaDecryptDirect](#)

Return: none 戻り値なし。

Example

```
Camellia cam;
// キーと iv で cam 構造体を初期化
byte plain[] = { // 暗号化するメッセージで初期化 };
byte cipher[16];

wc_CamelliaEncryptDirect(&cam, cipher, plain);
```

C.7.2.4 function wc_CamelliaDecryptDirect

```
int wc_CamelliaDecryptDirect(
    wc_Camellia * cam,
    byte * out,
    const byte * in
)
```

この関数は、提供された camellia オブジェクトを使用して 1 ブロックの復号を行います。バッファ in から最初の 16 バイトブロックを解析し、復号して、結果をバッファ out に格納します。この関数を使用する前に、wc_CamelliaSetKey を使用して camellia オブジェクトを初期化する必要があります。

Parameters:

- **cam** 暗号化に使用する camellia 構造体へのポインタ
- **out** 復号された平文ブロックを格納するバッファへのポインタ
- **in** 復号する暗号文ブロックを含むバッファへのポインタ

See: [wc_CamelliaEncryptDirect](#)

Return: none 戻り値なし。

Example

```
Camellia cam;
// キーと iv で cam 構造体を初期化
byte cipher[] = { // 復号する暗号化されたメッセージで初期化 };
```

```
byte decrypted[16];

wc_CamelliaDecryptDirect(&cam, decrypted, cipher);
```

C.7.2.5 function wc_CamelliaCbcEncrypt

```
int wc_CamelliaCbcEncrypt(
    wc_Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、バッファ in から平文を暗号化し、出力をバッファ out に格納します。暗号ブロック連鎖 (CBC) モードの Camellia を使用してこの暗号化を実行します。

Parameters:

- **cam** 暗号化に使用する camellia 構造体へのポインタ
- **out** 暗号化された暗号文を格納するバッファへのポインタ
- **in** 暗号化する平文を含むバッファへのポインタ
- **sz** 暗号化するメッセージのサイズ

See: [wc_CamelliaCbcDecrypt](#)

Return: none 戻り値なし。

Example

```
Camellia cam;
// キーと iv で cam 構造体を初期化
byte plain[] = { // 復号する暗号化されたメッセージで初期化 };
byte cipher[sizeof(plain)];

wc_CamelliaCbcEncrypt(&cam, cipher, plain, sizeof(plain));
```

C.7.2.6 function wc_CamelliaCbcDecrypt

```
int wc_CamelliaCbcDecrypt(
    wc_Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、バッファ in から暗号文を復号し、出力をバッファ out に格納します。暗号ブロック連鎖 (CBC) モードの Camellia を使用してこの復号を実行します。

Parameters:

- **cam** 暗号化に使用する camellia 構造体へのポインタ
- **out** 復号されたメッセージを格納するバッファへのポインタ
- **in** 暗号化された暗号文を含むバッファへのポインタ
- **sz** 暗号化するメッセージのサイズ

See: [wc_CamelliaCbcEncrypt](#)

Return: none 戻り値なし。

Example

```
Camellia cam;
// キーと iv で cam 構造体を初期化
byte cipher[] = { // 復号する暗号化されたメッセージで初期化 };
byte decrypted[sizeof(cipher)];

wc_CamelliaCbcDecrypt(&cam, decrypted, cipher, sizeof(cipher));
```

C.7.3 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.8 dox_comments/header_files-ja/chacha20_poly1305.h

C.8.1 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 outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]) この関数は、入力メッセージ inPlaintext を ChaCha20 ストリーム暗号を使用して暗号化し、出力バッファ outCiphertext に格納します。また、Poly_1305 認証 (暗号文に対して) を実行し、生成された認証タグを出力バッファ outAuthTag に格納します。

	Name
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) この関数は、入力暗号文 inCiphertext を ChaCha20 ストリーム暗号を使用して復号し、出力バッファ outPlaintext に格納します。また、Poly_1305 認証を実行し、指定された inAuthTag と inAAD(任意の長さの追加認証データ) で生成された認証を比較します。ゼロ以外のエラーコードが返された場合、出力データ outPlaintext は未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

C.8.2 Functions Documentation

C.8.2.1 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]
)
```

この関数は、入力メッセージ inPlaintext を ChaCha20 ストリーム暗号を使用して暗号化し、出力バッファ outCiphertext に格納します。また、Poly-1305 認証 (暗号文に対して) を実行し、生成された認証タグを出力バッファ outAuthTag に格納します。

Parameters:

- **inKey** 暗号化に使用する 32 バイトのキーを含むバッファへのポインタ
- **inIV** 暗号化に使用する 12 バイトの iv を含むバッファへのポインタ
- **inAAD** 任意の長さの追加認証データ (AAD) を含むバッファへのポインタ
- **inAADLen** 入力 AAD の長さ
- **inPlaintext** 暗号化する平文を含むバッファへのポインタ
- **inPlaintextLen** 暗号化する平文の長さ
- **outCiphertext** 暗号文を格納するバッファへのポインタ
- **outAuthTag** 認証タグを格納する 16 バイト幅のバッファへのポインタ

See:

- **wc_ChaCha20Poly1305_Decrypt**
- **wc_ChaCha_***
- **wc_Poly1305***

Return:

- 0 メッセージの暗号化に成功した場合に返されます
- BAD_FUNC_ARG 暗号化プロセス中にエラーが発生した場合に返されます

Example

```
byte key[] = { // 32 バイトのキーを初期化 };
byte iv[] = { // 12 バイトのキーを初期化 };
byte inAAD[] = { // AAD を初期化 };

byte plain[] = { // 暗号化するメッセージを初期化 };
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) {
    // 暗号化実行エラー
}
```

C.8.2.2 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
)
```

この関数は、入力暗号文 inCiphertext を ChaCha20 ストリーム暗号を使用して復号し、出力バッファ outPlaintext に格納します。また、Poly-1305 認証を実行し、指定された inAuthTag と inAAD(任意の長さの追加認証データ) で生成された認証を比較します。ゼロ以外のエラーコードが返された場合、出力データ outPlaintext は未定義です。ただし、呼び出し元は平文データの漏洩を防ぐために、無条件に出力バッファをゼロ化する必要があります。

Parameters:

- **inKey** 復号に使用する 32 バイトのキーを含むバッファへのポインタ
- **inIV** 復号に使用する 12 バイトの iv を含むバッファへのポインタ
- **inAAD** 任意の長さの追加認証データ (AAD) を含むバッファへのポインタ
- **inAADLen** 入力 AAD の長さ
- **inCiphertext** 復号する暗号文を含むバッファへのポインタ
- **outCiphertextLen** 復号する暗号文の長さ
- **inAuthTag** 認証用の 16 バイトのダイジェストを含むバッファへのポインタ
- **outPlaintext** 平文を格納するバッファへのポインタ

See:

- [wc_Chacha20Poly1305_Encrypt](#)
- [wc_Chacha_*](#)
- [wc_Poly1305*](#)

Return:

- 0 メッセージの復号と認証に成功した場合に返されます
- BAD_FUNC_ARG 関数の引数が期待される内容と一致しない場合に返されます
- MAC_CMP_FAILED_E 生成された認証タグが提供された inAuthTag と一致しない場合に返されます
- MEMORY_E 内部バッファの割り当てに失敗した場合に返されます
- CHACHA_POLY_OVERFLOW 入力が破損している場合に返される可能性があります

Example

```

byte key[]    = { // 32 バイトのキーを初期化 };
byte iv[]     = { // 12 バイトのキーを初期化 };
byte inAAD[]  = { // AAD を初期化 };

byte cipher[] = { // 受信した暗号文で初期化 };
byte authTag[16] = { // 受信した認証タグで初期化 };

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) {
    // 認証中のエラー
} else if( ret != 0) {
    // 関数引数のエラー
}

```

C.8.3 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);

```

C.9 dox_comments/header_files-ja/chacha.h**C.9.1 Functions**

	Name
int	wc_Chacha_SetIV (ChaCha * ctx, const byte * inIv, word32 counter) この関数は、ChaCha オブジェクトの初期化ベクトル (nonce) を設定し、暗号として使用するために初期化します。wc_Chacha_SetKey を使用してキーが設定された後に呼び出す必要があります。暗号化のラウンドごとに異なる nonce を使用する必要があります。
int	wc_Chacha_Process (ChaCha * ctx, byte * cipher, const byte * plain, word32 msglen) この関数は、バッファ input からテキストを処理し、暗号化または復号して、結果をバッファ output に格納します。
int	wc_Chacha_SetKey (ChaCha * ctx, const byte * key, word32 keySz) この関数は、ChaCha オブジェクトのキーを設定し、暗号として使用するために初期化します。wc_Chacha_SetIV で nonce を設定する前、および wc_Chacha_Process で暗号化に使用する前に呼び出す必要があります。

C.9.2 Functions Documentation

C.9.2.1 function wc_Chacha_SetIV

```
int wc_Chacha_SetIV(
    ChaCha * ctx,
    const byte * inIv,
    word32 counter
)
```

この関数は、ChaCha オブジェクトの初期化ベクトル (nonce) を設定し、暗号として使用するために初期化します。wc_Chacha_SetKey を使用してキーが設定された後に呼び出す必要があります。暗号化のラウンドごとに異なる nonce を使用する必要があります。

Parameters:

- **ctx** iv を設定する ChaCha 構造体へのポインタ
- **inIv** ChaCha 構造体を初期化するための 12 バイトの初期化ベクトルを含むバッファへのポインタ
- **counter** ブロックカウンタが開始すべき値-通常はゼロ。

See:

- **wc_Chacha_SetKey**
- **wc_Chacha_Process**

Return:

- 0 初期化ベクトルの設定に成功した場合に返されます
- BAD_FUNC_ARG ctx 入力引数の処理中にエラーが発生した場合に返されます

Example

```
ChaCha enc;
// wc_Chacha_SetKey で enc を初期化
byte iv[12];
// iv を初期化
if( wc_Chacha_SetIV(&enc, iv, 0) != 0) {
```

```
// ChaCha 構造体の初期化エラー
}
```

C.9.2.2 function wc_Chacha_Process

```
int wc_Chacha_Process(
    ChaCha * ctx,
    byte * cipher,
    const byte * plain,
    word32 msglen
)
```

この関数は、バッファ input からテキストを処理し、暗号化または復号して、結果をバッファ output に格納します。

Parameters:

- **ctx** iv を設定する ChaCha 構造体へのポインタ
- **output** 出力暗号文または復号された平文を格納するバッファへのポインタ
- **input** 暗号化する入力平文または復号する入力暗号文を含むバッファへのポインタ
- **msglen** 暗号化するメッセージまたは復号する暗号文の長さ

See:

- [wc_Chacha_SetKey](#)
- [wc_Chacha_Process](#)

Return:

- 0 入力の暗号化または復号に成功した場合に返されます
- BAD_FUNC_ARG ctx 入力引数の処理中にエラーが発生した場合に返されます

Example

```
ChaCha enc;
// wc_Chacha_SetKey と wc_Chacha_SetIV で enc を初期化

byte plain[] = { // 平文を初期化 };
byte cipher[sizeof(plain)];
if( wc_Chacha_Process(&enc, cipher, plain, sizeof(plain)) != 0) {
    // ChaCha 暗号の処理エラー
}
```

C.9.2.3 function wc_Chacha_SetKey

```
int wc_Chacha_SetKey(
    ChaCha * ctx,
    const byte * key,
    word32 keySz
)
```

この関数は、ChaCha オブジェクトのキーを設定し、暗号として使用するために初期化します。wc_Chacha_SetIV で nonce を設定する前、および wc_Chacha_Process で暗号化に使用する前に呼び出す必要があります。

Parameters:

- **ctx** キーを設定する ChaCha 構造体へのポインタ
- **key** ChaCha 構造体を初期化するための 16 バイトまたは 32 バイトのキーを含むバッファへのポインタ

- **keySz** 渡されるキーの長さ

See:

- [wc_Chacha_SetIV](#)
- [wc_Chacha_Process](#)

Return:

- 0 キーの設定に成功した場合に返されます
- BAD_FUNC_ARG ctx 入力引数の処理中にエラーが発生した場合、またはキーが 16 バイトまたは 32 バイトでない場合に返されます

Example

```
ChaCha enc;
byte key[] = { // キーを初期化 };

if( wc_Chacha_SetKey(&enc, key, sizeof(key)) != 0 ) {
    // ChaCha 構造体の初期化エラー
}
```

C.9.3 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);
```

C.10 dox_comments/header_files-ja/cmac.h

C.10.1 Functions

	Name
int	wc_InitCmac (Cmac * cmac, const byte * key, word32 keySz, int type, void * unused)Cmac 構造体をデフォルト値で初期化します
int	wc_InitCmac_ex (Cmac * cmac, const byte * key, word32 keySz, int type, void * unused, void * heap, int devId)Cmac 構造体をデフォルト値で初期化します
int	wc_CmacUpdate (Cmac * cmac, const byte * in, word32 inSz) 暗号ベースメッセージ認証コード入力データを追加します
int	wc_CmacFinalNoFree (Cmac * cmac, byte * out, word32 * outSz) 暗号ベースメッセージ認証コードを使用して最終結果を生成し、コンテキストのクリーンアップを延期します。
int	wc_CmacFinal (Cmac * cmac, byte * out, word32 * outSz) 暗号ベースメッセージ認証コードを使用して最終結果を生成し、wc_CmacFree() でコンテキストをクリーンアップします。

	Name
int	wc_CmacFree (Cmac * cmac)CMAC コンテキスト内の割り当てをクリーンアップします。
int	wc_AesCmacGenerate (byte * out, word32 * outSz, const byte * in, word32 inSz, const byte * key, word32 keySz)CMAC を生成するためのシングルショット関数
int	wc_AesCmacVerify (const byte * check, word32 checkSz, const byte * in, word32 inSz, const byte * key, word32 keySz)CMAC を検証するためのシングルショット関数
int	wc_CMAC_Grow (Cmac * cmac, const byte * in, int inSz) ハードウェアがシングルショットを必要とし、更新をメモリにキャッシュする必要がある場合に WOLFSSL_HASH_KEEP でのみ使用されます

C.10.2 Functions Documentation

C.10.2.1 function wc_InitCmac

```
int wc_InitCmac(
    Cmac * cmac,
    const byte * key,
    word32 keySz,
    int type,
    void * unused
)
```

Cmac 構造体をデフォルト値で初期化します

Parameters:

- **cmac** Cmac 構造体へのポインタ
- **key** キーポインタ
- **keySz** キーポインタのサイズ (16、24、または 32)
- **type** 常に WC_CMAC_AES = 1
- **unused** 使用されません。互換性に関する将来の使用の可能性のために存在します

See:

- **wc_InitCmac_ex**
- **wc_CmacUpdate**
- **wc_CmacFinal**
- **wc_CmacFinalNoFree**
- **wc_CmacFree**

Return: 0 成功時

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.10.2.2 function wc_InitCmac_ex

```
int wc_InitCmac_ex(
    Cmac * cmac,
    const byte * key,
    word32 keySz,
    int type,
    void * unused,
    void * heap,
    int devId
)
```

Cmac 構造体をデフォルト値で初期化します

Parameters:

- **cmac** Cmac 構造体へのポインタ
- **key** キーポインタ
- **keySz** キーポインタのサイズ (16、24、または 32)
- **type** 常に WC_CMAC_AES = 1
- **unused** 使用されません。互換性に関する将来の使用の可能性のために存在します
- **heap** 動的割り当てに使用されるヒープヒントへのポインタ。通常、静的メモリオプションで使用されます。NULL にできます。
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID(-2) に設定します

See:

- [wc_InitCmac_ex](#)
- [wc_CmacUpdate](#)
- [wc_CmacFinal](#)
- [wc_CmacFinalNoFree](#)
- [wc_CmacFree](#)

Return: 0 成功時

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.10.2.3 function wc_CmacUpdate

```
int wc_CmacUpdate(
    Cmac * cmac,
    const byte * in,
    word32 inSz
)
```

暗号ベースメッセージ認証コード入力データを追加します

Parameters:

- **cmac** Cmac 構造体へのポインタ
- **in** 処理する入力データ
- **inSz** 入力データのサイズ

See:

- [wc_InitCmac](#)
- [wc_CmacFinal](#)
- [wc_CmacFinalNoFree](#)
- [wc_CmacFree](#)

Return: 0 成功時

Example

```
ret = wc_CmacUpdate(cmac, in, inSz);
```

C.10.2.4 function wc_CmacFinalNoFree

```
int wc_CmacFinalNoFree(  
    Cmac * cmac,  
    byte * out,  
    word32 * outSz  
)
```

暗号ベースメッセージ認証コードを使用して最終結果を生成し、コンテキストのクリーンアップを延期します。

Parameters:

- **cmac** Cmac 構造体へのポインタ
- **out** 結果を返すポインタ
- **outSz** 出力のポインタサイズ (入出力)

See:

- [wc_InitCmac](#)
- [wc_CmacFinal](#)
- [wc_CmacFinalNoFree](#)
- [wc_CmacFree](#)

Return: 0 成功時

Example

```
ret = wc_CmacFinalNoFree(cmac, out, &outSz);  
(void)wc_CmacFree(cmac);
```

C.10.2.5 function wc_CmacFinal

```
int wc_CmacFinal(  
    Cmac * cmac,  
    byte * out,  
    word32 * outSz  
)
```

暗号ベースメッセージ認証コードを使用して最終結果を生成し、wc_CmacFree() でコンテキストをクリーンアップします。

Parameters:

- **cmac** Cmac 構造体へのポインタ
- **out** 結果を返すポインタ
- **outSz** 出力のポインタサイズ (入出力)

See:

- [wc_InitCmac](#)
- [wc_CmacFinalNoFree](#)
- [wc_CmacFinalNoFree](#)
- [wc_CmacFree](#)

Return: 0 成功時

Example

```
ret = wc_CmacFinal(cmac, out, &outSz);
```

C.10.2.6 function wc_CmacFree

```
int wc_CmacFree(  
    Cmac * cmac  
)
```

CMAC コンテキスト内の割り当てをクリーンアップします。

Parameters:

- **cmac** Cmac 構造体へのポインタ

See:

- [wc_InitCmac](#)
- [wc_CmacFinalNoFree](#)
- [wc_CmacFinal](#)
- [wc_CmacFree](#)

Return: 0 成功時

Example

```
ret = wc_CmacFinalNoFree(cmac, out, &outSz);  
(void)wc_CmacFree(cmac);
```

C.10.2.7 function wc_AesCmacGenerate

```
int wc_AesCmacGenerate(  
    byte * out,  
    word32 * outSz,  
    const byte * in,  
    word32 inSz,  
    const byte * key,  
    word32 keySz  
)
```

CMAC を生成するためのシングルショット関数

Parameters:

- **out** 結果を返すポインタ
- **outSz** 出力のポインタサイズ (入出力)
- **in** 処理する入力データ

- **inSz** 入力データのサイズ
- **key** キーポインタ
- **keySz** キーポインタのサイズ (16、24、または 32)

See: [wc_AesCmacVerify](#)

Return: 0 成功時

Example

```
ret = wc_AesCmacGenerate(mac, &macSz, msg, msgSz, key, keySz);
```

C.10.2.8 function wc_AesCmacVerify

```
int wc_AesCmacVerify(  
    const byte * check,  
    word32 checkSz,  
    const byte * in,  
    word32 inSz,  
    const byte * key,  
    word32 keySz  
)
```

CMAC を検証するためのシングルショット関数

Parameters:

- **check** 検証する CMAC 値
- **checkSz** check バッファのサイズ
- **in** 処理する入力データ
- **inSz** 入力データのサイズ
- **key** キーポインタ
- **keySz** キーポインタのサイズ (16、24、または 32)

See: [wc_AesCmacGenerate](#)

Return: 0 成功時

Example

```
ret = wc_AesCmacVerify(mac, macSz, msg, msgSz, key, keySz);
```

C.10.2.9 function wc_CMAC_Grow

```
int wc_CMAC_Grow(  
    Cmac * cmac,  
    const byte * in,  
    int inSz  
)
```

ハードウェアがシングルショットを必要とし、更新をメモリにキャッシュする必要がある場合に WOLF-SSL_HASH_KEEP でのみ使用されます

Parameters:

- **in** 処理する入力データ
- **inSz** 入力データのサイズ

Return: 0 成功時

Example

```
ret = wc_CMAC_Grow(cmac, in, inSz)
```

C.10.3 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);

```

C.11 dox_comments/header_files-ja/coding.h

C.11.1 Functions

	Name
int	Base64_Decode (const byte * in, word32 inLen, byte * out, word32 * outLen) この関数は、指定された Base64 エンコードされた入力 in を復号し、結果を出力バッファ out に格納します。また、変数 outLen に出力バッファに書き込まれたサイズを設定します。
int	Base64_Encode (const byte * in, word32 inLen, byte * out, word32 * outLen) この関数は、指定された入力 in をエンコードし、Base64 エンコードされた結果を出力バッファ out に格納します。エスケープされた %0A 行末の代わりに、従来の'' 行末でデータを書き込みます。正常に完了すると、この関数は outLen を出力バッファに書き込まれたバイト数に設定します。

	Name
int	Base64_EncodeEsc (const byte * in, word32 inLen, byte * out, word32 * outLen) この関数は、指定された入力 in をエンコードし、Base64 エンコードされた結果を出力バッファ out に格納します。'' 行末の代わりに%0A エスケープされた行末でデータを書き込みます。正常に完了すると、この関数は outLen を出力バッファに書き込まれたバイト数に設定します。
int	Base64_Encode_NoNL (const byte * in, word32 inLen, byte * out, word32 * outLen) この関数は、指定された入力 in をエンコードし、Base64 エンコードされた結果を出力バッファ out に格納します。改行なしでデータを書き込みます。正常に完了すると、この関数は outLen を出力バッファに書き込まれたバイト数に設定します。
int	Base16_Decode (const byte * in, word32 inLen, byte * out, word32 * outLen) この関数は、指定された Base16 エンコードされた入力 in を復号し、結果を出力バッファ out に格納します。また、変数 outLen に出力バッファに書き込まれたサイズを設定します。
int	Base16_Encode (const byte * in, word32 inLen, byte * out, word32 * outLen) 入力を base16 出力にエンコードします。

C.11.2 Functions Documentation

C.11.2.1 function Base64_Decode

```
int Base64_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この関数は、指定された Base64 エンコードされた入力 in を復号し、結果を出力バッファ out に格納します。また、変数 outLen に出力バッファに書き込まれたサイズを設定します。

Parameters:

- **in** 復号する入力バッファへのポインタ
- **inLen** 復号する入力バッファの長さ
- **out** 復号されたメッセージを格納する出力バッファへのポインタ
- **outLen** 出力バッファの長さへのポインタ。関数呼び出しの最後に書き込まれたバイト数で更新されます

See:

- [Base64_Encode](#)
- [Base16_Decode](#)

Return:

- 0 Base64 エンコードされた入力の復号に成功した場合に返されます
- BAD_FUNC_ARG 出力バッファが復号された入力を格納するには小さすぎる場合に返されます

- ASN_INPUT_E 入力バッファ内の文字が Base64 の範囲 ([A-Za-z0-9+/=]) 外にある場合、または Base64 エンコードされた入力に無効な行末がある場合に返されます

Example

```
byte encoded[] = { // 復号するテキストを初期化 };
byte decoded[sizeof(encoded)];
// 少なくとも (sizeof(encoded) * 3 + 3) / 4 のスペースが必要

int outLen = sizeof(decoded);

if( Base64_Decode(encoded, sizeof(encoded), decoded, &outLen) != 0 ) {
    // 入力バッファの復号エラー
}
```

C.11.2.2 function Base64_Encode

```
int Base64_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この関数は、指定された入力 in をエンコードし、Base64 エンコードされた結果を出力バッファ out に格納します。エスケープされた%0A 行末の代わりに、従来の' '行末でデータを書き込みます。正常に完了すると、この関数は outLen を出力バッファに書き込まれたバイト数に設定します。

Parameters:

- **in** エンコードする入力バッファへのポインタ
- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを格納する出力バッファへのポインタ
- **outLen** エンコードされたメッセージを格納する出力バッファの長さへのポインタ

See:

- [Base64_EncodeEsc](#)
- [Base64_Decode](#)

Return:

- 0 Base64 エンコードされた入力の復号に成功した場合に返されます
- BAD_FUNC_ARG 出力バッファがエンコードされた入力を格納するには小さすぎる場合に返されます
- BUFFER_E エンコード中に出力バッファのスペースが不足した場合に返されます

Example

```
byte plain[] = { // エンコードするテキストを初期化 };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_Encode(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // 入力バッファのエンコードエラー
}
```

C.11.2.3 function Base64_EncodeEsc

```
int Base64_EncodeEsc(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この関数は、指定された入力 `in` をエンコードし、Base64 エンコードされた結果を出力バッファ `out` に格納します。行末の代わりに %0A エスケープされた行末でデータを書き込みます。正常に完了すると、この関数は `outLen` を出力バッファに書き込まれたバイト数に設定します。

Parameters:

- **in** エンコードする入力バッファへのポインタ
- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを格納する出力バッファへのポインタ
- **outLen** エンコードされたメッセージを格納する出力バッファの長さへのポインタ

See:

- [Base64_Encode](#)
- [Base64_Decode](#)

Return:

- 0 Base64 エンコードされた入力の復号に成功した場合に返されます
- BAD_FUNC_ARG 出力バッファがエンコードされた入力を格納するには小さすぎる場合に返されます
- BUFFER_E エンコード中に出力バッファのスペースが不足した場合に返されます
- ASN_INPUT_E 入力メッセージの復号処理中にエラーが発生した場合に返されます

Example

```
byte plain[] = { // エンコードするテキストを初期化 };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_EncodeEsc(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // 入力バッファのエンコードエラー
}
```

C.11.2.4 function Base64_Encode_NoNL

```
int Base64_Encode_NoNL(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この関数は、指定された入力 `in` をエンコードし、Base64 エンコードされた結果を出力バッファ `out` に格納します。改行なしでデータを書き込みます。正常に完了すると、この関数は `outLen` を出力バッファに書き込まれたバイト数に設定します。

Parameters:

- **in** エンコードする入力バッファへのポインタ
- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを格納する出力バッファへのポインタ
- **outLen** エンコードされたメッセージを格納する出力バッファの長さへのポインタ

See:

- [Base64_Encode](#)
- [Base64_Decode](#)

Return:

- 0 Base64 エンコードされた入力の復号に成功した場合に返されます
- BAD_FUNC_ARG 出力バッファがエンコードされた入力を格納するには小さすぎる場合に返されます
- BUFFER_E エンコード中に出力バッファのスペースが不足した場合に返されます
- ASN_INPUT_E 入力メッセージの復号処理中にエラーが発生した場合に返されます

Example

```
byte plain[] = { // エンコードするテキストを初期化 };
byte encoded[MAX_BUFFER_SIZE];
int outLen = sizeof(encoded);
if( Base64_Encode_NoNl(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // 入力バッファのエンコードエラー
}
```

C.11.2.5 function Base16_Decode

```
int Base16_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この関数は、指定された Base16 エンコードされた入力 in を復号し、結果を出力バッファ out に格納します。また、変数 outLen に出力バッファに書き込まれたサイズを設定します。

Parameters:

- **in** 復号する入力バッファへのポインタ
- **inLen** 復号する入力バッファの長さ
- **out** 復号されたメッセージを格納する出力バッファへのポインタ
- **outLen** 出力バッファの長さへのポインタ。関数呼び出しの最後に書き込まれたバイト数で更新されます

See:

- [Base64_Encode](#)
- [Base64_Decode](#)
- [Base16_Encode](#)

Return:

- 0 Base16 エンコードされた入力の復号に成功した場合に返されます
- BAD_FUNC_ARG 出力バッファが復号された入力を格納するには小さすぎる場合、または入力長が 2 の倍数でない場合に返されます
- ASN_INPUT_E 入力バッファ内の文字が Base16 の範囲 ([0-9A-F]) 外にある場合に返されます

Example

```
byte encoded[] = { // 復号するテキストを初期化 };
byte decoded[sizeof(encoded)];
int outLen = sizeof(decoded);

if( Base16_Decode(encoded, sizeof(encoded), decoded, &outLen) != 0 ) {
```

```

    // 入力バッファの復号エラー
}

```

C.11.2.6 function Base16_Encode

```

int Base16_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)

```

入力を base16 出力にエンコードします。

Parameters:

- **in** エンコードする入力バッファへのポインタ。
- **inLen** 入力バッファの長さ。
- **out** 出力バッファへのポインタ。
- **outLen** 出力バッファの長さ。エンコードされた出力の長さに設定されます。

See:

- [Base64_Encode](#)
- [Base64_Decode](#)
- [Base16_Decode](#)

Return:

- 0 成功
- BAD_FUNC_ARG in、out、または outLen が null の場合、または outLen が inLen の 2 倍プラス 1 未満の場合に返されます。

Example

```

byte in[] = { // エンコードする何かの内容 };
byte out[NECESSARY_OUTPUT_SIZE];
word32 outSz = sizeof(out);

if(Base16_Encode(in, sizeof(in), out, &outSz) != 0)
{
    // エンコードエラーを処理
}

```

C.11.3 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);
```

C.12 dox_comments/header_files-ja/compress.h

C.12.1 Functions

	Name
int	wc_Compress (byte * out, word32 outSz, const byte * in, word32 inSz, word32 flags) この関数は、ハフマン符号化を使用して指定された入力データを圧縮し、出力を out に格納します。出力バッファは入力バッファよりも大きくする必要がありますことに注意してください。圧縮が不可能な特定の入力が存在する場合でも、ルックアップテーブルが必要になるためです。出力バッファには srcSz + 0.1% + 12 を割り当てることが推奨されます。
int	wc_DeCompress (byte * out, word32 outSz, const byte * in, word32 inSz) この関数は、ハフマン符号化を使用して指定された圧縮データを解凍し、出力を out に格納します。

C.12.2 Functions Documentation

C.12.2.1 function wc_Compress

```
int wc_Compress(  
    byte * out,  
    word32 outSz,  
    const byte * in,  
    word32 inSz,  
    word32 flags  
)
```

この関数は、ハフマン符号化を使用して指定された入力データを圧縮し、出力を out に格納します。出力バッファは入力バッファよりも大きくする必要がありますことに注意してください。圧縮が不可能な特定の入力が存在する場合でも、ルックアップテーブルが必要になるためです。出力バッファには srcSz + 0.1% + 12 を割り当てることが推奨されます。

Parameters:

- **out** 圧縮されたデータを格納する出力バッファへのポインタ
- **outSz** 格納に使用できる出力バッファのサイズ
- **in** 圧縮するメッセージを含むバッファへのポインタ
- **inSz** 圧縮する入力メッセージのサイズ
- **flags** 圧縮の動作を制御するフラグ。通常の解凍には 0 を使用します

See: [wc_DeCompress](#)

Return:

- 入力データの圧縮に成功した場合、出力バッファに格納されたバイト数を返します
- COMPRESS_INIT_E 圧縮用のストリームの初期化中にエラーが発生した場合に返されます
- COMPRESS_E 圧縮中にエラーが発生した場合に返されます

Example

```
byte message[] = { // 圧縮するテキストを初期化 };
byte compressed[(sizeof(message) + sizeof(message) * .001 + 12)];
// 少なくとも srcSz + .1% + 12 を推奨

if( wc_Compress(compressed, sizeof(compressed), message, sizeof(message),
0) != 0){
    // データの圧縮エラー
}
```

C.12.2.2 function wc_DeCompress

```
int wc_DeCompress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz
)
```

この関数は、ハフマン符号化を使用して指定された圧縮データを解凍し、出力を out に格納します。

Parameters:

- **out** 解凍されたデータを格納する出力バッファへのポインタ
- **outSz** 格納に使用できる出力バッファのサイズ
- **in** 解凍するメッセージを含むバッファへのポインタ
- **inSz** 解凍する入力メッセージのサイズ

See: [wc_Compress](#)

Return:

- Success 入力データの解凍に成功した場合、出力バッファに格納されたバイト数を返します
- COMPRESS_INIT_E: 圧縮用のストリームの初期化中にエラーが発生した場合に返されます
- COMPRESS_E: 圧縮中にエラーが発生した場合に返されます

Example

```
byte compressed[] = { // 圧縮されたメッセージを初期化 };
byte decompressed[MAX_MESSAGE_SIZE];

if( wc_DeCompress(decompressed, sizeof(decompressed),
compressed, sizeof(compressed)) != 0 ) {
    // データの解凍エラー
}
```

C.12.3 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);
```

C.13 dox_comments/header_files-ja/cryptocb.h**C.13.1 Functions**

	Name
int	wc_CryptoCb_RegisterDevice (int devId, CryptoDevCallbackFunc cb, void * ctx) この関数は、Key Store、Secure Element、HSM、PKCS11、TPM などの外部ハードウェアへの暗号操作のオフロードのために、一意のデバイス識別子 (devID) とコールバック関数を登録します。
void	wc_CryptoCb_UnRegisterDevice (int devId) この関数は、一意のデバイス識別子 (devID) のコールバック関数の登録を解除します。

C.13.2 Functions Documentation

C.13.2.1 function wc_CryptoCb_RegisterDevice

```
int wc_CryptoCb_RegisterDevice(
    int devId,
    CryptoDevCallbackFunc cb,
    void * ctx
)
```

この関数は、Key Store、Secure Element、HSM、PKCS11、TPM などの外部ハードウェアへの暗号操作のオフロードのために、一意のデバイス識別子 (devID) とコールバック関数を登録します。

Parameters:

- **devId** -2(INVALID_DEVID) 以外の任意の一意の値
- **cb** 次のプロトタイプを持つコールバック関数: typedef int (CryptoDevCallbackFunc)(int devId, wc_CryptoInfo info, void* ctx);

See:

- **wc_CryptoCb_UnRegisterDevice**
- **wolfSSL_SetDevId**
- **wolfSSL_CTX_SetDevId**

Return:

- CRYPTO_CB_UNAVAILABLE ソフトウェア暗号の使用にフォールバックする場合
- 0 成功時
- 負の値 失敗時

Crypto Callbacks を使用した STSAFE の例については、wolfcrypt/src/port/st/stsafe.c と wolfSSL_STSAFE_CryptoDevCb 関数を参照してください。

TPM ベースの暗号コールバックの例については、wolfTPM src/tpm2_wrap.c の wolfTPM2_CryptoDevCb 関数を参照してください。

Example

```
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/cryptocb.h>
static int myCryptoCb_Func(int devId, wc_CryptoInfo* info, void* ctx)
{
    int ret = CRYPTO_CB_UNAVAILABLE;

    if (info->algo_type == WC_ALGO_TYPE_PK) {
#ifdef NO_RSA
        if (info->pk.type == WC_PK_TYPE_RSA) {
```

```

        switch (info->pk.rsa.type) {
            case RSA_PUBLIC_ENCRYPT:
            case RSA_PUBLIC_DECRYPT:
                // RSA 公開鍵操作
                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 秘密鍵操作
                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
#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 署名
        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.13.2.2 function wc_CryptoCb_UnRegisterDevice

```

void wc_CryptoCb_UnRegisterDevice(
    int devId

```

)

この関数は、一意のデバイス識別子 (devID) のコールバック関数の登録を解除します。

Parameters:

- **devId** -2(INVALID_DEVID) 以外の任意の一意の値

See:

- [wc_CryptoCb_RegisterDevice](#)
- [wolfSSL_SetDevId](#)
- [wolfSSL_CTX_SetDevId](#)

Return: none 戻り値なし。

Example

```
wc_CryptoCb_UnRegisterDevice(devId);
devId = INVALID_DEVID;
wolfSSL_CTX_SetDevId(ctx, devId);
```

C.13.3 Source code

```
int wc_CryptoCb_RegisterDevice(int devId, CryptoDevCallbackFunc cb, void*
↪ ctx);
```

```
void wc_CryptoCb_UnRegisterDevice(int devId);
```

C.14 dox_comments/header_files-ja/curve25519.h

C.14.1 Functions

	Name
int	wc_curve25519_make_key (WC_RNG * rng, int keysize, curve25519_key * key) この関数は、与えられた乱数生成器 rng を使用して、与えられたサイズ (keysize) の Curve25519 鍵を生成し、与えられた curve25519_key 構造体に格納します。wc_curve25519_init() を通じて鍵構造体が初期化された後に呼び出す必要があります。
int	wc_curve25519_shared_secret (curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen) この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンのみをサポートします。
int	wc_curve25519_shared_secret_ex (curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen, int endian) この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンとリトルエンディアンの両方をサポートします。

	Name
int	wc_curve25519_init (curve25519_key * key) この関数は Curve25519 鍵を初期化します。構造体の鍵を生成する前に呼び出す必要があります。
void	wc_curve25519_free (curve25519_key * key) この関数は Curve25519 オブジェクトを解放します。
int	wc_curve25519_import_private (const byte * priv, word32 privSz, curve25519_key * key) この関数は curve25519 秘密鍵のみをインポートします。(ビッグエンディアン)
int	wc_curve25519_import_private_ex (const byte * priv, word32 privSz, curve25519_key * key, int endian) curve25519 秘密鍵のみのインポート。(ビッグエンディアンまたはリトルエンディアン)
int	wc_curve25519_import_private_raw (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key) この関数は、公開鍵-秘密鍵ペアを curve25519_key 構造体にインポートします。ビッグエンディアンのみ。
int	wc_curve25519_import_private_raw_ex (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key, int endian) この関数は、公開鍵-秘密鍵ペアを curve25519_key 構造体にインポートします。ビッグエンディアンとリトルエンディアンの両方をサポートします。
int	wc_curve25519_export_private_raw (curve25519_key * key, byte * out, word32 * outLen) この関数は、curve25519_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンのみ。
int	wc_curve25519_export_private_raw_ex (curve25519_key * key, byte * out, word32 * outLen, int endian) この関数は、curve25519_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンまたはリトルエンディアンを指定できます。
int	wc_curve25519_import_public (const byte * in, word32 inLen, curve25519_key * key) この関数は、与えられた in バッファから公開鍵をインポートし、curve25519_key 構造体に格納します。
int	wc_curve25519_import_public_ex (const byte * in, word32 inLen, curve25519_key * key, int endian) この関数は、与えられた in バッファから公開鍵をインポートし、curve25519_key 構造体に格納します。

	Name
int	wc_curve25519_check_public (const byte * pub, word32 pubSz, int endian) この関数は、エンディアン順序を考慮して、公開鍵バッファが有効な Curve25519 鍵値を保持しているかどうかをチェックします。
int	wc_curve25519_export_public (curve25519_key * key, byte * out, word32 * outLen) この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。
int	wc_curve25519_export_public_ex (curve25519_key * key, byte * out, word32 * outLen, int endian) この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンとリトルエンディアンの両方をサポートします。
int	wc_curve25519_export_key_raw (curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) Curve25519 鍵ペアをエクスポートします。ビッグエンディアンのみ。
int	wc_curve25519_export_key_raw_ex (curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian) curve25519 鍵ペアをエクスポートします。ビッグエンディアンまたはリトルエンディアン。
int	wc_curve25519_size (curve25519_key * key) この関数は、与えられた鍵構造体の鍵サイズを返します。

C.14.2 Functions Documentation

C.14.2.1 function wc_curve25519_make_key

```
int wc_curve25519_make_key(
    WC_RNG * rng,
    int keysize,
    curve25519_key * key
)
```

この関数は、与えられた乱数生成器 rng を使用して、与えられたサイズ (keysizes) の Curve25519 鍵を生成し、与えられた curve25519_key 構造体に格納します。wc_curve25519_init() を通じて鍵構造体が初期化された後に呼び出す必要があります。

Parameters:

- **rng** ecc 鍵を生成するために使用される RNG オブジェクトへのポインタ。
- **keysizes** 生成する鍵のサイズ。curve25519 では 32 バイトである必要があります。
- **key** 生成された鍵を格納する curve25519_key 構造体へのポインタ。

See: [wc_curve25519_init](#)

Return:

- 0 鍵の生成に成功し、与えられた curve25519_key 構造体に格納された場合に返されます。

- ECC_BAD_ARG_E 入力 keysize が curve25519 鍵の keysize に対応していない場合 (32 バイト) に返されます。
- RNG_FAILURE_E rng 内部ステータスが DRBG_OK でない場合、または rng で次のランダムブロックを生成する際にエラーがある場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

curve25519_key key;
wc_curve25519_init(&key); // 鍵を初期化
WC_RNG rng;
wc_InitRng(&rng); // 乱数生成器を初期化

ret = wc_curve25519_make_key(&rng, 32, &key);
if (ret != 0) {
    // Curve25519 鍵作成エラー
}
```

C.14.2.2 function wc_curve25519_shared_secret

```
int wc_curve25519_shared_secret(
    curve25519_key * private_key,
    curve25519_key * public_key,
    byte * out,
    word32 * outlen
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンのみをサポートします。

Parameters:

- **private_key** ユーザーの秘密鍵で初期化された curve25519_key 構造体へのポインタ。
- **public_key** 受信した公開鍵を含む curve25519_key 構造体へのポインタ。
- **out** 32 バイトの計算された秘密鍵を格納するバッファへのポインタ。
- **outlen** 出力バッファに書き込まれた長さを格納するポインタ。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_make_key](#)
- [wc_curve25519_shared_secret_ex](#)

Return:

- 0 共有秘密鍵の計算に成功した場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E 実装フィンガープリントを回避するために、公開鍵の最初のビットが設定されている場合に返されます。

Example

```
int ret;

byte sharedKey[32];
word32 keySz;
curve25519_key privKey, pubKey;
// 両方の鍵を初期化
```



```
ret = wc_curve25519_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
if (ret != 0) {
    // 共有鍵生成エラー
}
```

C.14.2.3 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
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **private_key** ユーザーの秘密鍵で初期化された curve25519_key 構造体へのポインタ。
- **public_key** 受信した公開鍵を含む curve25519_key 構造体へのポインタ。
- **out** 32 バイトの計算された秘密鍵を格納するバッファへのポインタ。
- **outlen** 出力バッファに書き込まれた長さを格納するポインタ。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_make_key](#)
- [wc_curve25519_shared_secret](#)

Return:

- 0 共有秘密鍵の計算に成功した場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E 実装フィンガープリントを回避するために、公開鍵の最初のビットが設定されている場合に返されます。

Example

```
int ret;

byte sharedKey[32];
word32 keySz;

curve25519_key privKey, pubKey;
// 両方の鍵を初期化

ret = wc_curve25519_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
    EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 共有鍵生成エラー
}
```


C.14.2.4 function wc_curve25519_init

```
int wc_curve25519_init(  
    curve25519_key * key  
)
```

この関数は Curve25519 鍵を初期化します。構造体の鍵を生成する前に呼び出す必要があります。

Parameters:

- **key** 初期化する curve25519_key 構造体へのポインタ。

See: [wc_curve25519_make_key](#)

Return:

- 0 curve25519_key 構造体の初期化に成功した場合に返されます。
- BAD_FUNC_ARG key が NULL の場合に返されます。

Example

```
curve25519_key key;  
wc_curve25519_init(&key); // 鍵を初期化  
// 鍵を作成し、暗号化に進む
```

C.14.2.5 function wc_curve25519_free

```
void wc_curve25519_free(  
    curve25519_key * key  
)
```

この関数は Curve25519 オブジェクトを解放します。

Parameters:

- **key** 解放する鍵オブジェクトへのポインタ。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_make_key](#)

Example

```
curve25519_key privKey;  
// 鍵を初期化し、共有秘密鍵の生成に使用  
wc_curve25519_free(&privKey);
```

C.14.2.6 function wc_curve25519_import_private

```
int wc_curve25519_import_private(  
    const byte * priv,  
    word32 privSz,  
    curve25519_key * key  
)
```

この関数は curve25519 秘密鍵のみをインポートします。(ビッグエンディアン)

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。

See:

- `wc_curve25519_import_private_ex`
- `wc_curve25519_size`

Return:

- 0 秘密鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG key または priv が null の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE25519_KEY_SIZE と等しくない場合に返されます。

Example

```
int ret;

byte priv[] = { // 秘密鍵の内容 };
curve25519_key key;
wc_curve25519_init(&key);

ret = wc_curve25519_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

C.14.2.7 function wc_curve25519_import_private_ex

```
int wc_curve25519_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve25519_key * key,
    int endian
)
```

curve25519 秘密鍵のみのインポート。(ビッグエンディアンまたはリトルエンディアン)

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- `wc_curve25519_import_private`
- `wc_curve25519_size`

Return:

- 0 秘密鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG key または priv が null の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE25519_KEY_SIZE と等しくない場合に返されます。

Example

```
int ret;

byte priv[] = { // 秘密鍵の内容 };
curve25519_key key;
wc_curve25519_init(&key);
```

```
ret = wc_curve25519_import_private_ex(priv, sizeof(priv), &key,
    EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

C.14.2.8 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
)
```

この関数は、公開鍵-秘密鍵ペアを curve25519_key 構造体にインポートします。ビッグエンディアンのみ。

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **pub** インポートする公開鍵を含むバッファへのポインタ。
- **pubSz** インポートする公開鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_make_key](#)
- [wc_curve25519_import_public](#)
- [wc_curve25519_export_private_raw](#)

Return:

- 0 curve25519_key 構造体へのインポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが null の場合に返されます。
- ECC_BAD_ARG_E 入力鍵の鍵サイズが公開鍵または秘密鍵のサイズと一致しない場合に返されます。

Example

```
int ret;

byte priv[32];
byte pub[32];
// 公開鍵と秘密鍵で初期化
curve25519_key key;

wc_curve25519_init(&key);
// 鍵を初期化

ret = wc_curve25519_import_private_raw(&priv, sizeof(priv), pub,
    sizeof(pub), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

C.14.2.9 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
)
```

この関数は、公開鍵-秘密鍵ペアを curve25519_key 構造体にインポートします。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **pub** インポートする公開鍵を含むバッファへのポインタ。
- **pubSz** インポートする公開鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_make_key](#)
- [wc_curve25519_import_public](#)
- [wc_curve25519_export_private_raw](#)
- [wc_curve25519_import_private_raw](#)

Return:

- 0 curve25519_key 構造体へのインポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが null の場合に返されます。
- ECC_BAD_ARG_E 入力鍵の鍵サイズが公開鍵または秘密鍵のサイズと一致しない場合に返されます。

Example

```
int ret;
byte priv[32];
byte pub[32];
// 公開鍵と秘密鍵で初期化
curve25519_key key;

wc_curve25519_init(&key);
// 鍵を初期化

ret = wc_curve25519_import_private_raw_ex(&priv, sizeof(priv), pub,
    sizeof(pub), &key, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

C.14.2.10 function wc_curve25519_export_private_raw

```
int wc_curve25519_export_private_raw(
    curve25519_key * key,
    byte * out,
```

```
    word32 * outLen
)
```

この関数は、curve25519_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンのみ。

Parameters:

- **key** 鍵をエクスポートする構造体へのポインタ。
- **out** エクスポートされた鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_make_key](#)
- [wc_curve25519_import_private_raw](#)
- [wc_curve25519_export_private_raw_ex](#)

Return:

- 0 curve25519_key 構造体から秘密鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E [wc_curve25519_size\(\)](#) が key と等しくない場合に返されます。

Example

```
int ret;
byte priv[32];
int privSz;

curve25519_key key;
// 鍵を初期化して作成

ret = wc_curve25519_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

C.14.2.11 function wc_curve25519_export_private_raw_ex

```
int wc_curve25519_export_private_raw_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

この関数は、curve25519_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンまたはリトルエンディアンを指定できます。

Parameters:

- **key** 鍵をエクスポートする構造体へのポインタ。
- **out** エクスポートされた鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- `wc_curve25519_init`
- `wc_curve25519_make_key`
- `wc_curve25519_import_private_raw`
- `wc_curve25519_export_private_raw`
- `wc_curve25519_size`

Return:

- 0 `curve25519_key` 構造体から秘密鍵のエクスポートに成功した場合に返されます。
- `BAD_FUNC_ARG` 入力パラメータのいずれかが `NULL` の場合に返されます。
- `ECC_BAD_ARG_E` `wc_curve25519_size()` が `key` と等しくない場合に返されます。

Example

```
int ret;

byte priv[32];
int privSz;
curve25519_key key;
// 鍵を初期化して作成
ret = wc_curve25519_export_private_raw_ex(&key, priv, &privSz,
    EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

C.14.2.12 function `wc_curve25519_import_public`

```
int wc_curve25519_import_public(
    const byte * in,
    word32 inLen,
    curve25519_key * key
)
```

この関数は、与えられた `in` バッファから公開鍵をインポートし、`curve25519_key` 構造体に格納します。

Parameters:

- **in** インポートする公開鍵を含むバッファへのポインタ。
- **inLen** インポートする公開鍵の長さ。
- **key** 鍵を格納する `curve25519_key` 構造体へのポインタ。

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 `curve25519_key` 構造体への公開鍵のインポートに成功した場合に返されます。
- `ECC_BAD_ARG_E` `inLen` パラメータが鍵構造体の鍵サイズと一致しない場合に返されます。
- `BAD_FUNC_ARG` 入力パラメータのいずれかが `NULL` の場合に返されます。

Example

```

int ret;

byte pub[32];
// pub を公開鍵で初期化

curve25519_key key;
// 鍵を初期化

ret = wc_curve25519_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}

```

C.14.2.13 function wc_curve25519_import_public_ex

```

int wc_curve25519_import_public_ex(
    const byte * in,
    word32 inLen,
    curve25519_key * key,
    int endian
)

```

この関数は、与えられた in バッファから公開鍵をインポートし、curve25519_key 構造体に格納します。

Parameters:

- **in** インポートする公開鍵を含むバッファへのポインタ。
- **inLen** インポートする公開鍵の長さ。
- **key** 鍵を格納する curve25519_key 構造体へのポインタ。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

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 curve25519_key 構造体への公開鍵のインポートに成功した場合に返されます。
- ECC_BAD_ARG_E inLen パラメータが鍵構造体の鍵サイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```

int ret;

byte pub[32];
// pub を公開鍵で初期化
curve25519_key key;
// 鍵を初期化

ret = wc_curve25519_import_public_ex(pub, sizeof(pub), &key,
    EC25519_BIG_ENDIAN);
if (ret != 0) {

```

```
    // 鍵のインポートエラー  
}
```

C.14.2.14 function wc_curve25519_check_public

```
int wc_curve25519_check_public(  
    const byte * pub,  
    word32 pubSz,  
    int endian  
)
```

この関数は、エンディアン順序を考慮して、公開鍵バッファが有効な Curve25519 鍵値を保持しているかどうかをチェックします。

Parameters:

- **pub** チェックする公開鍵を含むバッファへのポインタ。
- **pubSz** チェックする公開鍵の長さ。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_import_public](#)
- [wc_curve25519_import_public_ex](#)
- [wc_curve25519_size](#)

Return:

- 0 公開鍵の値が有効な場合に返されます。
- ECC_BAD_ARG_E 公開鍵の値が有効でない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;  
  
byte pub[] = { // 公開鍵の内容 };  
  
ret = wc_curve25519_check_public_ex(pub, sizeof(pub), EC25519_BIG_ENDIAN);  
if (ret != 0) {  
    // 鍵のインポートエラー  
}
```

C.14.2.15 function wc_curve25519_export_public

```
int wc_curve25519_export_public(  
    curve25519_key * key,  
    byte * out,  
    word32 * outLen  
)
```

この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。

Parameters:

- **key** 鍵をエクスポートする curve25519_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。

- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_export_private_raw](#)
- [wc_curve25519_import_public](#)

Return:

- 0 curve25519_key 構造体から公開鍵のエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E outLen が CURVE25519_PUB_KEY_SIZE より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// 鍵を初期化して作成
ret = wc_curve25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

C.14.2.16 function wc_curve25519_export_public_ex

```
int wc_curve25519_export_public_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **key** 鍵をエクスポートする curve25519_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_export_private_raw](#)
- [wc_curve25519_import_public](#)

Return:

- 0 curve25519_key 構造体から公開鍵のエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E outLen が CURVE25519_PUB_KEY_SIZE より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// 鍵を初期化して作成

ret = wc_curve25519_export_public_ex(&key, pub, &pubSz, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

C.14.2.17 function wc_curve25519_export_key_raw

```
int wc_curve25519_export_key_raw(
    curve25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

Curve25519 鍵ペアをエクスポートします。ビッグエンディアンのみ。

Parameters:

- **key** 鍵ペアをエクスポートする curve448_key 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** 入力時は、priv バッファのバイト単位のサイズ。出力時は、priv バッファに書き込まれたバイト数を格納します。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** 入力時は、pub バッファのバイト単位のサイズ。出力時は、pub バッファに書き込まれたバイト数を格納します。

See:

- [wc_curve25519_export_key_raw_ex](#)
- [wc_curve25519_export_private_raw](#)

Return:

- 0 curve25519_key 構造体から鍵ペアのエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE25519_KEY_SIZE より小さい、または pubSz が CURVE25519_PUB_KEY_SIZE より小さい場合に返されます。

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
```

```
// 鍵を初期化して作成
```

```
ret = wc_curve25519_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

C.14.2.18 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
)
```

curve25519 鍵ペアをエクスポートします。ビッグエンディアンまたはリトルエンディアン。

Parameters:

- **key** 鍵ペアをエクスポートする curve448_key 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** 入力時は、priv バッファのバイト単位のサイズ。出力時は、priv バッファに書き込まれたバイト数を格納します。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** 入力時は、pub バッファのバイト単位のサイズ。出力時は、pub バッファに書き込まれたバイト数を格納します。
- **endian** 使用する形式を設定するための EC25519_BIG_ENDIAN または EC25519_LITTLE_ENDIAN。

See:

- [wc_curve25519_export_key_raw](#)
- [wc_curve25519_export_private_raw_ex](#)
- [wc_curve25519_export_public_ex](#)

Return:

- 0 curve25519_key 構造体から鍵ペアのエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE25519_KEY_SIZE より小さい、または pubSz が CURVE25519_PUB_KEY_SIZE より小さい場合に返されます。

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
// 鍵を初期化して作成

ret = wc_curve25519_export_key_raw_ex(&key, priv, &privSz, pub, &pubSz,
    EC25519_BIG_ENDIAN);
if (ret != 0) {
```

```
// 鍵のエクスポートエラー
}
```

C.14.2.19 function wc_curve25519_size

```
int wc_curve25519_size(
    curve25519_key * key
)
```

この関数は、与えられた鍵構造体の鍵サイズを返します。

Parameters:

- **key** 鍵サイズを決定する curve25519_key 構造体へのポインタ。

See:

- [wc_curve25519_init](#)
- [wc_curve25519_make_key](#)

Return:

- Success 有効で初期化された curve25519_key 構造体を与えられた場合、鍵のサイズを返します。
- 0 key が NULL の場合に返されます。

Example

```
int keySz;

curve25519_key key;
// 鍵を初期化して作成

keySz = wc_curve25519_size(&key);
```

C.14.3 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);

```

C.15 dox_comments/header_files-ja/curve448.h

C.15.1 Functions

	Name
int	wc_curve448_make_key (WC_RNG *rng, int keysize, curve448_key *key) この関数は、与えられた乱数生成器 rng を使用して、与えられたサイズ (keysize) の Curve448 鍵を生成し、与えられた curve448_key 構造体に格納します。wc_curve448_init() を通じて鍵構造体が初期化された後に呼び出す必要があります。

	Name
int	wc_curve448_shared_secret (curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen) この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンのみをサポートします。
int	wc_curve448_shared_secret_ex (curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen, int endian) この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンとリトルエンディアンの両方をサポートします。
int	wc_curve448_init (curve448_key * key) この関数は Curve448 鍵を初期化します。構造体の鍵を生成する前に呼び出す必要があります。
void	wc_curve448_free (curve448_key * key) この関数は Curve448 オブジェクトを解放します。
int	wc_curve448_import_private (const byte * priv, word32 privSz, curve448_key * key) この関数は curve448 秘密鍵のみをインポートします。(ビッグエンディアン)
int	wc_curve448_import_private_ex (const byte * priv, word32 privSz, curve448_key * key, int endian) curve448 秘密鍵のみのインポート。(ビッグエンディアンまたはリトルエンディアン)
int	wc_curve448_import_private_raw (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key) この関数は、公開鍵-秘密鍵ペアを curve448_key 構造体にインポートします。ビッグエンディアンのみ。
int	wc_curve448_import_private_raw_ex (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key, int endian) この関数は、公開鍵-秘密鍵ペアを curve448_key 構造体にインポートします。ビッグエンディアンとリトルエンディアンの両方をサポートします。
int	wc_curve448_export_private_raw (curve448_key * key, byte * out, word32 * outLen) この関数は、curve448_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンのみ。
int	wc_curve448_export_private_raw_ex (curve448_key * key, byte * out, word32 * outLen, int endian) この関数は、curve448_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンまたはリトルエンディアンを指定できます。

	Name
int	wc_curve448_import_public (const byte * in, word32 inLen, curve448_key * key) この関数は、与えられた in バッファから公開鍵をインポートし、curve448_key 構造体に格納します。
int	wc_curve448_import_public_ex (const byte * in, word32 inLen, curve448_key * key, int endian) この関数は、与えられた in バッファから公開鍵をインポートし、curve448_key 構造体に格納します。
int	wc_curve448_check_public (const byte * pub, word32 pubSz, int endian) この関数は、エンディアン順序を考慮して、公開鍵バッファが有効な Curve448 鍵値を保持しているかどうかをチェックします。
int	wc_curve448_export_public (curve448_key * key, byte * out, word32 * outLen) この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。
int	wc_curve448_export_public_ex (curve448_key * key, byte * out, word32 * outLen, int endian) この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンとリトルエンディアンの両方をサポートします。
int	wc_curve448_export_key_raw (curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、与えられた鍵構造体から鍵ペアをエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。
int	wc_curve448_export_key_raw_ex (curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian) curve448 鍵ペアをエクスポートします。ビッグエンディアンまたはリトルエンディアン。
int	wc_curve448_size (curve448_key * key) この関数は、与えられた鍵構造体の鍵サイズを返します。

C.15.2 Functions Documentation

C.15.2.1 function wc_curve448_make_key

```
int wc_curve448_make_key(
    WC_RNG * rng,
    int keysize,
    curve448_key * key
)
```

この関数は、与えられた乱数生成器 rng を使用して、与えられたサイズ (keysizes) の Curve448 鍵を生成し、与えられた curve448_key 構造体に格納します。wc_curve448_init() を通じて鍵構造体が初期化された後に呼び出す必要があります。

Parameters:

- **rng** ecc 鍵を生成するために使用される RNG オブジェクトへのポインタ。
- **keysize** 生成する鍵のサイズ。curve448 では 56 バイトである必要があります。
- **key** 生成された鍵を格納する curve448_key 構造体へのポインタ。

See: [wc_curve448_init](#)

Return:

- 0 鍵の生成に成功し、与えられた curve448_key 構造体に格納された場合に返されます。
- ECC_BAD_ARG_E 入力 keysize が curve448 鍵の keysize に対応していない場合（56 バイト）に返されます。
- RNG_FAILURE_E rng 内部ステータスが DRBG_OK でない場合、または rng で次のランダムブロックを生成する際にエラーがある場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

curve448_key key;
wc_curve448_init(&key); // 鍵を初期化
WC_RNG rng;
wc_InitRng(&rng); // 乱数生成器を初期化

ret = wc_curve448_make_key(&rng, 56, &key);
if (ret != 0) {
    // Curve448 鍵作成エラー
}
```

C.15.2.2 function wc_curve448_shared_secret

```
int wc_curve448_shared_secret(
    curve448_key * private_key,
    curve448_key * public_key,
    byte * out,
    word32 * outlen
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンのみをサポートします。

Parameters:

- **private_key** ユーザーの秘密鍵で初期化された curve448_key 構造体へのポインタ。
- **public_key** 受信した公開鍵を含む curve448_key 構造体へのポインタ。
- **out** 56 バイトの計算された秘密鍵を格納するバッファへのポインタ。
- **outlen** 出力バッファに書き込まれた長さを格納するポインタ。

See:

- [wc_curve448_init](#)
- [wc_curve448_make_key](#)
- [wc_curve448_shared_secret_ex](#)

Return:

- 0 共有秘密鍵の計算に成功した場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

Example

```

int ret;

byte sharedKey[56];
word32 keySz;
curve448_key privKey, pubKey;
// 両方の鍵を初期化

ret = wc_curve448_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
if (ret != 0) {
    // 共有鍵生成エラー
}

```

C.15.2.3 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
)

```

この関数は、秘密の秘密鍵と受信した公開鍵を与えられた共有秘密鍵を計算します。生成された秘密鍵をバッファ out に格納し、秘密鍵の変数を outlen に割り当てます。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **private_key** ユーザーの秘密鍵で初期化された curve448_key 構造体へのポインタ。
- **public_key** 受信した公開鍵を含む curve448_key 構造体へのポインタ。
- **out** 56 バイトの計算された秘密鍵を格納するバッファへのポインタ。
- **outlen** 出力バッファに書き込まれた長さを格納するポインタ。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- [wc_curve448_init](#)
- [wc_curve448_make_key](#)
- [wc_curve448_shared_secret](#)

Return:

- 0 共有秘密鍵の計算に成功した場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

Example

```

int ret;

byte sharedKey[56];
word32 keySz;

curve448_key privKey, pubKey;
// 両方の鍵を初期化

ret = wc_curve448_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
    EC448_BIG_ENDIAN);

```

```
if (ret != 0) {  
    // 共有鍵生成エラー  
}
```

C.15.2.4 function wc_curve448_init

```
int wc_curve448_init(  
    curve448_key * key  
)
```

この関数は Curve448 鍵を初期化します。構造体の鍵を生成する前に呼び出す必要があります。

Parameters:

- **key** 初期化する curve448_key 構造体へのポインタ。

See: [wc_curve448_make_key](#)

Return:

- 0 curve448_key 構造体の初期化に成功した場合に返されます。
- BAD_FUNC_ARG key が NULL の場合に返されます。

Example

```
curve448_key key;  
wc_curve448_init(&key); // 鍵を初期化  
// 鍵を作成し、暗号化に進む
```

C.15.2.5 function wc_curve448_free

```
void wc_curve448_free(  
    curve448_key * key  
)
```

この関数は Curve448 オブジェクトを解放します。

Parameters:

- **key** 解放する鍵オブジェクトへのポインタ。

See:

- [wc_curve448_init](#)
- [wc_curve448_make_key](#)

Example

```
curve448_key privKey;  
// 鍵を初期化し、共有秘密鍵の生成に使用  
wc_curve448_free(&privKey);
```

C.15.2.6 function wc_curve448_import_private

```
int wc_curve448_import_private(  
    const byte * priv,  
    word32 privSz,  
    curve448_key * key  
)
```

この関数は curve448 秘密鍵のみをインポートします。(ビッグエンディアン)

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。

See:

- [wc_curve448_import_private_ex](#)
- [wc_curve448_size](#)

Return:

- 0 秘密鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG key または priv が null の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE448_KEY_SIZE と等しくない場合に返されます。

Example

```
int ret;

byte priv[] = { // 秘密鍵の内容 };
curve448_key key;
wc_curve448_init(&key);

ret = wc_curve448_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

C.15.2.7 function wc_curve448_import_private_ex

```
int wc_curve448_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve448_key * key,
    int endian
)
```

curve448 秘密鍵のみのインポート。(ビッグエンディアンまたはリトルエンディアン)

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- [wc_curve448_import_private](#)
- [wc_curve448_size](#)

Return:

- 0 秘密鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG key または priv が null の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE448_KEY_SIZE と等しくない場合に返されます。

Example

```
int ret;

byte priv[] = { // 秘密鍵の内容 };
```

```

curve448_key key;
wc_curve448_init(&key);

ret = wc_curve448_import_private_ex(priv, sizeof(priv), &key,
    EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}

```

C.15.2.8 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
)

```

この関数は、公開鍵-秘密鍵ペアを curve448_key 構造体にインポートします。ビッグエンディアンのみ。

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **pub** インポートする公開鍵を含むバッファへのポインタ。
- **pubSz** インポートする公開鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。

See:

- [wc_curve448_init](#)
- [wc_curve448_make_key](#)
- [wc_curve448_import_public](#)
- [wc_curve448_export_private_raw](#)

Return:

- 0 curve448_key 構造体へのインポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが null の場合に返されます。
- ECC_BAD_ARG_E 入力鍵の鍵サイズが公開鍵または秘密鍵のサイズと一致しない場合に返されます。

Example

```

int ret;

byte priv[56];
byte pub[56];
// 公開鍵と秘密鍵で初期化
curve448_key key;

wc_curve448_init(&key);
// 鍵を初期化

ret = wc_curve448_import_private_raw(&priv, sizeof(priv), pub, sizeof(pub),
    &key);
if (ret != 0) {
    // 鍵のインポートエラー
}

```

C.15.2.9 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
)
```

この関数は、公開鍵-秘密鍵ペアを curve448_key 構造体にインポートします。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **priv** インポートする秘密鍵を含むバッファへのポインタ。
- **privSz** インポートする秘密鍵の長さ。
- **pub** インポートする公開鍵を含むバッファへのポインタ。
- **pubSz** インポートする公開鍵の長さ。
- **key** インポートされた鍵を格納する構造体へのポインタ。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- [wc_curve448_init](#)
- [wc_curve448_make_key](#)
- [wc_curve448_import_public](#)
- [wc_curve448_export_private_raw](#)
- [wc_curve448_import_private_raw](#)

Return:

- 0 curve448_key 構造体へのインポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが null の場合に返されます。
- ECC_BAD_ARG_E 入力鍵の鍵サイズが公開鍵または秘密鍵のサイズと一致しない場合に返されます。

Example

```
int ret;

byte priv[56];
byte pub[56];
// 公開鍵と秘密鍵で初期化
curve448_key key;

wc_curve448_init(&key);
// 鍵を初期化

ret = wc_curve448_import_private_raw_ex(&priv, sizeof(priv), pub,
    sizeof(pub), &key, EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

C.15.2.10 function wc_curve448_export_private_raw

```
int wc_curve448_export_private_raw(
    curve448_key * key,
```

```

    byte * out,
    word32 * outLen
)

```

この関数は、curve448_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンのみ。

Parameters:

- **key** 鍵をエクスポートする構造体へのポインタ。
- **out** エクスポートされた鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。

See:

- [wc_curve448_init](#)
- [wc_curve448_make_key](#)
- [wc_curve448_import_private_raw](#)
- [wc_curve448_export_private_raw_ex](#)

Return:

- 0 curve448_key 構造体から秘密鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E [wc_curve448_size\(\)](#) が key と等しくない場合に返されます。

Example

```

int ret;
byte priv[56];
int privSz;

curve448_key key;
// 鍵を初期化して作成

ret = wc_curve448_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}

```

C.15.2.11 function wc_curve448_export_private_raw_ex

```

int wc_curve448_export_private_raw_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)

```

この関数は、curve448_key 構造体から秘密鍵をエクスポートし、与えられた out バッファに格納します。また、outLen をエクスポートされた鍵のサイズに設定します。ビッグエンディアンまたはリトルエンディアンを指定できます。

Parameters:

- **key** 鍵をエクスポートする構造体へのポインタ。
- **out** エクスポートされた鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。

- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- `wc_curve448_init`
- `wc_curve448_make_key`
- `wc_curve448_import_private_raw`
- `wc_curve448_export_private_raw`
- `wc_curve448_size`

Return:

- 0 `curve448_key` 構造体から秘密鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E `wc_curve448_size()` が `key` と等しくない場合に返されます。

Example

```
int ret;

byte priv[56];
int privSz;
curve448_key key;
// 鍵を初期化して作成
ret = wc_curve448_export_private_raw_ex(&key, priv, &privSz,
    EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

C.15.2.12 function `wc_curve448_import_public`

```
int wc_curve448_import_public(
    const byte * in,
    word32 inLen,
    curve448_key * key
)
```

この関数は、与えられた `in` バッファから公開鍵をインポートし、`curve448_key` 構造体に格納します。

Parameters:

- **in** インポートする公開鍵を含むバッファへのポインタ。
- **inLen** インポートする公開鍵の長さ。
- **key** 鍵を格納する `curve448_key` 構造体へのポインタ。

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 `curve448_key` 構造体への公開鍵のインポートに成功した場合に返されます。
- ECC_BAD_ARG_E `inLen` パラメータが鍵構造体の鍵サイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

byte pub[56];
// pub を公開鍵で初期化

curve448_key key;
// 鍵を初期化

ret = wc_curve448_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

C.15.2.13 function wc_curve448_import_public_ex

```
int wc_curve448_import_public_ex(
    const byte * in,
    word32 inLen,
    curve448_key * key,
    int endian
)
```

この関数は、与えられた in バッファから公開鍵をインポートし、curve448_key 構造体に格納します。

Parameters:

- **in** インポートする公開鍵を含むバッファへのポインタ。
- **inLen** インポートする公開鍵の長さ。
- **key** 鍵を格納する curve448_key 構造体へのポインタ。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

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 curve448_key 構造体への公開鍵のインポートに成功した場合に返されます。
- ECC_BAD_ARG_E inLen パラメータが鍵構造体の鍵サイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

byte pub[56];
// pub を公開鍵で初期化
curve448_key key;
// 鍵を初期化

ret = wc_curve448_import_public_ex(pub, sizeof(pub), &key,
    EC448_BIG_ENDIAN);
```



```
if (ret != 0) {  
    // 鍵のインポートエラー  
}
```

C.15.2.14 function wc_curve448_check_public

```
int wc_curve448_check_public(  
    const byte * pub,  
    word32 pubSz,  
    int endian  
)
```

この関数は、エンディアン順序を考慮して、公開鍵バッファが有効な Curve448 鍵値を保持しているかどうかをチェックします。

Parameters:

- **pub** チェックする公開鍵を含むバッファへのポインタ。
- **pubSz** チェックする公開鍵の長さ。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- [wc_curve448_init](#)
- [wc_curve448_import_public](#)
- [wc_curve448_import_public_ex](#)
- [wc_curve448_size](#)

Return:

- 0 公開鍵の値が有効な場合に返されます。
- ECC_BAD_ARG_E 公開鍵の値が有効でない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;  
  
byte pub[] = { // 公開鍵の内容 };  
  
ret = wc_curve448_check_public_ex(pub, sizeof(pub), EC448_BIG_ENDIAN);  
if (ret != 0) {  
    // 鍵のインポートエラー  
}
```

C.15.2.15 function wc_curve448_export_public

```
int wc_curve448_export_public(  
    curve448_key * key,  
    byte * out,  
    word32 * outLen  
)
```

この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。

Parameters:

- **key** 鍵をエクスポートする curve448_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。

- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。

See:

- [wc_curve448_init](#)
- [wc_curve448_export_private_raw](#)
- [wc_curve448_import_public](#)

Return:

- 0 curve448_key 構造体から公開鍵のエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E outLen が CURVE448_PUB_KEY_SIZE より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```
int ret;

byte pub[56];
int pubSz;

curve448_key key;
// 鍵を初期化して作成

ret = wc_curve448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

C.15.2.16 function wc_curve448_export_public_ex

```
int wc_curve448_export_public_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

この関数は、与えられた鍵構造体から公開鍵をエクスポートし、結果を out バッファに格納します。ビッグエンディアンとリトルエンディアンの両方をサポートします。

Parameters:

- **key** 鍵をエクスポートする curve448_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。
- **outLen** 入力時は、out のバイト単位のサイズ。出力時は、出力バッファに書き込まれたバイト数を格納します。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- [wc_curve448_init](#)
- [wc_curve448_export_private_raw](#)
- [wc_curve448_import_public](#)

Return:

- 0 curve448_key 構造体から公開鍵のエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E outLen が CURVE448_PUB_KEY_SIZE より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

Example

```

int ret;

byte pub[56];
int pubSz;

curve448_key key;
// 鍵を初期化して作成

ret = wc_curve448_export_public_ex(&key, pub, &pubSz, EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のエクスポートエラー
}

```

C.15.2.17 function wc_curve448_export_key_raw

```

int wc_curve448_export_key_raw(
    curve448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)

```

この関数は、与えられた鍵構造体から鍵ペアをエクスポートし、結果を out バッファに格納します。ビッグエンディアンのみ。

Parameters:

- **key** 鍵ペアをエクスポートする curve448_key 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** 入力時は、priv バッファのバイト単位のサイズ。出力時は、priv バッファに書き込まれたバイト数を格納します。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** 入力時は、pub バッファのバイト単位のサイズ。出力時は、pub バッファに書き込まれたバイト数を格納します。

See:

- [wc_curve448_export_key_raw_ex](#)
- [wc_curve448_export_private_raw](#)

Return:

- 0 curve448_key 構造体から鍵ペアのエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE448_KEY_SIZE より小さい、または pubSz が CURVE448_PUB_KEY_SIZE より小さい場合に返されます。

Example

```

int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

```

```

curve448_key key;
// 鍵を初期化して作成

ret = wc_curve448_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
    // 鍵のエクスポートエラー
}

```

C.15.2.18 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
)

```

curve448 鍵ペアをエクスポートします。ビッグエンディアンまたはリトルエンディアン。

Parameters:

- **key** 鍵ペアをエクスポートする curve448_key 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** 入力時は、priv バッファのバイト単位のサイズ。出力時は、priv バッファに書き込まれたバイト数を格納します。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** 入力時は、pub バッファのバイト単位のサイズ。出力時は、pub バッファに書き込まれたバイト数を格納します。
- **endian** 使用する形式を設定するための EC448_BIG_ENDIAN または EC448_LITTLE_ENDIAN。

See:

- wc_curve448_export_key_raw
- wc_curve448_export_private_raw_ex
- wc_curve448_export_public_ex

Return:

- 0 成功。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E privSz が CURVE448_KEY_SIZE より小さい、または pubSz が CURVE448_PUB_KEY_SIZE より小さい場合に返されます。

この関数は、与えられた鍵構造体から鍵ペアをエクスポートし、結果を out バッファに格納します。ビッグエンディアンまたはリトルエンディアン。

Example

```

int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

curve448_key key;
// 鍵を初期化して作成

```

```
ret = wc_curve448_export_key_raw_ex(&key, priv, &privSz, pub, &pubSz,
    EC448_BIG_ENDIAN);
if (ret != 0) {
    // 鍵のエクスポートエラー
}
```

C.15.2.19 function wc_curve448_size

```
int wc_curve448_size(
    curve448_key * key
)
```

この関数は、与えられた鍵構造体の鍵サイズを返します。

Parameters:

- **key** 鍵サイズを決定する curve448_key 構造体へのポインタ。

See:

- [wc_curve448_init](#)
- [wc_curve448_make_key](#)

Return:

- Success 有効で初期化された curve448_key 構造体を与えられた場合、鍵のサイズを返します。
- 0 key が NULL の場合に返されます。

Example

```
int keySz;

curve448_key key;
// 鍵を初期化して作成

keySz = wc_curve448_size(&key);
```

C.15.3 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);

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);

```

C.16 dox_comments/header_files-ja/des3.h

C.16.1 Functions

	Name
int	wc_Des_SetKey (Des * des, const byte * key, const byte * iv, int dir) この関数は、引数として与えられた Des 構造体の鍵と初期化ベクトル (iv) を設定します。また、暗号化と復号に必要なバッファがまだ初期化されていない場合、それらを初期化し、スペースを割り当てます。注意: iv が提供されない場合 (つまり iv == NULL)、初期化ベクトルはデフォルトで 0 の iv になります。

	Name
void	wc_Des_SetIV (Des * des, const byte * iv) この関数は、引数として与えられた Des 構造体の初期化ベクトル (iv) を設定します。NULL の iv が渡された場合、初期化ベクトルを 0 に設定します。
int	wc_Des_CbcEncrypt (Des * des, byte * out, const byte * in, word32 sz) この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。
int	wc_Des_CbcDecrypt (Des * des, byte * out, const byte * in, word32 sz) この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。
int	wc_Des_EcbEncrypt (Des * des, byte * out, const byte * in, word32 sz) この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。電子コードブック (ECB) モードの Des 暗号化を使用します。
int	wc_Des3_EcbEncrypt (Des3 * des, byte * out, const byte * in, word32 sz) この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。電子コードブック (ECB) モードの Des3 暗号化を使用します。警告: ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。
int	wc_Des3_SetKey (Des3 * des, const byte * key, const byte * iv, int dir) この関数は、引数として与えられた Des3 構造体の鍵と初期化ベクトル (iv) を設定します。また、暗号化と復号に必要なバッファがまだ初期化されていない場合、それらを初期化し、スペースを割り当てます。注意: iv が提供されない場合 (つまり iv == NULL)、初期化ベクトルはデフォルトで 0 の iv になります。
int	wc_Des3_SetIV (Des3 * des, const byte * iv) この関数は、引数として与えられた Des3 構造体の初期化ベクトル (iv) を設定します。NULL の iv が渡された場合、初期化ベクトルを 0 に設定します。
int	wc_Des3_CbcEncrypt (Des3 * des, byte * out, const byte * in, word32 sz) この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。
int	wc_Des3_CbcDecrypt (Des3 * des, byte * out, const byte * in, word32 sz) この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。

C.16.2 Functions Documentation

C.16.2.1 function wc_Des_SetKey

```
int wc_Des_SetKey(
    Des * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

この関数は、引数として与えられた Des 構造体の鍵と初期化ベクトル (iv) を設定します。また、暗号化と復号に必要なバッファがまだ初期化されていない場合、それらを初期化し、スペースを割り当てます。注意: iv が提供されない場合 (つまり iv == NULL)、初期化ベクトルはデフォルトで 0 の iv になります。

Parameters:

- **des** 初期化する Des 構造体へのポインタ
- **key** Des 構造体を初期化するための 8 バイトの鍵を含むバッファへのポインタ
- **iv** Des 構造体を初期化するための 8 バイトの iv を含むバッファへのポインタ。これが提供されない場合、iv はデフォルトで 0 になります
- **dir** 暗号化の方向。有効なオプションは: DES_ENCRYPTION と DES_DECRYPTION です

See:

- [wc_Des_SetIV](#)
- [wc_Des3_SetKey](#)

Return: 0 Des 構造体の鍵と初期化ベクトルの設定に成功した場合

3

Example

```
Des enc; // 暗号化に使用される Des 構造体
int ret;
byte key[] = { // 8 バイトの鍵で初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };

ret = wc_Des_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // des 構造体の初期化エラー
}
```

C.16.2.2 function wc_Des_SetIV

```
void wc_Des_SetIV(
    Des * des,
    const byte * iv
)
```

この関数は、引数として与えられた Des 構造体の初期化ベクトル (iv) を設定します。NULL の iv が渡された場合、初期化ベクトルを 0 に設定します。

Parameters:

- **des** iv を設定する Des 構造体へのポインタ
- **iv** Des 構造体を初期化するための 8 バイトの iv を含むバッファへのポインタ。これが提供されない場合、iv はデフォルトで 0 になります

See: [wc_Des_SetKey](#)

Return: none 戻り値なし。

3

Example

```
Des enc; // 暗号化に使用される Des 構造体
// wc_Des_SetKey で enc を初期化
byte iv[] = { // 8 バイトの iv で初期化 };
wc_Des_SetIV(&enc, iv);
}
```

C.16.2.3 function wc_Des_CbcEncrypt

```
int wc_Des_CbcEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。

Parameters:

- **des** 暗号化に使用する Des 構造体へのポインタ
- **out** 暗号化された暗号文を格納するバッファへのポインタ
- **in** 暗号化するメッセージを含む入力バッファへのポインタ
- **sz** 暗号化するメッセージの長さ

See:

- [wc_Des_SetKey](#)
- [wc_Des_CbcDecrypt](#)

Return: 0 指定された入力メッセージの暗号化に成功した場合に返されます

3

Example

```
Des enc; // 暗号化に使用される Des 構造体
// wc_Des_SetKey で enc を初期化、モード DES_ENCRYPTION を使用

byte plain[] = { // メッセージで初期化 };
byte cipher[sizeof(plain)];

if ( wc_Des_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0 ) {
    // メッセージの暗号化エラー
}
```

C.16.2.4 function wc_Des_CbcDecrypt

```
int wc_Des_CbcDecrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。

Parameters:

- **des** 復号に使用する Des 構造体へのポインタ
- **out** 復号された平文を格納するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号する暗号文の長さ

See:

- `wc_Des_SetKey`
- `wc_Des_CbcEncrypt`

Return: 0 指定された暗号文の復号に成功した場合に返されます

3

Example

```
Des dec; // 復号に使用される Des 構造体
// wc_Des_SetKey で dec を初期化、モード DES_DECRYPTION を使用

byte cipher[] = { // 暗号文で初期化 };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
    // メッセージの復号エラー
}
```

C.16.2.5 function wc_Des_EcbEncrypt

```
int wc_Des_EcbEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。電子コードブック (ECB) モードの Des 暗号化を使用します。

Parameters:

- **des** 暗号化に使用する Des 構造体へのポインタ
- **out** 暗号化されたメッセージを格納するバッファへのポインタ
- **in** 暗号化する平文を含む入力バッファへのポインタ
- **sz** 暗号化する平文の長さ

See: `wc_Des_SetKe`

Return: 0: 指定された平文の暗号化に成功した場合に返されます。

3

Example

```
Des enc; // 暗号化に使用される Des 構造体
// wc_Des_SetKey で enc を初期化、モード DES_ENCRYPTION を使用

byte plain[] = { // 暗号化するメッセージで初期化 };
```

```

byte cipher[sizeof(plain)];

if ( wc_Des3_EcbEncrypt(&enc,cipher, plain, sizeof(plain)) != 0) {
    // メッセージの暗号化エラー
}

```

C.16.2.6 function wc_Des3_EcbEncrypt

```

int wc_Des3_EcbEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)

```

この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。電子コードブック (ECB) モードの Des3 暗号化を使用します。警告: ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

Parameters:

- **des3** 暗号化に使用する Des3 構造体へのポインタ
- **out** 暗号化されたメッセージを格納するバッファへのポインタ
- **in** 暗号化する平文を含む入力バッファへのポインタ
- **sz** 暗号化する平文の長さ

See: [wc_Des3_SetKey](#)

Return: 0 指定された平文の暗号化に成功した場合に返されます

3

Example

```

Des3 enc; // 暗号化に使用される Des3 構造体
// wc_Des3_SetKey で enc を初期化、モード DES_ENCRYPTION を使用

byte plain[] = { // 暗号化するメッセージで初期化 };
byte cipher[sizeof(plain)];

if ( wc_Des3_EcbEncrypt(&enc,cipher, plain, sizeof(plain)) != 0) {
    // メッセージの暗号化エラー
}

```

C.16.2.7 function wc_Des3_SetKey

```

int wc_Des3_SetKey(
    Des3 * des,
    const byte * key,
    const byte * iv,
    int dir
)

```

この関数は、引数として与えられた Des3 構造体の鍵と初期化ベクトル (iv) を設定します。また、暗号化と復号に必要なバッファがまだ初期化されていない場合、それらを初期化し、スペースを割り当てます。注意: iv が提供されない場合 (つまり iv == NULL)、初期化ベクトルはデフォルトで 0 の iv になります。

Parameters:

- **des3** 初期化する Des3 構造体へのポインタ

- **key** Des3 構造体を初期化するための 24 バイトの鍵を含むバッファへのポインタ
- **iv** Des3 構造体を初期化するための 8 バイトの iv を含むバッファへのポインタ。これが提供されない場合、iv はデフォルトで 0 になります
- **dir** 暗号化の方向。有効なオプションは: DES_ENCRYPTION と DES_DECRYPTION です

See:

- [wc_Des3_SetIV](#)
- [wc_Des3_CbcEncrypt](#)
- [wc_Des3_CbcDecrypt](#)

Return: 0 Des 構造体の鍵と初期化ベクトルの設定に成功した場合

3

Example

```
Des3 enc; // 暗号化に使用される Des3 構造体
int ret;
byte key[] = { // 24 バイトの鍵で初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };

ret = wc_Des3_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // des 構造体の初期化エラー
}
```

C.16.2.8 function wc_Des3_SetIV

```
int wc_Des3_SetIV(
    Des3 * des,
    const byte * iv
)
```

この関数は、引数として与えられた Des3 構造体の初期化ベクトル (iv) を設定します。NULL の iv が渡された場合、初期化ベクトルを 0 に設定します。

Parameters:

- **des** iv を設定する Des3 構造体へのポインタ
- **iv** Des3 構造体を初期化するための 8 バイトの iv を含むバッファへのポインタ。これが提供されない場合、iv はデフォルトで 0 になります

See: [wc_Des3_SetKey](#)

Return: none 戻り値なし。

3

Example

```
Des3 enc; // 暗号化に使用される Des3 構造体
// wc_Des3_SetKey で enc を初期化

byte iv[] = { // 8 バイトの iv で初期化 };

wc_Des3_SetIV(&enc, iv);
}
```

C.16.2.9 function wc_Des3_CbcEncrypt

```
int wc_Des3_CbcEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、入力メッセージ in を暗号化し、結果を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。

Parameters:

- **des** 暗号化に使用する Des3 構造体へのポインタ
- **out** 暗号化された暗号文を格納するバッファへのポインタ
- **in** 暗号化するメッセージを含む入力バッファへのポインタ
- **sz** 暗号化するメッセージの長さ

See:

- [wc_Des3_SetKey](#)
- [wc_Des3_CbcDecrypt](#)

Return: 0 指定された入力メッセージの暗号化に成功した場合に返されます

3

Example

```
Des3 enc; // 暗号化に使用される Des3 構造体
// wc_Des3_SetKey で enc を初期化、モード DES_ENCRYPTION を使用

byte plain[] = { // メッセージで初期化 };
byte cipher[sizeof(plain)];

if ( wc_Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0 ) {
    // メッセージの暗号化エラー
}
```

C.16.2.10 function wc_Des3_CbcDecrypt

```
int wc_Des3_CbcDecrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。

Parameters:

- **des** 復号に使用する Des3 構造体へのポインタ
- **out** 復号された平文を格納するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号する暗号文の長さ

See:

- `wc_Des3_SetKey`
- `wc_Des3_CbcEncrypt`

Return: 0 指定された暗号文の復号に成功した場合に返されます

3

Example

```
Des3 dec; // 復号に使用される Des 構造体
// wc_Des3_SetKey で dec を初期化、モード DES_DECRYPTION を使用

byte cipher[] = { // 暗号文で初期化 };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0 ) {
    // メッセージの復号エラー
}
```

C.16.3 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);
```

C.17 dox_comments/header_files-ja/dh.h

C.17.1 Functions

	Name
int	wc_InitDhKey (DhKey * key) この関数は、Diffie-Hellman 鍵交換プロトコルで安全な秘密鍵をネゴシエートするために使用する Diffie-Hellman 鍵を初期化します。
int	wc_FreeDhKey (DhKey * key) この関数は、Diffie-Hellman 鍵交換プロトコルで安全な秘密鍵をネゴシエートするために使用された後、Diffie-Hellman 鍵を解放します。
int	wc_DhGenerateKeyPair (DhKey * key, WC_RNG * rng, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、Diffie-Hellman 公開パラメータに基づいて公開/秘密鍵ペアを生成し、秘密鍵を priv に、公開鍵を pub に格納します。初期化された Diffie-Hellman 鍵と初期化された rng 構造体を受け取ります。
int	wc_DhAgree (DhKey * key, byte * agree, word32 * agreeSz, const byte * priv, word32 * privSz, const byte * otherPub, word32 * pubSz) この関数は、ローカル秘密鍵と受信した公開鍵に基づいて合意された秘密鍵を生成します。交換の両側で完了すると、この関数は対称通信用の合意された秘密鍵を生成します。共有秘密鍵の生成に成功すると、書き込まれた秘密鍵のサイズが agreeSz に格納されます。
int	wc_DhKeyDecode (const byte * input, word32 * inOutIdx, DhKey * key, word32 inSz) この関数は、DER フォーマットの鍵を含む指定された入力バッファから Diffie-Hellman 鍵をデコードします。結果を DhKey 構造体に格納します。
int	wc_DhSetKey (DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz) この関数は、入力秘密鍵パラメータを使用して DhKey 構造体の鍵を設定します。wc_DhKeyDecode とは異なり、この関数は入力鍵が DER フォーマットでフォーマットされている必要はなく、代わりに単純に解析された入力パラメータ p(素数) と g(ベース) を受け入れます。
int	wc_DhParamsLoad (const byte * input, word32 inSz, byte * p, word32 * pInOutSz, byte * g, word32 * gInOutSz) この関数は、指定された入力バッファから DER フォーマットの Diffie-Hellman パラメータ p(素数) と g(ベース) をロードします。
const DhParams *	wc_Dh_ffdhe2048_Get (void) この関数は...を返し、HAVE_FFDHE_2048 が定義されている必要があります。
const DhParams *	wc_Dh_ffdhe3072_Get (void) この関数は...を返し、HAVE_FFDHE_3072 が定義されている必要があります。
const DhParams *	wc_Dh_ffdhe4096_Get (void) この関数は...を返し、HAVE_FFDHE_4096 が定義されている必要があります。

	Name
const DhParams *	wc_Dh_ffdhe6144_Get (void) この関数は...を返し、HAVE_FFDHE_6144 が定義されている必要があります。
const DhParams *	wc_Dh_ffdhe8192_Get (void) この関数は...を返し、HAVE_FFDHE_8192 が定義されている必要があります。
int	wc_DhCheckKeyPair (DhKey * key, const byte * pub, word32 pubSz, const byte * priv, word32 privSz) FFC の SP 800_56Ar3、セクション 5.6.2.1.4、メソッド (b) のプロセスに従って、DH 鍵のペアワイズ整合性をチェックします。
int	wc_DhCheckPrivKey (DhKey * key, const byte * priv, word32 pubSz) 無効な数値について DH 秘密鍵をチェックします
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)

C.17.2 Functions Documentation

C.17.2.1 function wc_InitDhKey

```
int wc_InitDhKey(
    DhKey * key
)
```

この関数は、Diffie-Hellman 鍵交換プロトコルで安全な秘密鍵をネゴシエートするために使用する Diffie-Hellman 鍵を初期化します。

Parameters:

- **key** 安全な鍵交換で使用するために初期化する DhKey 構造体へのポインタ

See:

- **wc_FreeDhKey**
- **wc_DhGenerateKeyPair**

Return: none 戻り値なし。

Example

```
DhKey key;  
wc_InitDhKey(&key); // DH 鍵を初期化
```

C.17.2.2 function wc_FreeDhKey

```
int wc_FreeDhKey(  
    DhKey * key  
)
```

この関数は、Diffie-Hellman 鍵交換プロトコルで安全な秘密鍵をネゴシエートするために使用された後、Diffie-Hellman 鍵を解放します。

Parameters:

- **key** 解放する DhKey 構造体へのポインタ

See: [wc_InitDhKey](#)

Return: none 戻り値なし。

Example

```
DhKey key;  
// 鍵を初期化、鍵交換を実行
```

```
wc_FreeDhKey(&key); // メモリリークを避けるために DH 鍵を解放
```

C.17.2.3 function wc_DhGenerateKeyPair

```
int wc_DhGenerateKeyPair(  
    DhKey * key,  
    WC_RNG * rng,  
    byte * priv,  
    word32 * privSz,  
    byte * pub,  
    word32 * pubSz  
)
```

この関数は、Diffie-Hellman 公開パラメータに基づいて公開/秘密鍵ペアを生成し、秘密鍵を priv に、公開鍵を pub に格納します。初期化された Diffie-Hellman 鍵と初期化された rng 構造体を受け取ります。

Parameters:

- **key** 鍵ペアを生成する DhKey 構造体へのポインタ
- **rng** 鍵を生成するために使用する初期化された乱数生成器 (rng) へのポインタ
- **priv** 秘密鍵を格納するバッファへのポインタ
- **privSz** priv に書き込まれた秘密鍵のサイズを格納します
- **pub** 公開鍵を格納するバッファへのポインタ
- **pubSz** pub に書き込まれた秘密鍵のサイズを格納します

See:

- [wc_InitDhKey](#)
- [wc_DhSetKey](#)
- [wc_DhKeyDecode](#)

Return:

- BAD_FUNC_ARG この関数への入力の 1 つを解析する際にエラーがある場合に返されます
- RNG_FAILURE_E rng を使用して乱数を生成する際にエラーがある場合に返されます
- MP_INIT_E 公開鍵を生成する際に数学ライブラリでエラーがある場合に返される可能性があります
- MP_READ_E 公開鍵を生成する際に数学ライブラリでエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 公開鍵を生成する際に数学ライブラリでエラーがある場合に返される可能性があります
- MP_TO_E 公開鍵を生成する際に数学ライブラリでエラーがある場合に返される可能性があります

Example

```
DhKey key;
int ret;
byte priv[256];
byte pub[256];
word32 privSz, pubSz;

wc_InitDhKey(&key); // 鍵を初期化
// wc_DhSetKey または wc_DhKeyDecode を使用して DH パラメータを設定
WC_RNG rng;
wc_InitRng(&rng); // rng を初期化
ret = wc_DhGenerateKeyPair(&key, &rng, priv, &privSz, pub, &pubSz);
```

C.17.2.4 function wc_DhAgree

```
int wc_DhAgree(
    DhKey * key,
    byte * agree,
    word32 * agreeSz,
    const byte * priv,
    word32 privSz,
    const byte * otherPub,
    word32 pubSz
)
```

この関数は、ローカル秘密鍵と受信した公開鍵に基づいて合意された秘密鍵を生成します。交換の両側で完了すると、この関数は対称通信用の合意された秘密鍵を生成します。共有秘密鍵の生成に成功すると、書き込まれた秘密鍵のサイズが agreeSz に格納されます。

Parameters:

- **key** 共有鍵を計算するために使用する DhKey 構造体へのポインタ
- **agree** 秘密鍵を格納するバッファへのポインタ
- **agreeSz** 生成成功後に秘密鍵のサイズを保持します
- **priv** ローカル秘密鍵を含むバッファへのポインタ
- **privSz** ローカル秘密鍵のサイズ
- **otherPub** 受信した公開鍵を含むバッファへのポインタ
- **pubSz** 受信した公開鍵のサイズ

See: [wc_DhGenerateKeyPair](#)

Return:

- 0 合意された秘密鍵の生成に成功した場合に返されます
- MP_INIT_E 共有秘密鍵の生成中にエラーがある場合に返される可能性があります
- MP_READ_E 共有秘密鍵の生成中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 共有秘密鍵の生成中にエラーがある場合に返される可能性があります
- MP_TO_E 共有秘密鍵の生成中にエラーがある場合に返される可能性があります

Example

```

DhKey key;
int ret;
byte priv[256];
byte agree[256];
word32 agreeSz;

// 鍵を初期化、鍵素数とベースを設定
// wc_DhGenerateKeyPair -- 秘密鍵を priv に格納
byte pub[] = { // 受信した公開鍵で初期化 };
ret = wc_DhAgree(&key, agree, &agreeSz, priv, sizeof(priv), pub,
sizeof(pub));
if ( ret != 0 ) {
    // 共有鍵生成エラー
}

```

C.17.2.5 function wc_DhKeyDecode

```

int wc_DhKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DhKey * key,
    word32 inSz
)

```

この関数は、DER フォーマットの鍵を含む指定された入力バッファから Diffie-Hellman 鍵をデコードします。結果を DhKey 構造体に格納します。

Parameters:

- **input** DER フォーマットの Diffie-Hellman 鍵を含むバッファへのポインタ
- **inOutIdx** 鍵をデコードする際に解析されたインデックスを格納する整数へのポインタ
- **key** 入力鍵で初期化する DhKey 構造体へのポインタ
- **inSz** 入力バッファの長さ。読み取り可能な最大長を示します

See: [wc_DhSetKey](#)

Return:

- 0 入力鍵のデコードに成功した場合に返されます
- ASN_PARSE_E 入力のシーケンスを解析する際にエラーがある場合に返されます
- ASN_DH_KEY_E 解析された入力から秘密鍵パラメータを読み取る際にエラーがある場合に返されます

Example

```

DhKey key;
word32 idx = 0;

byte keyBuff[1024];
// DER フォーマットの鍵で初期化
wc_DhKeyInit(&key);
ret = wc_DhKeyDecode(keyBuff, &idx, &key, sizeof(keyBuff));

if ( ret != 0 ) {
    // 鍵のデコードエラー
}

```

C.17.2.6 function wc_DhSetKey

```
int wc_DhSetKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz
)
```

この関数は、入力秘密鍵パラメータを使用して DhKey 構造体の鍵を設定します。wc_DhKeyDecode とは異なり、この関数は入力鍵が DER フォーマットでフォーマットされている必要はなく、代わりに単純に解析された入力パラメータ p(素数) と g(ベース) を受け入れます。

Parameters:

- **key** 鍵を設定する DhKey 構造体へのポインタ
- **p** 鍵で使用する素数を含むバッファへのポインタ
- **pSz** 入力素数の長さ
- **g** 鍵で使用するベースを含むバッファへのポインタ
- **gSz** 入力ベースの長さ

See: [wc_DhKeyDecode](#)

Return:

- 0 鍵の設定に成功した場合に返されます
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価される場合に返されます
- MP_INIT_E 格納用の鍵パラメータを初期化する際にエラーがある場合に返されます
- ASN_DH_KEY_E DH 鍵パラメータ p と g を読み取る際にエラーがある場合に返されます

Example

```
DhKey key;
```

```
byte p[] = { // 素数で初期化 };
byte g[] = { // ベースで初期化 };
wc_DhKeyInit(&key);
ret = wc_DhSetKey(key, p, sizeof(p), g, sizeof(g));

if ( ret != 0 ) {
    // 鍵設定エラー
}
```

C.17.2.7 function wc_DhParamsLoad

```
int wc_DhParamsLoad(
    const byte * input,
    word32 inSz,
    byte * p,
    word32 * pInOutSz,
    byte * g,
    word32 * gInOutSz
)
```

この関数は、指定された入力バッファから DER フォーマットの Diffie-Hellman パラメータ p(素数) と g(ベース) をロードします。

Parameters:

- **input** 解析する DER フォーマットの Diffie-Hellman 証明書を含むバッファへのポインタ
- **inSz** 入力バッファのサイズ
- **p** 解析された素数を格納するバッファへのポインタ
- **pInOutSz** p バッファで利用可能なサイズを含む word32 オブジェクトへのポインタ。関数呼び出しの完了後、バッファに書き込まれたバイト数で上書きされます
- **g** 解析されたベースを格納するバッファへのポインタ
- **gInOutSz** g バッファで利用可能なサイズを含む word32 オブジェクトへのポインタ。関数呼び出しの完了後、バッファに書き込まれたバイト数で上書きされます

See:

- [wc_DhSetKey](#)
- [wc_DhKeyDecode](#)

Return:

- 0 DH パラメータの抽出に成功した場合に返されます
- ASN_PARSE_E DER フォーマットの DH 証明書を解析する際にエラーが発生した場合に返されます
- BUFFER_E p または g に解析されたパラメータを格納する十分なスペースがない場合に返されます

Example

```
byte dhCert[] = { DER フォーマットの証明書で初期化 };
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 ) {
    // 入力の解析エラー
}
```

C.17.2.8 function wc_Dh_ffdhe2048_Get

```
const DhParams * wc_Dh_ffdhe2048_Get(
    void
)
```

この関数は...を返し、HAVE_FFDHE_2048 が定義されている必要があります。

See:

- [wc_Dh_ffdhe3072_Get](#)
- [wc_Dh_ffdhe4096_Get](#)
- [wc_Dh_ffdhe6144_Get](#)
- [wc_Dh_ffdhe8192_Get](#)

C.17.2.9 function wc_Dh_ffdhe3072_Get

```
const DhParams * wc_Dh_ffdhe3072_Get(
    void
)
```

この関数は...を返し、HAVE_FFDHE_3072 が定義されている必要があります。

See:

- [wc_Dh_ffdhe2048_Get](#)
- [wc_Dh_ffdhe4096_Get](#)

- [wc_Dh_ffdhe6144_Get](#)
- [wc_Dh_ffdhe8192_Get](#)

C.17.2.10 function `wc_Dh_ffdhe4096_Get`

```
const DhParams * wc_Dh_ffdhe4096_Get(  
    void  
)
```

この関数は...を返し、HAVE_FFDHE_4096 が定義されている必要があります。

See:

- [wc_Dh_ffdhe2048_Get](#)
- [wc_Dh_ffdhe3072_Get](#)
- [wc_Dh_ffdhe6144_Get](#)
- [wc_Dh_ffdhe8192_Get](#)

C.17.2.11 function `wc_Dh_ffdhe6144_Get`

```
const DhParams * wc_Dh_ffdhe6144_Get(  
    void  
)
```

この関数は...を返し、HAVE_FFDHE_6144 が定義されている必要があります。

See:

- [wc_Dh_ffdhe2048_Get](#)
- [wc_Dh_ffdhe3072_Get](#)
- [wc_Dh_ffdhe4096_Get](#)
- [wc_Dh_ffdhe8192_Get](#)

C.17.2.12 function `wc_Dh_ffdhe8192_Get`

```
const DhParams * wc_Dh_ffdhe8192_Get(  
    void  
)
```

この関数は...を返し、HAVE_FFDHE_8192 が定義されている必要があります。

See:

- [wc_Dh_ffdhe2048_Get](#)
- [wc_Dh_ffdhe3072_Get](#)
- [wc_Dh_ffdhe4096_Get](#)
- [wc_Dh_ffdhe6144_Get](#)

C.17.2.13 function `wc_DhCheckKeyPair`

```
int wc_DhCheckKeyPair(  
    DhKey * key,  
    const byte * pub,  
    word32 pubSz,  
    const byte * priv,  
    word32 privSz  
)
```

FFC の SP 800-56Ar3、セクション 5.6.2.1.4、メソッド (b) のプロセスに従って、DH 鍵のペアワイズ整合性をチェックします。

C.17.2.14 function wc_DhCheckPrivKey

```
int wc_DhCheckPrivKey(  
    DhKey * key,  
    const byte * priv,  
    word32 pubSz  
)
```

無効な数値について DH 秘密鍵をチェックします

C.17.2.15 function wc_DhCheckPrivKey_ex

```
int wc_DhCheckPrivKey_ex(  
    DhKey * key,  
    const byte * priv,  
    word32 pubSz,  
    const byte * prime,  
    word32 primeSz  
)
```

C.17.2.16 function wc_DhCheckPubKey

```
int wc_DhCheckPubKey(  
    DhKey * key,  
    const byte * pub,  
    word32 pubSz  
)
```

C.17.2.17 function wc_DhCheckPubKey_ex

```
int wc_DhCheckPubKey_ex(  
    DhKey * key,  
    const byte * pub,  
    word32 pubSz,  
    const byte * prime,  
    word32 primeSz  
)
```

C.17.2.18 function wc_DhExportParamsRaw

```
int wc_DhExportParamsRaw(  
    DhKey * dh,  
    byte * p,  
    word32 * pSz,  
    byte * q,  
    word32 * qSz,  
    byte * g,  
    word32 * gSz  
)
```

C.17.2.19 function wc_DhGenerateParams

```
int wc_DhGenerateParams(  
    WC_RNG * rng,  
    int modSz,
```

```

    DhKey * dh
)

```

C.17.2.20 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.17.2.21 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
)

```

C.17.3 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);

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_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.18 dox_comments/header_files-ja/doxygen_groups.h

C.19 dox_comments/header_files-ja/doxygen_pages.h

C.20 dox_comments/header_files-ja/dsa.h

C.20.1 Functions

	Name
int	wc_InitDsaKey (DsaKey * key) この関数は、デジタル署名アルゴリズム (DSA) による認証に使用するために、DsaKey オブジェクトを初期化します。
void	wc_FreeDsaKey (DsaKey * key) この関数は、使用後に DsaKey オブジェクトを解放します。
int	wc_DsaSign (const byte * digest, byte * out, DsaKey * key, WC_RNG * rng) この関数は、入力ダイジェストに署名し、結果を出力バッファ out に格納します。

	Name
int	wc_DsaVerify (const byte * digest, const byte * sig, DsaKey * key, int * answer) この関数は、秘密鍵を使用してダイジェストの署名を検証します。検証が正しく行われたかどうかを answer パラメータに格納します。1 は検証成功、0 は検証失敗に対応します。
int	wc_DsaPublicKeyDecode (const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz) この関数は、DSA 公開鍵を含む DER フォーマットの証明書バッファをデコードし、与えられた DsaKey 構造体に鍵を格納します。また、読み取られた入力の長さに応じて inOutIdx パラメータを設定します。
int	wc_DsaPrivateKeyDecode (const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz) この関数は、DSA 秘密鍵を含む DER フォーマットの証明書バッファをデコードし、与えられた DsaKey 構造体に鍵を格納します。また、読み取られた入力の長さに応じて inOutIdx パラメータを設定します。
int	wc_DsaKeyToDer (DsaKey * key, byte * output, word32 inLen) DsaKey 鍵を DER フォーマットに変換し、output(inLen) に書き込み、書き込まれたバイト数を返します。
int	wc_MakeDsaKey (WC_RNG * rng, DsaKey * dsa) DSA 鍵を作成します。
int	wc_MakeDsaParameters (WC_RNG * rng, int modulus_size, DsaKey * dsa) FIPS 186_4 は、modulus_size 値として (1024, 160) (2048, 256) (3072, 256) を有効と定義しています

C.20.2 Functions Documentation

C.20.2.1 function wc_InitDsaKey

```
int wc_InitDsaKey(
    DsaKey * key
)
```

この関数は、デジタル署名アルゴリズム (DSA) による認証に使用するために、DsaKey オブジェクトを初期化します。

Parameters:

- **key** 初期化する DsaKey 構造体へのポインタ

See: [wc_FreeDsaKey](#)

Return:

- 0 成功時に返されます。
- BAD_FUNC_ARG NULL キーが渡された場合に返されます。

Example

```
DsaKey key;
int ret;
ret = wc_InitDsaKey(&key); // DSA 鍵を初期化
```

C.20.2.2 function wc_FreeDsaKey

```
void wc_FreeDsaKey(
    DsaKey * key
)
```

この関数は、使用後に DsaKey オブジェクトを解放します。

Parameters:

- **key** 解放する DsaKey 構造体へのポインタ

See: [wc_FreeDsaKey](#)

Return: none 戻り値なし。

Example

```
DsaKey key;
// 鍵を初期化、認証に使用
...
wc_FreeDsaKey(&key); // DSA 鍵を解放
```

C.20.2.3 function wc_DsaSign

```
int wc_DsaSign(
    const byte * digest,
    byte * out,
    DsaKey * key,
    WC_RNG * rng
)
```

この関数は、入力ダイジェストに署名し、結果を出力バッファ out に格納します。

Parameters:

- **digest** 署名するハッシュへのポインタ
- **out** 署名を格納するバッファへのポインタ
- **key** 署名を生成するために使用する初期化された DsaKey 構造体へのポインタ
- **rng** 署名生成で使用する初期化された RNG へのポインタ

See: [wc_DsaVerify](#)

Return:

- 0 入力ダイジェストへの署名に成功した場合に返されます
- MP_INIT_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_READ_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_MUL_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_TO_E DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- MP_MEM DSA 署名の処理中にエラーが発生した場合に返される可能性があります。

Example

```

DsaKey key;
// DSA 鍵を初期化、秘密鍵をロード
int ret;
WC_RNG rng;
wc_InitRng(&rng);
byte hash[] = { // ハッシュダイジェストで初期化 };
byte signature[40]; // 署名は 40 バイト (320 ビット) になります

ret = wc_DsaSign(hash, signature, &key, &rng);
if (ret != 0) {
    // DSA 署名生成エラー
}

```

C.20.2.4 function wc_DsaVerify

```

int wc_DsaVerify(
    const byte * digest,
    const byte * sig,
    DsaKey * key,
    int * answer
)

```

この関数は、秘密鍵を使用してダイジェストの署名を検証します。検証が正しく行われたかどうかを `answer` パラメータに格納します。1 は検証成功、0 は検証失敗に対応します。

Parameters:

- **digest** 署名の対象を含むダイジェストへのポインタ
- **sig** 検証する署名を含むバッファへのポインタ
- **key** 署名を検証するために使用する初期化された `DsaKey` 構造体へのポインタ
- **answer** 検証が成功したかどうかを格納する整数へのポインタ

See: `wc_DsaSign`

Return:

- 0 検証リクエストの処理に成功した場合に返されます。注意: これは署名が検証されたことを意味するのではなく、関数が成功したことを意味します
- `MP_INIT_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_READ_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_CMP_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_INVMOD_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_EXPTMOD_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_MOD_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_MUL_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_ADD_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_MULMOD_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_TO_E` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。
- `MP_MEM` DSA 署名の処理中にエラーが発生した場合に返される可能性があります。

Example

```

DsaKey key;
// DSA 鍵を初期化、公開鍵をロード

int ret;
int verified;

```

```

byte hash[] = { // ハッシュダイジェストで初期化 };
byte signature[] = { // 検証する署名で初期化 };
ret = wc_DsaVerify(hash, signature, &key, &verified);
if (ret != 0) {
    // 検証リクエストの処理エラー
} else if (answer == 0) {
    // 無効な署名
}

```

C.20.2.5 function wc_DsaPublicKeyDecode

```

int wc_DsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)

```

この関数は、DSA 公開鍵を含む DER フォーマットの証明書バッファをデコードし、与えられた DsaKey 構造体に鍵を格納します。また、読み取られた入力長さに応じて inOutIdx パラメータを設定します。

Parameters:

- **input** DER フォーマットの DSA 公開鍵を含むバッファへのポインタ
- **inOutIdx** 読み取られた証明書の最終インデックスを格納する整数へのポインタ
- **key** 公開鍵を格納する DsaKey 構造体へのポインタ
- **inSz** 入力バッファのサイズ

See:

- [wc_InitDsaKey](#)
- [wc_DsaPrivateKeyDecode](#)

Return:

- 0 DsaKey オブジェクトの公開鍵の設定に成功した場合に返されます
- ASN_PARSE_E 証明書バッファの読み取り中にエンコーディングエラーがある場合に返されます
- ASN_DH_KEY_E DSA パラメータの 1 つが正しくフォーマットされていない場合に返されます

Example

```

int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);
byte derBuff[] = { // DSA 公開鍵 };
ret = wc_DsaPublicKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // 公開鍵の読み取りエラー
}

```

C.20.2.6 function wc_DsaPrivateKeyDecode

```

int wc_DsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)

```

この関数は、DSA 秘密鍵を含む DER フォーマットの証明書バッファをデコードし、与えられた DsaKey 構造体に鍵を格納します。また、読み取られた入力の長さに応じて inOutIdx パラメータを設定します。

Parameters:

- **input** DER フォーマットの DSA 秘密鍵を含むバッファへのポインタ
- **inOutIdx** 読み取られた証明書の最終インデックスを格納する整数へのポインタ
- **key** 秘密鍵を格納する DsaKey 構造体へのポインタ
- **inSz** 入力バッファのサイズ

See:

- [wc_InitDsaKey](#)
- [wc_DsaPublicKeyDecode](#)

Return:

- 0 DsaKey オブジェクトの秘密鍵の設定に成功した場合に返されます
- ASN_PARSE_E 証明書バッファの読み取り中にエンコーディングエラーがある場合に返されます
- ASN_DH_KEY_E DSA パラメータの 1 つが正しくフォーマットされていない場合に返されます

Example

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);
byte derBuff[] = { // DSA 秘密鍵 };
ret = wc_DsaPrivateKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // 秘密鍵の読み取りエラー
}
```

C.20.2.7 function wc_DsaKeyToDer

```
int wc_DsaKeyToDer(
    DsaKey * key,
    byte * output,
    word32 inLen
)
```

DsaKey 鍵を DER フォーマットに変換し、output(inLen) に書き込み、書き込まれたバイト数を返します。

Parameters:

- **key** 変換する DsaKey 構造体へのポインタ。
- **output** 変換された鍵用の出力バッファへのポインタ。
- **inLen** 鍵入力の長さ。

See:

- [wc_InitDsaKey](#)
- [wc_FreeDsaKey](#)
- [wc_MakeDsaKey](#)

Return:

- outLen 成功、書き込まれたバイト数
- BAD_FUNC_ARG key または output が null または key->type が DSA_PRIVATE でない場合。
- MEMORY_E メモリ割り当てエラー。

Example

```

DsaKey key;
WC_RNG rng;
int derSz;
int bufferSize = // 十分なバッファサイズ;
byte der[bufferSize];

wc_InitDsaKey(&key);
wc_InitRng(&rng);
wc_MakeDsaKey(&rng, &key);
derSz = wc_DsaKeyToDer(&key, der, bufferSize);

```

C.20.2.8 function wc_MakeDsaKey

```

int wc_MakeDsaKey(
    WC_RNG * rng,
    DsaKey * dsa
)

```

DSA 鍵を作成します。

Parameters:

- **rng** WC_RNG 構造体へのポインタ。
- **dsa** DsaKey 構造体へのポインタ。

See:

- [wc_InitDsaKey](#)
- [wc_FreeDsaKey](#)
- [wc_DsaSign](#)

Return:

- MP_OKAY 成功
- BAD_FUNC_ARG rng または dsa が null の場合。
- MEMORY_E バッファ用のメモリを割り当てられませんでした。
- MP_INIT_E mp_int の初期化エラー

Example

```

WC_RNG rng;
DsaKey dsa;
wc_InitRng(&rng);
wc_InitDsa(&dsa);
if(wc_MakeDsaKey(&rng, &dsa) != 0)
{
    // 鍵作成エラー
}

```

C.20.2.9 function wc_MakeDsaParameters

```

int wc_MakeDsaParameters(
    WC_RNG * rng,
    int modulus_size,
    DsaKey * dsa
)

```

FIPS 186-4 は、modulus_size 値として (1024, 160) (2048, 256) (3072, 256) を有効と定義しています

Parameters:

- **rng** wolfCrypt rng へのポインタ。
- **modulus_size** 1024、2048、または 3072 が有効な値です。
- **dsa** DsaKey 構造体へのポインタ。

See:

- [wc_MakeDsaKey](#)
- [wc_DsaKeyToDer](#)
- [wc_InitDsaKey](#)

Return:

- 0 成功
- BAD_FUNC_ARG rng または dsa が null または modulus_size が無効な場合。
- MEMORY_E メモリ割り当て試行エラー。

Example

```
DsaKey key;
WC_RNG rng;
wc_InitDsaKey(&key);
wc_InitRng(&rng);
if(wc_MakeDsaParameters(&rng, 1024, &genKey) != 0)
{
    // エラーを処理
}
```

C.20.3 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_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 inLen);

int wc_MakeDsaKey(WC_RNG *rng, DsaKey *dsa);

int wc_MakeDsaParameters(WC_RNG *rng, int modulus_size, DsaKey *dsa);
```

C.21 dox_comments/header_files-ja/ecc.h

C.21.1 Functions

	Name
int	wc_ecc_make_key (WC_RNG * rng, int keysize, ecc_key * key) この関数は新しい ecc_key を生成し、key に格納します。
int	wc_ecc_make_key_ex (WC_RNG * rng, int keysize, ecc_key * key, int curve_id) この関数は新しい ecc_key を生成し、key に格納します。
int	wc_ecc_make_pub (ecc_key * key, ecc_point * pubOut) wc_ecc_make_pub は、既存の秘密成分を持つ ecc_key から公開成分を計算します。pubOut が提供されている場合、計算された公開鍵はそこに格納されます。そうでない場合は、提供された ecc_key の公開成分スロットに格納されます。
int	wc_ecc_make_pub_ex (ecc_key * key, ecc_point * pubOut, WC_RNG * rng) wc_ecc_make_pub_ex は、既存の秘密成分を持つ ecc_key から公開成分を計算します。pubOut が提供されている場合、計算された公開鍵はそこに格納されます。そうでない場合は、提供された ecc_key の公開成分スロットに格納されます。提供された rng が non-NULL の場合、計算で使用する秘密鍵値をブラインドするために使用されます。
int	wc_ecc_check_key (ecc_key * key) ecc キーの有効性に関する健全性チェックを実行します。
void	wc_ecc_key_free (ecc_key * key) この関数は、使用後に ecc_key キーを解放します。
int	wc_ecc_shared_secret (ecc_key * private_key, ecc_key * public_key, byte * out, word32 * outlen) この関数は、ローカル秘密鍵と受信した公開鍵を使用して新しい共有秘密鍵を生成します。この共有秘密鍵をバッファ out に格納し、outlen を更新して出力バッファに書き込まれたバイト数を保持します。
int	wc_ecc_shared_secret_ex (ecc_key * private_key, ecc_point * point, byte * out, word32 * outlen) 秘密鍵と公開ポイント間で ECC 共有秘密を作成します。
int	wc_ecc_sign_hash (const byte * in, word32 inlen, byte * out, word32 * outlen, WC_RNG * rng, ecc_key * key) この関数は、ecc_key オブジェクトを使用してメッセージダイジェストに署名し、真正性を保証します。
int	wc_ecc_sign_hash_ex (const byte * in, word32 inlen, WC_RNG * rng, ecc_key * key, mp_int * r, mp_int * s) メッセージダイジェストに署名します。
int	wc_ecc_verify_hash (const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * stat, ecc_key * key) この関数は、真正性を確保するためにハッシュの ECC 署名を検証します。stat を通じて答えを返し、1 は有効な署名に対応し、0 は無効な署名に対応します。

	Name
int	wc_ecc_verify_hash_ex (mp_int * r, mp_int * s, const byte * hash, word32 hashlen, int * stat, ecc_key * key) ECC 署名を検証します。結果は stat に書き込まれます。1 は有効、0 は無効です。注：有効性のテストに戻り値を使用しないでください。stat のみを使用してください。
int	wc_ecc_init (ecc_key * key) この関数は、メッセージ検証または鍵交換で将来使用するために ecc_key オブジェクトを初期化します。
int	wc_ecc_init_ex (ecc_key * key, void * heap, int devId) この関数は、メッセージ検証または鍵交換で将来使用するために ecc_key オブジェクトを初期化します。
ecc_key *	wc_ecc_key_new (void * heap) この関数は、ユーザー定義のヒープを使用し、キー構造体のスペースを割り当てます。
int	wc_ecc_free (ecc_key * key) この関数は、使用後に ecc_key オブジェクトを解放します。
void	wc_ecc_fp_free (void) この関数は、固定小数点キャッシュを解放します。これは計算時間を高速化するために ecc で使用できます。この機能を使用するには、FP_ECC (固定小数点 ecc) を定義する必要があります。スレッド化されたアプリケーションは、スレッドを終了する前にこの関数を呼び出す必要があります。
int	wc_ecc_is_valid_idx (int n) ECC idx が有効かどうかをチェックします。
ecc_point *	wc_ecc_new_point (void) 新しい ECC ポイントを割り当てます。
void	wc_ecc_del_point (ecc_point * p) メモリから ECC ポイントを解放します。
int	wc_ecc_copy_point (const ecc_point * p, ecc_point * r) あるポイントの値を別のポイントにコピーします。
int	wc_ecc_cmp_point (ecc_point * a, ecc_point * b) ポイントの値を別のポイントと比較します。
int	wc_ecc_point_is_at_infinity (ecc_point * p) ポイントが無限遠点にあるかどうかをチェックします。ポイントが無限遠点にある場合は 1 を返し、そうでない場合は 0 を返し、エラーの場合は <0 を返します
int	wc_ecc_mulmod (const mp_int * k, ecc_point * G, ecc_point * R, mp_int * a, mp_int * modulus, int map) ECC 固定点乗算を実行します。
int	wc_ecc_export_x963 (ecc_key * key, byte * out, word32 * outLen) この関数は、ecc_key 構造体から ECC キーをエクスポートし、結果を out に格納します。キーは ANSI X9.63 形式で格納されます。出力バッファに書き込まれたバイト数を outLen に格納します。

	Name
int	wc_ecc_export_x963_ex (ecc_key * key, byte * out, word32 * outLen, int compressed) この関数は、ecc_key 構造体から公開鍵をエクスポートし、結果を out に格納します。キーは ANSI X9.63 形式で格納されます。出力バッファに書き込まれたバイト数を outLen に格納します。この関数は、compressed パラメータを通じて証明書を圧縮する追加オプションを提供します。このパラメータが true の場合、キーは ANSI X9.63 圧縮形式で格納されます。
int	wc_ecc_import_x963 (const byte * in, word32 inLen, ecc_key * key) この関数は、ANSI X9.63 形式で格納されたキーを含むバッファから公開 ECC キーをインポートします。この関数は、コンパイル時に HAVE_COMP_KEY オプションを通じて圧縮キーが有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。
int	wc_ecc_import_private_key (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ecc_key * key) この関数は、生の秘密鍵を含むバッファと ANSI X9.63 形式の公開鍵を含む 2 番目のバッファから公開/秘密 ECC キーペアをインポートします。この関数は、コンパイル時に HAVE_COMP_KEY オプションを通じて圧縮キーが有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。
int	wc_ecc_rs_to_sig (const char * r, const char * s, byte * out, word32 * outlen) この関数は、ECC 署名の R と S 部分を DER エンコードされた ECDSA 署名に変換します。この関数は、出力バッファ out に書き込まれた長さも outlen に格納します。
int	wc_ecc_import_raw (ecc_key * key, const char * qx, const char * qy, const char * d, const char * curveName) この関数は、ECC 署名の生の成分で ecc_key 構造体を埋めます。
int	wc_ecc_export_private_only (ecc_key * key, byte * out, word32 * outLen) この関数は、ecc_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ecc_export_point_der (const int curve_idx, ecc_point * point, byte * out, word32 * outLen) ポイントを der 形式にエクスポートします。
int	wc_ecc_import_point_der (const byte * in, word32 inLen, const int curve_idx, ecc_point * point) der 形式からポイント进行インポートします。
int	wc_ecc_size (ecc_key * key) この関数は、ecc_key 構造体のキーサイズをオクテット単位で返します。

	Name
int	wc_ecc_sig_size_calc (int sz) この関数は、ECC 署名の最悪の場合のサイズを返します。これは $(keySz * 2) + SIG\_HEADER\_SZ + ECC\_MAX\_PAD\_SZ$ で与えられます。実際の署名サイズは、wc_ecc_sign_hash で計算できます。
int	wc_ecc_sig_size (const ecc_key * key) この関数は、ECC 署名の最悪の場合のサイズを返します。これは $(keySz * 2) + SIG\_HEADER\_SZ + ECC\_MAX\_PAD\_SZ$ で与えられます。実際の署名サイズは、wc_ecc_sign_hash で計算できます。
ecEncCtx *	wc_ecc_ctx_new (int flags, WC_RNG * rng) この関数は、ECC を使用した安全なメッセージ交換を可能にするために、新しい ECC コンテキストオブジェクト用のスペースを割り当てて初期化します。
void	wc_ecc_ctx_free (ecEncCtx * ctx) この関数は、メッセージの暗号化と復号に使用される ecEncCtx オブジェクトを解放します。
int	wc_ecc_ctx_reset (ecEncCtx * ctx, WC_RNG * rng) この関数は、新しいコンテキストオブジェクトを解放して割り当てる必要がないように、ecEncCtx 構造体をリセットします。
int	wc_ecc_ctx_set_algo (ecEncCtx * ctx, byte encAlgo, byte kdfAlgo, byte macAlgo) この関数は、wc_ecc_ctx_new の後にオプションで呼び出すことができます。ecEncCtx オブジェクトに暗号化、KDF、および MAC アルゴリズムを設定します。
const byte *	wc_ecc_ctx_get_own_salt (ecEncCtx *) この関数は、ecEncCtx オブジェクトのソルトを返します。この関数は、ecEncCtx の状態が ecSRV_INIT または ecCLI_INIT の場合にのみ呼び出す必要があります。
int	wc_ecc_ctx_set_peer_salt (ecEncCtx * ctx, const byte * salt) この関数は、ecEncCtx オブジェクトのピアソルトを設定します。
int	wc_ecc_ctx_set_kdf_salt (ecEncCtx * ctx, const byte * salt, word32 sz) この関数は、KDF で使用するソルトポインタと長さを ecEncCtx オブジェクトに設定します。
int	wc_ecc_ctx_set_info (ecEncCtx * ctx, const byte * info, int sz) この関数は、wc_ecc_ctx_set_peer_salt の前または後にオプションで呼び出すことができます。ecEncCtx オブジェクトのオプション情報を設定します。

	Name
int	wc_ecc_encrypt (ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx) この関数は、msg から out へ指定された入力メッセージを暗号化します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。
int	wc_ecc_encrypt_ex (ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx, int compressed) この関数は、msg から out へ指定された入力メッセージを暗号化します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。
int	wc_ecc_decrypt (ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx) この関数は、msg から out へ暗号文を復号します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。
int	wc_ecc_set_nonblock (ecc_key * key, ecc_nb_ctx_t * ctx) ノンブロッキング操作のための ECC サポートを有効にします。次のビルドオプションで Single Precision (SP) 数学でサポートされています：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) 指定されたサイズより大きいキーを持つカーブまたはカーブ ID に一致するカーブを比較し、より小さいキーサイズを持つカーブをキーに設定します。

C.21.2 Functions Documentation

C.21.2.1 function wc_ecc_make_key

```
int wc_ecc_make_key(  
    WC_RNG * rng,  
    int keysize,  
    ecc_key * key  
)
```

この関数は新しい ecc_key を生成し、key に格納します。

Parameters:

- **rng** キーの生成に使用する初期化された RNG オブジェクトへのポインタ
- **keysizes** ecc_key の希望する長さ
- **key** キーを生成する ecc_key へのポインタ

See:

- [wc_ecc_init](#)
- [wc_ecc_shared_secret](#)

Return:

- 0 成功時に返されます。
- ECC_BAD_ARG_E rng または key が NULL と評価された場合に返されます
- BAD_FUNC_ARG 指定されたキーサイズがサポートされているキーの正しい範囲内でない場合に返されます
- MEMORY_E ecc キーの計算中にメモリ割り当てエラーがある場合に返されます
- MP_INIT_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_READ_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_CMP_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MUL_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_ADD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_TO_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MEM ecc キーの計算中にエラーがある場合に返される可能性があります

Example

```
ecc_key key;  
wc_ecc_init(&key);  
WC_RNG rng;  
wc_InitRng(&rng);  
wc_ecc_make_key(&rng, 32, &key); // 32 バイトの ecc キーを初期化
```

C.21.2.2 function wc_ecc_make_key_ex

```
int wc_ecc_make_key_ex(  
    WC_RNG * rng,  
    int keysize,  
    ecc_key * key,  
    int curve_id  
)
```

この関数は新しい ecc_key を生成し、key に格納します。

Parameters:

- **key** 作成されたキーを格納するポインタ。
- **keysize** 作成されるキーのサイズ (バイト単位)、curveId に基づいて設定
- **rng** キー作成で使用される Rng
- **curve_id** キーに使用するカーブ

See:

- [wc_ecc_make_key](#)
- [wc_ecc_get_curve_size_from_id](#)

Return:

- 0 成功時に返されます。
- ECC_BAD_ARG_E rng または key が NULL と評価された場合に返されます
- BAD_FUNC_ARG 指定されたキーサイズがサポートされているキーの正しい範囲内でない場合に返されます
- MEMORY_E ecc キーの計算中にメモリ割り当てエラーがある場合に返されます
- MP_INIT_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_READ_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_CMP_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MUL_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_ADD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_TO_E ecc キーの計算中にエラーがある場合に返される可能性があります
- MP_MEM ecc キーの計算中にエラーがある場合に返される可能性があります

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) {
    // エラー処理
}
```

C.21.2.3 function wc_ecc_make_pub

```
int wc_ecc_make_pub(
    ecc_key * key,
    ecc_point * pubOut
)
```

wc_ecc_make_pub は、既存の秘密成分を持つ ecc_key から公開成分を計算します。pubOut が提供されている場合、計算された公開鍵はそこに格納されます。そうでない場合は、提供された ecc_key の公開成分スロットに格納されます。

Parameters:

- **key** 有効な秘密成分を含む ecc_key へのポインタ

- **pubOut** 計算された公開鍵を格納する ecc_point 構造体へのオプションのポインタ

See:

- [wc_ecc_make_pub_ex](#)
- [wc_ecc_make_key](#)

Return:

- 0 成功時に返されます。
- ECC_BAD_ARG_E key が NULL の場合に返されます
- BAD_FUNC_ARG 提供されたキーが有効な ecc_key でない場合に返されます。
- MEMORY_E 公開鍵の計算中にメモリ割り当てエラーがある場合に返されます
- MP_INIT_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_READ_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_ADD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_TO_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MEM 公開鍵の計算中にエラーがある場合に返される可能性があります
- ECC_OUT_OF_RANGE_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- ECC_PRIV_KEY_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- ECC_INF_E 公開鍵の計算中にエラーがある場合に返される可能性があります

C.21.2.4 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 は、既存の秘密成分を持つ ecc_key から公開成分を計算します。pubOut が提供されている場合、計算された公開鍵はそこに格納されます。そうでない場合は、提供された ecc_key の公開成分スロットに格納されます。提供された rng が non-NULL の場合、計算で使用される秘密鍵値をブラインドするために使用されます。

Parameters:

- **key** 有効な秘密成分を含む ecc_key へのポインタ
- **pubOut** 計算された公開鍵を格納する ecc_point 構造体へのオプションのポインタ
- **rng** 公開鍵の計算で使用する Rng

See:

- [wc_ecc_make_pub](#)
- [wc_ecc_make_key](#)
- [wc_InitRng](#)

Return:

- 0 成功時に返されます。
- ECC_BAD_ARG_E key が NULL の場合に返されます
- BAD_FUNC_ARG 提供されたキーが有効な ecc_key でない場合に返されます。
- MEMORY_E 公開鍵の計算中にメモリ割り当てエラーがある場合に返されます
- MP_INIT_E 公開鍵の計算中にエラーがある場合に返される可能性があります

- MP_READ_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_ADD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_TO_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- MP_MEM 公開鍵の計算中にエラーがある場合に返される可能性があります
- ECC_OUT_OF_RANGE_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- ECC_PRIV_KEY_E 公開鍵の計算中にエラーがある場合に返される可能性があります
- ECC_INF_E 公開鍵の計算中にエラーがある場合に返される可能性があります

C.21.2.5 function wc_ecc_check_key

```
int wc_ecc_check_key(
    ecc_key * key
)
```

ecc キーの有効性に関する健全性チェックを実行します。

Parameters:

- **key** チェックするキーへのポインタ。

See: [wc_ecc_point_is_at_infinity](#)

Return:

- MP_OKAY 成功、キーは正常です。
- BAD_FUNC_ARG key が NULL の場合に返されます。
- ECC_INF_E wc_ecc_point_is_at_infinity が 1 を返す場合に返されます。

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)
{
    // キーチェック成功
}
else
{
    // キーチェック失敗
}
```

C.21.2.6 function wc_ecc_key_free

```
void wc_ecc_key_free(
    ecc_key * key
)
```

この関数は、使用後に ecc_key キーを解放します。

Parameters:

- **key** 解放する ecc_key 構造体へのポインタ

See:

- `wc_ecc_key_new`
- `wc_ecc_init_ex`

Example

```
// キーを初期化して ECC 操作を実行
...
wc_ecc_key_free(&key);
```

C.21.2.7 function wc_ecc_shared_secret

```
int wc_ecc_shared_secret(
    ecc_key * private_key,
    ecc_key * public_key,
    byte * out,
    word32 * outlen
)
```

この関数は、ローカル秘密鍵と受信した公開鍵を使用して新しい共有秘密鍵を生成します。この共有秘密鍵をバッファ out に格納し、outlen を更新して出力バッファに書き込まれたバイト数を保持します。

Parameters:

- **private_key** ローカル秘密鍵を含む ecc_key 構造体へのポインタ
- **public_key** 受信した公開鍵を含む ecc_key 構造体へのポインタ
- **out** 生成された共有秘密鍵を格納する出力バッファへのポインタ
- **outlen** 出力バッファの長さを含む word32 オブジェクトへのポインタ。共有秘密鍵の生成に成功すると、出力バッファに書き込まれた長さで上書きされます

See:

- `wc_ecc_init`
- `wc_ecc_make_key`

Return:

- 0 共有秘密鍵の生成に成功した場合に返されます
- BAD_FUNC_ARG いずれかの入力パラメータが NULL と評価された場合に返されます
- ECC_BAD_ARG_E 引数として与えられた秘密鍵 private_key のタイプが ECC_PRIVATEKEY でない場合、または公開鍵と秘密鍵のタイプ (ecc->dp で指定) が同等でない場合に返されます
- MEMORY_E 新しい ecc ポイントの生成中にエラーがある場合に返されます
- BUFFER_E 生成された共有秘密鍵が提供されたバッファに格納するには長すぎる場合に返されます
- MP_INIT_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_READ_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_ADD_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_TO_E 共有鍵の計算中にエラーがある場合に返される可能性があります
- MP_MEM 共有鍵の計算中にエラーがある場合に返される可能性があります

Example

```

ecc_key priv, pub;
WC_RNG rng;
byte secret[1024]; // 1024 バイトの共有秘密鍵を保持可能
word32 secretSz = sizeof(secret);
int ret;

wc_InitRng(&rng); // rng を初期化
wc_ecc_init(&priv); // キーを初期化
wc_ecc_make_key(&rng, 32, &priv); // 公開/秘密鍵ペアを作成
// 公開鍵を受信し、pub に初期化
ret = wc_ecc_shared_secret(&priv, &pub, secret, &secretSz);
// 共有秘密鍵を生成
if ( ret != 0 ) {
    // 共有秘密鍵の生成エラー
}

```

C.21.2.8 function wc_ecc_shared_secret_ex

```

int wc_ecc_shared_secret_ex(
    ecc_key * private_key,
    ecc_point * point,
    byte * out,
    word32 * outlen
)

```

秘密鍵と公開ポイント間で ECC 共有秘密を作成します。

Parameters:

- **private_key** 秘密 ECC キー。
- **point** 使用するポイント（公開鍵）。
- **out** 共有秘密の出力先。ANSI X9.63 の EC-DH に準拠。
- **outlen** 最大サイズを入力し、共有秘密の結果サイズを出力。

See: [wc_ecc_verify_hash_ex](#)

Return:

- MP_OKAY 成功を示します。
- BAD_FUNC_ARG いずれかの引数が null の場合に返されるエラー。
- ECC_BAD_ARG_E private_key->type が ECC_PRIVATEKEY でない場合、または private_key->idx の検証に失敗した場合に返されるエラー。
- BUFFER_E outlen が小さすぎる場合のエラー。
- MEMORY_E 新しいポイントを作成する際のエラー。
- MP_VAL 初期化失敗が発生した場合に返される可能性があります。
- MP_MEM 初期化失敗が発生した場合に返される可能性があります。

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)
{
    // エラーを処理
}
```

C.21.2.9 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
)
```

この関数は、ecc_key オブジェクトを使用してメッセージダイジェストに署名し、真正性を保証します。

Parameters:

- **in** 署名するメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージハッシュの長さ
- **out** 生成された署名を格納するバッファ
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功すると、out に書き込まれたバイト数を格納します
- **key** 署名を生成するために使用する秘密 ECC キーへのポインタ

See: [wc_ecc_verify_hash](#)

Return:

- 0 メッセージダイジェストの署名を正常に生成した場合に返されます
- BAD_FUNC_ARG いずれかの入力パラメータが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます
- ECC_BAD_ARG_E 入力キーが秘密鍵でない場合、または ECC OID が無効な場合に返されます
- RNG_FAILURE_E rng が満足のいくキーを正常に生成できない場合に返されます
- MP_INIT_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_READ_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_ADD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_TO_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MEM メッセージ署名の計算中にエラーがある場合に返される可能性があります

Example

```
ecc_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[512]; // 生成された署名を保持
```

```

sigSz = sizeof(sig);
byte digest[] = { // メッセージハッシュで初期化 };
wc_InitRng(&rng); // rng を初期化
wc_ecc_init(&key); // キーを初期化
wc_ecc_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
ret = wc_ecc_sign_hash(digest, sizeof(digest), sig, &sigSz, &key);
if ( ret != 0 ) {
    // メッセージ署名の生成エラー
}

```

C.21.2.10 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
)

```

メッセージダイジェストに署名します。

Parameters:

- **in** 署名するメッセージダイジェスト。
- **inlen** ダイジェストの長さ。
- **rng** WC_RNG 構造体へのポインタ。
- **key** 秘密 ECC キー。
- **r** 署名の r 成分の出力先。
- **s** 署名の s 成分の出力先。

See: [wc_ecc_verify_hash_ex](#)

Return:

- MP_OKAY メッセージダイジェストの署名を正常に生成した場合に返されます
- ECC_BAD_ARG_E 入力キーが秘密鍵でない場合、または ECC IDX が無効な場合、またはいずれかの入力パラメータが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます
- RNG_FAILURE_E rng が満足のいくキーを正常に生成できない場合に返されます
- MP_INIT_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_READ_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_ADD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_TO_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MEM メッセージ署名の計算中にエラーがある場合に返される可能性があります

Example

```

ecc_key key;
WC_RNG rng;
int ret, sigSz;

```

```

mp_int r; // 署名の r 成分の出力先。
mp_int s; // 署名の s 成分の出力先。

byte sig[512]; // 生成された署名を保持
sigSz = sizeof(sig);
byte digest[] = { メッセージハッシュで初期化 };
wc_InitRng(&rng); // rng を初期化
wc_ecc_init(&key); // キーを初期化
mp_init(&r); // r 成分を初期化
mp_init(&s); // s 成分を初期化
wc_ecc_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
ret = wc_ecc_sign_hash_ex(digest, sizeof(digest), &rng, &key, &r, &s);

if ( ret != MP_OKAY ) {
    // メッセージ署名の生成エラー
}

```

C.21.2.11 function wc_ecc_verify_hash

```

int wc_ecc_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashlen,
    int * stat,
    ecc_key * key
)

```

この関数は、真正性を確保するためにハッシュの ECC 署名を検証します。stat を通じて答えを返し、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ
- **siglen** 検証する署名の長さ
- **hash** 検証されたメッセージのハッシュを含むバッファへのポインタ
- **hashlen** 検証されたメッセージのハッシュの長さ
- **stat** 検証結果へのポインタ。1 はメッセージが正常に検証されたことを示します
- **key** 署名を検証するために使用する公開 ECC キーへのポインタ

See:

- [wc_ecc_sign_hash](#)
- [wc_ecc_verify_hash_ex](#)

Return:

- 0 署名検証の実行に成功した場合に返されます。注：これは署名が検証されたことを意味するものではありません。真正性情報は代わりに stat に格納されます
- BAD_FUNC_ARG いずれかの入力パラメータが NULL と評価された場合に返されます
- MEMORY_E メモリ割り当てエラーがある場合に返されます
- MP_INIT_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_READ_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_CMP_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_INVMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MUL_E メッセージ署名の計算中にエラーがある場合に返される可能性があります

- MP_ADD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MULMOD_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_TO_E メッセージ署名の計算中にエラーがある場合に返される可能性があります
- MP_MEM メッセージ署名の計算中にエラーがある場合に返される可能性があります

Example

```
ecc_key key;
int ret, verified = 0;

byte sig[1024] { 受信した署名で初期化 };
byte digest[] = { メッセージハッシュで初期化 };
// 受信した公開鍵でキーを初期化
ret = wc_ecc_verify_hash(sig, sizeof(sig), digest, sizeof(digest),
&verified, &key);
if ( ret != 0 ) {
    // 検証実行エラー
} else if ( verified == 0 ) {
    // 署名が無効
}
```

C.21.2.12 function wc_ecc_verify_hash_ex

```
int wc_ecc_verify_hash_ex(
    mp_int * r,
    mp_int * s,
    const byte * hash,
    word32 hashlen,
    int * stat,
    ecc_key * key
)
```

ECC 署名を検証します。結果は stat に書き込まれます。1 は有効、0 は無効です。注：有効性のテストに戻り値を使用しないでください。stat のみを使用してください。

Parameters:

- **r** 検証する署名の R 成分
- **s** 検証する署名の S 成分
- **hash** 署名されたハッシュ（メッセージダイジェスト）
- **hashlen** ハッシュの長さ（オクテット）
- **stat** 署名の結果、1== 有効、0== 無効
- **key** 対応する公開 ECC キー

See: [wc_ecc_verify_hash](#)

Return:

- MP_OKAY 成功した場合（署名が有効でない場合でも）
- ECC_BAD_ARG_E 引数が null の場合、または key-idx が無効な場合に返されます。
- MEMORY_E 整数またはポイントの割り当てエラー。

Example

```
mp_int r;
mp_int s;
int stat;
byte hash[] = { いくつかのハッシュ }
ecc_key key;
```



```
if(wc_ecc_verify_hash_ex(&r, &s, hash, hashlen, &stat, &key) == MP_OKAY)
{
    // stat をチェック
}
```

C.21.2.13 function wc_ecc_init

```
int wc_ecc_init(
    ecc_key * key
)
```

この関数は、メッセージ検証または鍵交換で将来使用するために ecc_key オブジェクトを初期化します。

Parameters:

- **key** 初期化する ecc_key オブジェクトへのポインタ

See:

- [wc_ecc_make_key](#)
- [wc_ecc_free](#)

Return:

- 0 ecc_key オブジェクトの初期化に成功した場合に返されます
- MEMORY_E メモリ割り当てエラーがある場合に返されます

Example

```
ecc_key key;
wc_ecc_init(&key);
```

C.21.2.14 function wc_ecc_init_ex

```
int wc_ecc_init_ex(
    ecc_key * key,
    void * heap,
    int devId
)
```

この関数は、メッセージ検証または鍵交換で将来使用するために ecc_key オブジェクトを初期化します。

Parameters:

- **key** 初期化する ecc_key オブジェクトへのポインタ
- **heap** ヒープ識別子へのポインタ
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定

See:

- [wc_ecc_make_key](#)
- [wc_ecc_free](#)
- [wc_ecc_init](#)

Return:

- 0 ecc_key オブジェクトの初期化に成功した場合に返されます
- MEMORY_E メモリ割り当てエラーがある場合に返されます

Example


```
ecc_key key;  
wc_ecc_init_ex(&key, heap, devId);
```

C.21.2.15 function `wc_ecc_key_new`

```
ecc_key * wc_ecc_key_new(  
    void * heap  
)
```

この関数は、ユーザー定義のヒープを使用し、キー構造体用のスペースを割り当てます。

See:

- `wc_ecc_make_key`
- `wc_ecc_key_free`
- `wc_ecc_init`

Return:

- 0 `ecc_key` オブジェクトの初期化に成功した場合に返されます
- `MEMORY_E` メモリ割り当てエラーがある場合に返されます

Example

```
wc_ecc_key_new(&heap);
```

C.21.2.16 function `wc_ecc_free`

```
int wc_ecc_free(  
    ecc_key * key  
)
```

この関数は、使用後に `ecc_key` オブジェクトを解放します。

Parameters:

- **key** 解放する `ecc_key` オブジェクトへのポインタ

See: `wc_ecc_init`

Return: int wolfSSL のエラーまたは成功ステータスを示す整数が返されます。

Example

```
// キーを初期化して安全な交換を実行  
...  
wc_ecc_free(&key);
```

C.21.2.17 function `wc_ecc_fp_free`

```
void wc_ecc_fp_free(  
    void  
)
```

この関数は、固定小数点キャッシュを解放します。これは計算時間を高速化するために `ecc` で使用できます。この機能を使用するには、`FP_ECC` (固定小数点 `ecc`) を定義する必要があります。スレッド化されたアプリケーションは、スレッドを終了する前にこの関数を呼び出す必要があります。

Parameters:

- **none** パラメータなし。

See: `wc_ecc_free`

Return: none 返り値なし。

Example

```
ecc_key key;  
// キーを初期化して安全な交換を実行  
...  
  
wc_ecc_fp_free();
```

C.21.2.18 function `wc_ecc_is_valid_idx`

```
int wc_ecc_is_valid_idx(  
    int n  
)
```

ECC idx が有効かどうかをチェックします。

Parameters:

- **n** チェックする idx 番号。

See: なし

Return:

- 1 有効な場合に返されます。
- 0 有効でない場合に返されます。

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 は有効  
}  
else if (is_valid == 0)  
{  
    // idx は無効  
}
```

C.21.2.19 function `wc_ecc_new_point`

```
ecc_point * wc_ecc_new_point(  
    void  
)
```

新しい ECC ポイントを割り当てます。

Parameters:

- **none** パラメータなし。

See:

- `wc_ecc_del_point`
- `wc_ecc_cmp_point`
- `wc_ecc_copy_point`

Return:

- `p` 新しく割り当てられたポイント。
- NULL エラー時に NULL を返します。

Example

```
ecc_point* point;
point = wc_ecc_new_point();
if (point == NULL)
{
    // ポイント作成エラーを処理
}
// ポイントで何かを行う
```

C.21.2.20 function `wc_ecc_del_point`

```
void wc_ecc_del_point(
    ecc_point * p
)
```

メモリから ECC ポイントを解放します。

Parameters:

- `p` 解放するポイント。

See:

- `wc_ecc_new_point`
- `wc_ecc_cmp_point`
- `wc_ecc_copy_point`

Return: none 返り値なし。

Example

```
ecc_point* point;
point = wc_ecc_new_point();
if (point == NULL)
{
    // ポイント作成エラーを処理
}
// ポイントで何かを行う
wc_ecc_del_point(point);
```

C.21.2.21 function `wc_ecc_copy_point`

```
int wc_ecc_copy_point(
    const ecc_point * p,
    ecc_point * r
)
```

あるポイントの値を別のポイントにコピーします。

Parameters:

- `p` コピーするポイント。

- **r** 作成されたポイント。

See:

- `wc_ecc_new_point`
- `wc_ecc_cmp_point`
- `wc_ecc_del_point`

Return:

- `ECC_BAD_ARG_E` `p` または `r` が `null` の場合にスローされるエラー。
- `MP_OKAY` ポイントが正常にコピーされました
- `ret` 内部関数からのエラー。次のような可能性があります...

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)
{
    // エラーを処理
}
```

C.21.2.22 function wc_ecc_cmp_point

```
int wc_ecc_cmp_point(
    ecc_point * a,
    ecc_point * b
)
```

ポイントの値を別のポイントと比較します。

Parameters:

- **a** 比較する最初のポイント。
- **b** 比較する 2 番目のポイント。

See:

- `wc_ecc_new_point`
- `wc_ecc_del_point`
- `wc_ecc_copy_point`

Return:

- `BAD_FUNC_ARG` 一方または両方の引数が `NULL`。
- `MP_EQ` ポイントが等しい。
- `ret` `MP_LT` または `MP_GT` のいずれかで、ポイントが等しくないことを示します。

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)
{
    // 引数が無効
}
else if (cmp_result == MP_EQ)
{
    // ポイントが等しい
}
else
{
    // ポイントが等しくない
}

```

C.21.2.23 function wc_ecc_point_is_at_infinity

```

int wc_ecc_point_is_at_infinity(
    ecc_point * p
)

```

ポイントが無限遠点にあるかどうかをチェックします。ポイントが無限遠点にある場合は 1 を返し、そうでない場合は 0 を返し、エラーの場合は <0 を返します

Parameters:

- **p** チェックするポイント。

See:

- [wc_ecc_new_point](#)
- [wc_ecc_del_point](#)
- [wc_ecc_cmp_point](#)
- [wc_ecc_copy_point](#)

Return:

- 1 p が無限遠点にある。
- 0 p が無限遠点にない。
- <0 エラー。

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)
{
    // エラーを処理
}
else if (is_infinity == 0)
{
    // ポイントは無限遠点にない
}
else if (is_infinity == 1)
{
    // ポイントは無限遠点にある
}

```

C.21.2.24 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
)
```

ECC 固定点乗算を実行します。

Parameters:

- **k** 被乗数。
- **G** 乗算する基点。
- **R** 積の出力先。
- **modulus** カーブのモジュラス。
- **map** ゼロでない場合、ポイントをアフィン座標に戻してマップします。そうでない場合は、ヤコビ・モンゴメリー形式のままです。

See: なし

Return:

- MP_OKAY 操作が成功した場合に返されます。
- MP_INIT_E 多精度整数 (mp_int) ライブラリで使用するために整数を初期化の際にエラーがある場合に返されます。

Example

```
ecc_point* base;
ecc_point* destination;
// ポイントを初期化
base = wc_ecc_new_point();
destination = wc_ecc_new_point();
// 他の引数を設定
mp_int multiplicand;
mp_int modulus;
int map;
```

C.21.2.25 function wc_ecc_export_x963

```
int wc_ecc_export_x963(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ecc_key 構造体から ECC キーをエクスポートし、結果を out に格納します。キーは ANSI X9.63 形式で格納されます。出力バッファに書き込まれたバイト数を outLen に格納します。

Parameters:

- **key** エクスポートする ecc_key オブジェクトへのポインタ
- **out** ANSI X9.63 形式のキーを格納するバッファへのポインタ
- **outLen** 出力バッファのサイズ。キーの格納に成功すると、出力バッファに書き込まれたバイト数を保持します

See:

- `wc_ecc_export_x963_ex`
- `wc_ecc_import_x963`
- `wc_ecc_make_pub`

Return:

- 0 `ecc_key` のエクスポートに成功した場合に返されます
- `LENGTH_ONLY_E` 出力バッファが `NULL` と評価されるが、他の 2 つの入力パラメータが有効な場合に返されます。関数がキーを格納するために必要な長さのみを返していることを示します
- `ECC_BAD_ARG_E` いずれかの入力パラメータが `NULL` の場合、またはキーがサポートされていない (無効なインデックスを持つ) 場合に返されます
- `BUFFER_E` 出力バッファが `ecc` キーを格納するには小さすぎる場合に返されます。出力バッファが小さすぎる場合、必要なサイズが `outLen` で返されます
- `MEMORY_E` `XMALLOC` でメモリを割り当てる際にエラーがある場合に返されます
- `MP_INIT_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_READ_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_CMP_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_INVMOD_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_EXPTMOD_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MOD_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MUL_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_ADD_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MULMOD_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_TO_E` `ecc_key` の処理中にエラーがある場合に返される可能性があります
- `MP_MEM` `ecc_key` の処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);

ecc_key key;
// キーを初期化し、キーを作成
ret = wc_ecc_export_x963(&key, buff, &buffSz);
if ( ret != 0 ) {
    // キーのエクスポートエラー
}
```

C.21.2.26 function `wc_ecc_export_x963_ex`

```
int wc_ecc_export_x963_ex(
    ecc_key * key,
    byte * out,
    word32 * outLen,
    int compressed
)
```

この関数は、`ecc_key` 構造体から公開鍵をエクスポートし、結果を `out` に格納します。キーは ANSI X9.63 形式で格納されます。出力バッファに書き込まれたバイト数を `outLen` に格納します。この関数は、`compressed` パラメータを通じて証明書を圧縮する追加オプションを提供します。このパラメータが `true` の場合、キーは ANSI X9.63 圧縮形式で格納されます。

Parameters:

- **key** エクスポートする `ecc_key` オブジェクトへのポインタ
- **out** ANSI X9.63 形式の公開鍵を格納するバッファへのポインタ

- **outLen** 出力バッファのサイズ。公開鍵の格納に成功すると、出力バッファに書き込まれたバイト数を保持します
- **compressed** キーを圧縮形式で格納するかどうかの指標。1== 圧縮、0== 非圧縮

See:

- [wc_ecc_export_x963](#)
- [wc_ecc_import_x963](#)
- [wc_ecc_make_pub](#)

Return:

- 0 ecc_key 公開成分のエクスポートに成功した場合に返されます
- ECC_PRIVATEKEY_ONLY ecc_key 公開成分が欠落している場合に返されます
- NOT_COMPILED_IN コンパイル時に HAVE_COMP_KEY が有効になっていないが、キーが圧縮形式で要求された場合に返されます
- LENGTH_ONLY_E 出力バッファが NULL と評価されるが、他の 2 つの入力パラメータが有効な場合に返されます。関数が公開鍵を格納するために必要な長さのみを返していることを示します
- ECC_BAD_ARG_E いずれかの入力パラメータが NULL の場合、またはキーがサポートされていない（無効なインデックスを持つ）場合に返されます
- BUFFER_E 出力バッファが公開鍵を格納するには小さすぎる場合に返されます。出力バッファが小さすぎる場合、必要なサイズが outLen で返されます
- MEMORY_E XMALLOC でメモリを割り当てる際にエラーがある場合に返されます
- MP_INIT_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_READ_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_CMP_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_INVMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MUL_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_ADD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MULMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_TO_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MEM ecc_key の処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);
ecc_key key;
// キーを初期化し、キーを作成
ret = wc_ecc_export_x963_ex(&key, buff, &buffSz, 1);
if ( ret != 0 ) {
    // キーのエクスポートエラー
}
```

C.21.2.27 function wc_ecc_import_x963

```
int wc_ecc_import_x963(
    const byte * in,
    word32 inLen,
    ecc_key * key
)
```

この関数は、ANSI X9.63 形式で格納されたキーを含むバッファから公開 ECC キーをインポートします。この関数は、コンパイル時に HAVE_COMP_KEY オプションを通じて圧縮キーが有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **in** ANSI x9.63 形式の ECC キーを含むバッファへのポインタ
- **inLen** 入力バッファの長さ
- **key** インポートされたキーを格納する ecc_key オブジェクトへのポインタ

See:

- [wc_ecc_export_x963](#)
- [wc_ecc_import_private_key](#)

Return:

- 0 ecc_key のインポートに成功した場合に返されます
- NOT_COMPILED_IN コンパイル時に HAVE_COMP_KEY が有効になっていないが、キーが圧縮形式で格納されている場合に返されます
- ECC_BAD_ARG_E in または key が NULL と評価される場合、または inLen が偶数の場合に返されます (x9.63 標準によると、キーは奇数でなければなりません)
- MEMORY_E メモリの割り当てエラーがある場合に返されます
- ASN_PARSE_E ECC キーの解析エラーがある場合に返されます。ECC キーが有効な ANSI X9.63 形式で格納されていないことを示す可能性があります
- IS_POINT_E エクスポートされた公開鍵が ECC 曲線上のポイントでない場合に返されます
- MP_INIT_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_READ_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_CMP_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_INVMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MUL_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_ADD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MULMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_TO_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MEM ecc_key の処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
byte buff[] = { ANSI X9.63形式のキーで初期化 };

ecc_key pubKey;
wc_ecc_init(&pubKey);

ret = wc_ecc_import_x963(buff, sizeof(buff), &pubKey);
if ( ret != 0 ) {
    // キーのインポートエラー
}
```

C.21.2.28 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
)
```

この関数は、生の秘密鍵を含むバッファと ANSI X9.63 形式の公開鍵を含む 2 番目のバッファから公開/秘密 ECC キーペアをインポートします。この関数は、コンパイル時に HAVE_COMP_KEY オプションを通じて圧縮キーが有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **priv** 生の秘密鍵を含むバッファへのポインタ
- **privSz** 秘密鍵バッファのサイズ
- **pub** ANSI x9.63 形式の ECC 公開鍵を含むバッファへのポインタ
- **pubSz** 公開鍵入力バッファの長さ
- **key** インポートされた秘密/公開鍵ペアを格納する ecc_key オブジェクトへのポインタ

See:

- [wc_ecc_export_x963](#)
- [wc_ecc_import_private_key](#)

Return:

- 0 ecc_key のインポートに成功した場合に返されます NOT_COMPILED_IN コンパイル時に HAVE_COMP_KEY が有効になっていないが、キーが圧縮形式で格納されている場合に返されます
- ECC_BAD_ARG_E in または key が NULL と評価される場合、または inLen が偶数の場合に返されます (x9.63 標準によると、キーは奇数でなければなりません)
- MEMORY_E メモリの割り当てエラーがある場合に返されます
- ASN_PARSE_E ECC キーの解析エラーがある場合に返されます。ECC キーが有効な ANSI X9.63 形式で格納されていないことを示す可能性があります
- IS_POINT_E エクスポートされた公開鍵が ECC 曲線上のポイントでない場合に返されます
- MP_INIT_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_READ_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_CMP_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_INVMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MUL_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_ADD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MULMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_TO_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MEM ecc_key の処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
byte pub[] = { ANSI X9.63形式のキーで初期化 };
byte priv[] = { 生の秘密鍵で初期化 };

ecc_key key;
wc_ecc_init(&key);
ret = wc_ecc_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
&key);
if ( ret != 0 ) {
    // キーのインポートエラー
}
```

C.21.2.29 function wc_ecc_rs_to_sig

```
int wc_ecc_rs_to_sig(
    const char * r,
    const char * s,
```

```

    byte * out,
    word32 * outlen
)

```

この関数は、ECC 署名の R と S 部分を DER エンコードされた ECDSA 署名に変換します。この関数は、出力バッファ out に書き込まれた長さも outlen に格納します。

Parameters:

- **r** 文字列として署名の R 部分を含むバッファへのポインタ
- **s** 文字列として署名の S 部分を含むバッファへのポインタ
- **out** DER エンコードされた ECDSA 署名を格納するバッファへのポインタ
- **outlen** 利用可能な出力バッファの長さ。署名を ECDSA 形式に正常に変換した後、バッファに書き込まれたバイト数を格納します

See:

- [wc_ecc_sign_hash](#)
- [wc_ecc_sig_size](#)

Return:

- 0 署名の変換に成功した場合に返されます
- ECC_BAD_ARG_E いずれかの入力パラメータが NULL と評価される場合、または入力バッファが DER エンコードされた ECDSA 署名を保持するのに十分な大きさでない場合に返されます
- MP_INIT_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_READ_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_CMP_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_INVMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MUL_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_ADD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MULMOD_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_TO_E ecc_key の処理中にエラーがある場合に返される可能性があります
- MP_MEM ecc_key の処理中にエラーがある場合に返される可能性があります

Example

```

int ret;
ecc_key key;
// キーを初期化し、R と S を生成

char r[] = { R で初期化 };
char s[] = { S で初期化 };
byte sig[wc_ecc_sig_size(key)];
// 署名サイズは 2 * ECC キーサイズ + ASN.1 オーバーヘッド用の約 10 バイトになります
word32 sigSz = sizeof(sig);
ret = wc_ecc_rs_to_sig(r, s, sig, &sigSz);
if ( ret != 0 ) {
    // パラメータから署名への変換エラー
}

```

C.21.2.30 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
)

```

この関数は、ECC 署名の生の成分で ecc_key 構造体を埋めます。

Parameters:

- **key** 埋める ecc_key 構造体へのポインタ
- **qx** ASCII 16 進文字列として基点の x 成分を含むバッファへのポインタ
- **qy** ASCII 16 進文字列として基点の y 成分を含むバッファへのポインタ
- **d** ASCII 16 進文字列として秘密鍵を含むバッファへのポインタ
- **curveName** ecc_sets にある ECC 曲線名を含む文字列へのポインタ

See: [wc_ecc_import_private_key](#)

Return:

- 0 ecc_key 構造体へのインポートに成功した場合に返されます
- ECC_BAD_ARG_E いずれかの入力値が NULL と評価された場合に返されます
- MEMORY_E ecc_key のパラメータを格納するためのスペースを初期化する際にエラーがある場合に返されます
- ASN_PARSE_E 入力 curveName が ecc_sets で定義されていない場合に返されます
- MP_INIT_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_READ_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_CMP_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_INVMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MUL_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_ADD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MULMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_TO_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MEM 入力パラメータの処理中にエラーがある場合に返される可能性があります

Example

```

int ret;
ecc_key key;
wc_ecc_init(&key);

char qx[] = { 基点の x 成分で初期化 };
char qy[] = { 基点の y 成分で初期化 };
char d[] = { 秘密鍵で初期化 };
ret = wc_ecc_import_raw(&key, qx, qy, d, "ECC-256");
if ( ret != 0 ) {
    // 指定された入力でキーを初期化する際のエラー
}

```

C.21.2.31 function wc_ecc_export_private_only

```

int wc_ecc_export_private_only(
    ecc_key * key,
    byte * out,
    word32 * outLen
)

```

この関数は、ecc_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 秘密鍵をエクスポートする ecc_key 構造体へのポインタ
- **out** 秘密鍵を格納するバッファへのポインタ
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。秘密鍵のエクスポートに成功した後、out に書き込まれたバイト数で設定されます

See: `wc_ecc_import_private_key`

Return:

- 0 秘密鍵のエクスポートに成功した場合に返されます
- ECC_BAD_ARG_E いずれかの入力値が NULL と評価された場合に返されます
- MEMORY_E ecc_key のパラメータを格納するためのスペースを初期化する際にエラーがある場合に返されます
- ASN_PARSE_E 入力 curveName が ecc_sets で定義されていない場合に返されます
- MP_INIT_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_READ_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_CMP_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_INVMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_EXPTMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MUL_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_ADD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MULMOD_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_TO_E 入力パラメータの処理中にエラーがある場合に返される可能性があります
- MP_MEM 入力パラメータの処理中にエラーがある場合に返される可能性があります

Example

```
int ret;
ecc_key key;
// キーを初期化し、キーを作成

char priv[ECC_KEY_SIZE];
word32 privSz = sizeof(priv);
ret = wc_ecc_export_private_only(&key, priv, &privSz);
if ( ret != 0 ) {
    // 秘密鍵のエクスポートエラー
}
```

C.21.2.32 function wc_ecc_export_point_der

```
int wc_ecc_export_point_der(
    const int curve_idx,
    ecc_point * point,
    byte * out,
    word32 * outLen
)
```

ポイントを der 形式にエクスポートします。

Parameters:

- **curve_idx** ecc_sets から使用されるカーブのインデックス。
- **point** der にエクスポートするポイント。
- **out** 出力の出力先。
- **outLen** 出力に許可される最大サイズ、出力の最終サイズの出力先

See: [wc_ecc_import_point_der](#)

Return:

- 0 成功時に返されます。
- ECC_BAD_ARG_E curve_idx が 0 未満または無効な場合に返されます。また、次の場合にも返されます
- LENGTH_ONLY_E outLen が設定されているが他は何もない。
- BUFFER_E outLen が 1 + 2 * カーブサイズ未満の場合に返されます。
- MEMORY_E メモリの割り当てに問題がある場合に返されます。

Example

```
int curve_idx;
ecc_point* point;
byte out[];
word32 outLen;
wc_ecc_export_point_der(curve_idx, point, out, &outLen);
```

C.21.2.33 function wc_ecc_import_point_der

```
int wc_ecc_import_point_der(
    const byte * in,
    word32 inLen,
    const int curve_idx,
    ecc_point * point
)
```

der 形式からポイントをインポートします。

Parameters:

- **in** ポイントをインポートする der バッファ。
- **inLen** der バッファの長さ。
- **curve_idx** カーブのインデックス。
- **point** ポイントの出力先。

See: [wc_ecc_export_point_der](#)

Return:

- ECC_BAD_ARG_E いずれかの引数が null の場合、または inLen が偶数の場合に返されます。
- MEMORY_E 初期化中にエラーがある場合に返されます
- NOT_COMPILED_IN HAVE_COMP_KEY が true でなく、in が圧縮証明書の場合に返されます
- MP_OKAY 操作が成功しました。

Example

```
byte in[];
word32 inLen;
int curve_idx;
ecc_point* point;
wc_ecc_import_point_der(in, inLen, curve_idx, point);
```

C.21.2.34 function wc_ecc_size

```
int wc_ecc_size(
    ecc_key * key
)
```


この関数は、ecc_key 構造体のキーサイズをオクテット単位で返します。

Parameters:

- **key** キーサイズを取得する ecc_key 構造体へのポインタ

See: [wc_ecc_make_key](#)

Return:

- Given a valid key, 有効なキーが与えられた場合、キーサイズをオクテット単位で返します
- 0 指定されたキーが NULL の場合に返されます

Example

```
int keySz;
ecc_key key;
// キーを初期化し、キーを作成
keySz = wc_ecc_size(&key);
if ( keySz == 0 ) {
    // キーサイズの判定エラー
}
```

C.21.2.35 function wc_ecc_sig_size_calc

```
int wc_ecc_sig_size_calc(
    int sz
)
```

この関数は、ECC 署名の最悪の場合のサイズを返します。これは (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ で与えられます。実際の署名サイズは、wc_ecc_sign_hash で計算できます。

Parameters:

- **key size** キーサイズ

See:

- [wc_ecc_sign_hash](#)
- [wc_ecc_sig_size](#)

Return: returns 最大署名サイズをオクテット単位で返します

Example

```
int sigSz = wc_ecc_sig_size_calc(32);
if ( sigSz == 0 ) {
    // 署名サイズの判定エラー
}
```

C.21.2.36 function wc_ecc_sig_size

```
int wc_ecc_sig_size(
    const ecc_key * key
)
```

この関数は、ECC 署名の最悪の場合のサイズを返します。これは (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ で与えられます。実際の署名サイズは、wc_ecc_sign_hash で計算できます。

Parameters:

- **key** 署名サイズを取得する ecc_key 構造体へのポインタ

See:

- `wc_ecc_sign_hash`
- `wc_ecc_sig_size_calc`

Return:

- Success 有効なキーが与えられた場合、最大署名サイズをオクテット単位で返します
- 0 指定されたキーが NULL の場合に返されます

Example

```
int sigSz;
ecc_key key;
// キーを初期化し、キーを作成

sigSz = wc_ecc_sig_size(&key);
if ( sigSz == 0 ) {
    // 署名サイズの判定エラー
}
```

C.21.2.37 function wc_ecc_ctx_new

```
ecEncCtx * wc_ecc_ctx_new(
    int flags,
    WC_RNG * rng
)
```

この関数は、ECC を使用した安全なメッセージ交換を可能にするために、新しい ECC コンテキストオブジェクト用のスペースを割り当てて初期化します。

Parameters:

- **flags** これがサーバーコンテキストかクライアントコンテキストかを示します オプションは：REQ_RESP_CLIENT および REQ_RESP_SERVER
- **rng** ソルトを生成するために使用する RNG オブジェクトへのポインタ

See:

- `wc_ecc_encrypt`
- `wc_ecc_encrypt_ex`
- `wc_ecc_decrypt`

Return:

- Success 新しい ecEncCtx オブジェクトの生成に成功すると、そのオブジェクトへのポインタを返します
- NULL 関数が新しい ecEncCtx オブジェクトの生成に失敗した場合に返されます

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
if(ctx == NULL) {
    // 新しい ecEncCtx オブジェクトの生成エラー
}
```

C.21.2.38 function wc_ecc_ctx_free


```
void wc_ecc_ctx_free(  
    ecEncCtx * ctx  
)
```

この関数は、メッセージの暗号化と復号に使用される ecEncCtx オブジェクトを解放します。

Parameters:

- **ctx** 解放する ecEncCtx オブジェクトへのポインタ

See: [wc_ecc_ctx_new](#)

Return: none 返り値なし。

Example

```
ecEncCtx* ctx;  
WC_RNG rng;  
wc_InitRng(&rng);  
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);  
// 安全な通信を行う  
...  
wc_ecc_ctx_free(&ctx);
```

C.21.2.39 function wc_ecc_ctx_reset

```
int wc_ecc_ctx_reset(  
    ecEncCtx * ctx,  
    WC_RNG * rng  
)
```

この関数は、新しいコンテキストオブジェクトを解放して割り当てる必要がないように、ecEncCtx 構造体をリセットします。

Parameters:

- **ctx** リセットする ecEncCtx オブジェクトへのポインタ
- **rng** 新しいソルトを生成するために使用する RNG オブジェクトへのポインタ

See: [wc_ecc_ctx_new](#)

Return:

- 0 ecEncCtx 構造体が正常にリセットされた場合に返されます
- BAD_FUNC_ARG rng または ctx のいずれかが NULL の場合に返されます
- RNG_FAILURE_E ECC オブジェクト用の新しいソルトを生成する際にエラーがある場合に返されます

Example

```
ecEncCtx* ctx;  
WC_RNG rng;  
wc_InitRng(&rng);  
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);  
// 安全な通信を行う  
...  
wc_ecc_ctx_reset(&ctx, &rng);  
// さらに安全な通信を行う
```

C.21.2.40 function wc_ecc_ctx_set_algo

```
int wc_ecc_ctx_set_algo(
    ecEncCtx * ctx,
    byte encAlgo,
    byte kdfAlgo,
    byte macAlgo
)
```

この関数は、wc_ecc_ctx_new の後にオプションで呼び出すことができます。ecEncCtx オブジェクトに暗号化、KDF、および MAC アルゴリズムを設定します。

Parameters:

- **ctx** 情報を設定する ecEncCtx へのポインタ
- **encAlgo** 使用する暗号化アルゴリズム。
- **kdfAlgo** 使用する KDF アルゴリズム。
- **macAlgo** 使用する MAC アルゴリズム。

See: [wc_ecc_ctx_new](#)

Return:

- 0 ecEncCtx オブジェクトの情報を正常に設定した場合に返されます。
- BAD_FUNC_ARG 指定された ecEncCtx オブジェクトが NULL の場合に返されます。

Example

```
ecEncCtx* ctx;
// ctx を初期化
if(wc_ecc_ctx_set_algo(&ctx, ecAES_128_CTR, ecHKDF_SHA256, ecHMAC_SHA256)) {
    // 情報設定エラー
}
```

C.21.2.41 function wc_ecc_ctx_get_own_salt

```
const byte * wc_ecc_ctx_get_own_salt(
    ecEncCtx *
)
```

この関数は、ecEncCtx オブジェクトのソルトを返します。この関数は、ecEncCtx の状態が ecSRV_INIT または ecCLI_INIT の場合にのみ呼び出す必要があります。

Parameters:

- **ctx** ソルトを取得する ecEncCtx オブジェクトへのポインタ

See:

- [wc_ecc_ctx_new](#)
- [wc_ecc_ctx_set_peer_salt](#)
- [wc_ecc_ctx_set_kdf_salt](#)

Return:

- Success 成功時に、ecEncCtx ソルトを返します
- NULL ecEncCtx オブジェクトが NULL の場合、または ecEncCtx の状態が ecSRV_INIT または ecCLI_INIT でない場合に返されます。後者の 2 つのケースでは、この関数は ecEncCtx の状態をそれぞれ ecSRV_BAD_STATE または ecCLI_BAD_STATE に設定します

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) {
    // ソルト取得エラー
}

```

C.21.2.42 function wc_ecc_ctx_set_peer_salt

```

int wc_ecc_ctx_set_peer_salt(
    ecEncCtx * ctx,
    const byte * salt
)

```

この関数は、ecEncCtx オブジェクトのピアソルトを設定します。

Parameters:

- **ctx** ソルトを設定する ecEncCtx へのポインタ
- **salt** ピアのソルトへのポインタ

See:

- [wc_ecc_ctx_get_own_salt](#)
- [wc_ecc_ctx_set_kdf_salt](#)

Return:

- 0 ecEncCtx オブジェクトのピアソルトを正常に設定した場合に返されます。
- BAD_FUNC_ARG 指定された ecEncCtx オブジェクトが NULL または無効なプロトコルを持つ場合、または指定されたソルトが NULL の場合に返されます
- BAD_ENC_STATE_E ecEncCtx の状態が ecSRV_SALT_GET または ecCLI_SALT_GET の場合に返されます。後者の 2 つのケースでは、この関数は ecEncCtx の状態をそれぞれ ecSRV_BAD_STATE または ecCLI_BAD_STATE に設定します

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.21.2.43 function wc_ecc_ctx_set_kdf_salt

```

int wc_ecc_ctx_set_kdf_salt(
    ecEncCtx * ctx,
    const byte * salt,
    word32 sz
)

```

この関数は、KDF で使用するソルトポインタと長さを ecEncCtx オブジェクトに設定します。

Parameters:

- **ctx** ソルトを設定する ecEncCtx へのポインタ
- **salt** ソルトバッファへのポインタ
- **sz** ソルトの長さ (バイト単位)

See:

- [wc_ecc_ctx_get_own_salt](#)
- [wc_ecc_ctx_get_peer_salt](#)

Return:

- 0 ecEncCtx オブジェクトのソルトを正常に設定した場合に返されます。
- BAD_FUNC_ARG 指定された ecEncCtx オブジェクトが NULL の場合、または指定されたソルトが NULL で長さが NULL でない場合に返されます。

Example

```
ecEncCtx* srvCtx;
WC_RNG rng;
byte cliSalt[] = { 固定ソルトデータ };
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.21.2.44 function wc_ecc_ctx_set_info

```
int wc_ecc_ctx_set_info(
    ecEncCtx * ctx,
    const byte * info,
    int sz
)
```

この関数は、wc_ecc_ctx_set_peer_salt の前または後にオプションで呼び出すことができます。ecEncCtx オブジェクトのオプション情報を設定します。

Parameters:

- **ctx** 情報を設定する ecEncCtx へのポインタ
- **info** 設定する情報を含むバッファへのポインタ
- **sz** info バッファのサイズ

See: [wc_ecc_ctx_new](#)

Return:

- 0 ecEncCtx オブジェクトの情報を正常に設定した場合に返されます。
- BAD_FUNC_ARG 指定された ecEncCtx オブジェクトが NULL の場合、入力 info が NULL の場合、またはそのサイズが無効な場合に返されます

Example

```
ecEncCtx* ctx;
byte info[] = { 情報で初期化 };
// ctx を初期化、ソルトを取得、
```

```
if(wc_ecc_ctx_set_info(&ctx, info, sizeof(info))) {
    // 情報設定エラー
}
```

C.21.2.45 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
)
```

この関数は、msg から out へ指定された入力メッセージを暗号化します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。

Parameters:

- **privKey** 暗号化に使用する秘密鍵を含む ecc_key オブジェクトへのポインタ
- **pubKey** 通信したいピアの公開鍵を含む ecc_key オブジェクトへのポインタ
- **msg** 暗号化するメッセージを保持するバッファへのポインタ
- **msgSz** 暗号化するバッファのサイズ
- **out** 暗号化された暗号文を格納するバッファへのポインタ
- **outSz** out バッファで利用可能なサイズを含む word32 オブジェクトへのポインタ。メッセージの暗号化に成功すると、出力バッファに書き込まれたバイト数を保持します
- **ctx** オプション：使用する異なる暗号化アルゴリズムを指定する ecEncCtx オブジェクトへのポインタ

See:

- [wc_ecc_encrypt_ex](#)
- [wc_ecc_decrypt](#)

Return:

- 0 入力メッセージの暗号化に成功した場合に返されます
- BAD_FUNC_ARG privKey、pubKey、msg、msgSz、out、または outSz が NULL の場合、または ctx オブジェクトがサポートされていない暗号化タイプを指定している場合に返されます
- BAD_ENC_STATE_E 指定された ctx オブジェクトが暗号化に適していない状態にある場合に返されます
- BUFFER_E 提供された出力バッファが暗号化された暗号文を格納するには小さすぎる場合に返されます
- MEMORY_E 共有秘密鍵用のメモリを割り当てる際にエラーがある場合に返されます

Example

```
byte msg[] = { 暗号化するメッセージで初期化。ブロックサイズにパディングされていることを確認
    ↪ };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// cli を秘密鍵で初期化
```

```
// serv を受信した公開鍵で初期化

ecEncCtx* cliCtx, servCtx;
// cliCtx と servCtx を初期化
// ソルトを交換
ret = wc_ecc_encrypt(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx);
if(ret != 0) {
    // メッセージの暗号化エラー
}
```

C.21.2.46 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
)
```

この関数は、msg から out へ指定された入力メッセージを暗号化します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。

Parameters:

- **privKey** 暗号化に使用する秘密鍵を含む ecc_key オブジェクトへのポインタ
- **pubKey** 通信したいピアの公開鍵を含む ecc_key オブジェクトへのポインタ
- **msg** 暗号化するメッセージを保持するバッファへのポインタ
- **msgSz** 暗号化するバッファのサイズ
- **out** 暗号化された暗号文を格納するバッファへのポインタ
- **outSz** out バッファで利用可能なサイズを含む word32 オブジェクトへのポインタ。メッセージの暗号化に成功すると、出力バッファに書き込まれたバイト数を保持します
- **ctx** オプション：使用する異なる暗号化アルゴリズムを指定する ecEncCtx オブジェクトへのポインタ
- **compressed** 公開鍵フィールドを圧縮形式で出力する。

See:

- [wc_ecc_encrypt](#)
- [wc_ecc_decrypt](#)

Return:

- 0 入力メッセージの暗号化に成功した場合に返されます
- BAD_FUNC_ARG privKey、pubKey、msg、msgSz、out、または outSz が NULL の場合、または ctx オブジェクトがサポートされていない暗号化タイプを指定している場合に返されます
- BAD_ENC_STATE_E 指定された ctx オブジェクトが暗号化に適していない状態にある場合に返されます
- BUFFER_E 提供された出力バッファが暗号化された暗号文を格納するには小さすぎる場合に返されます
- MEMORY_E 共有秘密鍵用のメモリを割り当てる際にエラーがある場合に返されます

Example

```

byte msg[] = { 暗号化するメッセージで初期化。ブロックサイズにパディングされていることを確認
    ↪ };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// cli を秘密鍵で初期化
// serv を受信した公開鍵で初期化

ecEncCtx* cliCtx, servCtx;
// cliCtx と servCtx を初期化
// ソルトを交換
ret = wc_ecc_encrypt_ex(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx,
    1);
if(ret != 0) {
    // メッセージの暗号化エラー
}

```

C.21.2.47 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
)

```

この関数は、msg から out へ暗号文を復号します。この関数は、オプションの ctx オブジェクトをパラメータとして受け取ります。提供された場合、暗号化は ecEncCtx の encAlgo、kdfAlgo、および macAlgo に基づいて進行します。ctx が提供されない場合、デフォルトのアルゴリズム ecAES_128_CBC、ecHKDF_SHA256、および ecHMAC_SHA256 で処理が完了します。この関数では、ctx で指定された暗号化タイプに応じてメッセージがパディングされている必要があります。

Parameters:

- **privKey** 復号に使用する秘密鍵を含む ecc_key オブジェクトへのポインタ
- **pubKey** 通信したいピアの公開鍵を含む ecc_key オブジェクトへのポインタ
- **msg** 復号する暗号文を保持するバッファへのポインタ
- **msgSz** 復号するバッファのサイズ
- **out** 復号された平文を格納するバッファへのポインタ
- **outSz** out バッファで利用可能なサイズを含む word32 オブジェクトへのポインタ。暗号文の復号に成功すると、出力バッファに書き込まれたバイト数を保持します
- **ctx** オプション：使用する異なる復号アルゴリズムを指定する ecEncCtx オブジェクトへのポインタ

See:

- [wc_ecc_encrypt](#)
- [wc_ecc_encrypt_ex](#)

Return:

- 0 入力メッセージの復号に成功した場合に返されます

- BAD_FUNC_ARG privKey、pubKey、msg、msgSz、out、または outSz が NULL の場合、または ctx オブジェクトがサポートされていない暗号化タイプを指定している場合に返されます
- BAD_ENC_STATE_E 指定された ctx オブジェクトが復号に適していない状態にある場合に返されます
- BUFFER_E 提供された出力バッファが復号された平文を格納するには小さすぎる場合に返されます
- MEMORY_E 共有秘密鍵用のメモリを割り当てる際にエラーがある場合に返されます

Example

```
byte cipher[] = { 復号する暗号文で初期化。ブロックサイズにパディングされていることを確認 };
byte plain[sizeof(cipher)];
word32 plainSz = sizeof(plain);
int ret;
ecc_key cli, serv;
// cli を秘密鍵で初期化
// serv を受信した公開鍵で初期化
ecEncCtx* cliCtx, servCtx;
// cliCtx と servCtx を初期化
// ソルトを交換
ret = wc_ecc_decrypt(&cli, &serv, cipher, sizeof(cipher),
plain, &plainSz, cliCtx);

if(ret != 0) {
    // メッセージの復号エラー
}
```

C.21.2.48 function wc_ecc_set_nonblock

```
int wc_ecc_set_nonblock(
    ecc_key * key,
    ecc_nb_ctx_t * ctx
)
```

ノンブロッキング操作のための ECC サポートを有効にします。次のビルドオプションで Single Precision (SP) 数学でサポートされています: WOLFSSL_SP_NONBLOCK WOLFSSL_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK

Parameters:

- **key** ecc_key オブジェクトへのポインタ
- **ctx** SP 用のスタックデータキャッシュを持つ ecc_nb_ctx_t 構造体へのポインタ

Return: 0 入力メッセージのコールバックコンテキストの設定に成功した場合に返されます

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,           // mp_int としての r/s
                hash, hashSz,    // 計算されたハッシュダイジェスト
                &verify_res,     // 検証結果 1= 成功
            );
        } while (ret != 0);
    }
}
```



```

        &key
    );

    // TODO: リアルタイム作業をここで呼び出すことができます
    } while (ret == FP_WOULDBLOCK);
}
wc_ecc_free(&key);
}

```

C.21.2.49 function wc_ecc_set_curve

```

int wc_ecc_set_curve(
    ecc_key * key,
    int keysize,
    int curve_id
)

```

指定されたサイズより大きいキーを持つカーブまたはカーブ ID に一致するカーブを比較し、より小さいキーサイズを持つカーブをキーに設定します。

Parameters:

- **keysizes** キーサイズ (バイト単位)
- **curve_id** カーブ 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 キーの設定に成功した場合に返されます

C.21.3 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* stat, ecc_key* key);

int wc_ecc_verify_hash_ex(mp_int *r, mp_int *s, const byte* hash,
                        word32 hashlen, int* stat, 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);

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_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); /* 割り当て/解放なしで再使用のため
↪ にリセット */

int wc_ecc_ctx_set_algo(ecEncCtx* ctx, byte encAlgo, byte kdfAlgo,
                       byte macAlgo);

const byte* wc_ecc_ctx_get_own_salt(ecEncCtx*);

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);
```

C.22 dox_comments/header_files-ja/eccsi.h

C.22.1 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.22.2 Functions Documentation

C.22.2.1 function wc_InitEccsiKey

```
int wc_InitEccsiKey(
    EccsiKey * key,
    void * heap,
    int devId
)
```

C.22.2.2 function wc_InitEccsiKey_ex

```
int wc_InitEccsiKey_ex(
    EccsiKey * key,
    int keySz,
    int curveId,
    void * heap,
    int devId
)
```

C.22.2.3 function wc_FreeEccsiKey

```
void wc_FreeEccsiKey(
    EccsiKey * key
)
```

C.22.2.4 function wc_MakeEccsiKey

```
int wc_MakeEccsiKey(
    EccsiKey * key,
    WC_RNG * rng
)
```

C.22.2.5 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.22.2.6 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.22.2.7 function wc_ValidateEccsiPvt

```
int wc_ValidateEccsiPvt(  
    EccsiKey * key,  
    const ecc_point * pvt,  
    int * valid  
)
```

C.22.2.8 function wc_EncodeEccsiPair

```
int wc_EncodeEccsiPair(  
    const EccsiKey * key,  
    mp_int * ssk,  
    ecc_point * pvt,  
    byte * data,  
    word32 * sz  
)
```

C.22.2.9 function wc_EncodeEccsiSsk

```
int wc_EncodeEccsiSsk(  
    const EccsiKey * key,  
    mp_int * ssk,  
    byte * data,  
    word32 * sz  
)
```

C.22.2.10 function wc_EncodeEccsiPvt

```
int wc_EncodeEccsiPvt(  
    const EccsiKey * key,  
    ecc_point * pvt,  
    byte * data,  
    word32 * sz,  
    int raw  
)
```

C.22.2.11 function wc_DecodeEccsiPair

```
int wc_DecodeEccsiPair(  
    const EccsiKey * key,  
    const byte * data,  
    word32 sz,  
    mp_int * ssk,  
    ecc_point * pvt  
)
```

C.22.2.12 function wc_DecodeEccsiSsk

```
int wc_DecodeEccsiSsk(  
    const EccsiKey * key,  
    const byte * data,  
    word32 sz,  
    mp_int * ssk  
)
```

C.22.2.13 function wc_DecodeEccsiPvt

```
int wc_DecodeEccsiPvt(  
    const EccsiKey * key,  
    const byte * data,  
    word32 sz,  
    ecc_point * pvt  
)
```

C.22.2.14 function wc_DecodeEccsiPvtFromSig

```
int wc_DecodeEccsiPvtFromSig(  
    const EccsiKey * key,  
    const byte * sig,  
    word32 sz,  
    ecc_point * pvt  
)
```

C.22.2.15 function wc_ExportEccsiKey

```
int wc_ExportEccsiKey(  
    EccsiKey * key,  
    byte * data,  
    word32 * sz  
)
```

C.22.2.16 function wc_ImportEccsiKey

```
int wc_ImportEccsiKey(  
    EccsiKey * key,  
    const byte * data,  
    word32 sz  
)
```

C.22.2.17 function wc_ExportEccsiPrivateKey

```
int wc_ExportEccsiPrivateKey(  
    EccsiKey * key,  
    byte * data,  
    word32 * sz  
)
```

C.22.2.18 function wc_ImportEccsiPrivateKey

```
int wc_ImportEccsiPrivateKey(  
    EccsiKey * key,  
    const byte * data,  
    word32 sz  
)
```

C.22.2.19 function wc_ExportEccsiPublicKey

```
int wc_ExportEccsiPublicKey(  
    EccsiKey * key,  
    byte * data,  
    word32 * sz,  
    int raw  
)
```

C.22.2.20 function wc_ImportEccsiPublicKey

```
int wc_ImportEccsiPublicKey(  
    EccsiKey * key,  
    const byte * data,  
    word32 sz,  
    int trusted  
)
```

C.22.2.21 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.22.2.22 function wc_SetEccsiHash

```
int wc_SetEccsiHash(  
    EccsiKey * key,  
    const byte * hash,  
    byte hashSz  
)
```


C.22.2.23 function wc_SetEccsiPair

```
int wc_SetEccsiPair(
    EccsiKey * key,
    const mp_int * ssk,
    const ecc_point * pvt
)
```

C.22.2.24 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.22.2.25 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.22.3 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.23 dox_comments/header_files-ja/ed25519.h

C.23.1 Functions

	Name
int	wc_ed25519_make_public (ed25519_key * key, unsigned char * pubKey, word32 pubKeySz) この関数は、ed25519_key オブジェクトに格納された秘密鍵から Ed25519 公開鍵を生成します。公開鍵をバッファ pubKey に格納します。
int	wc_ed25519_make_key (WC_RNG * rng, int keysize, ed25519_key * key) この関数は新しい Ed25519 キーを生成し、それを key に格納します。

	Name
int	wc_ed25519_sign_msg (const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key) この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージに署名します。
int	wc_ed25519ctx_sign_msg (const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部です。
int	wc_ed25519ph_sign_hash (const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed25519_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されます。
int	wc_ed25519ph_sign_msg (const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されます。
int	wc_ed25519_verify_msg (const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key) この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
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) この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

	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) この関数は、真正性を保証するためにメッセージのダイジェストの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。メッセージダイジェストの作成に使用されるハッシュアルゴリズムは SHA-512 である必要があります。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
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) この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。メッセージは検証前に事前ハッシュ化されます。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
int	wc_ed25519_init (ed25519_key * key) この関数は、メッセージ検証での将来の使用のために ed25519_key オブジェクトを初期化します。
void	wc_ed25519_free (ed25519_key * key) この関数は、使用後に Ed25519 オブジェクトを解放します。
int	wc_ed25519_import_public (const byte * in, word32 inLen, ed25519_key * key) この関数は、公開鍵を含むバッファから公開 ed25519_key をインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は、秘密鍵が存在する場合にそれと一致するかチェックされます。
int	wc_ed25519_import_public_ex (const byte * in, word32 inLen, ed25519_key * key, int trusted) この関数は、公開鍵を含むバッファから公開 ed25519_key をインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、秘密鍵が存在するときに公開鍵が秘密鍵と一致するかチェックします。
int	wc_ed25519_import_private_only (const byte * priv, word32 privSz, ed25519_key * key) この関数は、バッファから Ed25519 秘密鍵のみをインポートします。

	Name
int	wc_ed25519_import_private_key (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key) この関数は、一対のバッファから公開/秘密 Ed25519 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は信頼されていないと想定され、秘密鍵に対してチェックされます。
int	wc_ed25519_import_private_key_ex (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key, int trusted) この関数は、一対のバッファから公開/秘密 Ed25519 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、公開鍵が秘密鍵に対してチェックされます。
int	wc_ed25519_export_public (const ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体から公開鍵をエクスポートします。公開鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ed25519_export_private_only (const ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ed25519_export_private (const ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ed25519_export_key (const ed25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、ed25519_key 構造体から秘密鍵と公開鍵を個別にエクスポートします。秘密鍵をバッファ priv に格納し、このバッファに書き込まれたバイト数を privSz に設定します。公開鍵をバッファ pub に格納し、このバッファに書き込まれたバイト数を pubSz に設定します。
int	wc_ed25519_check_key (ed25519_key * key) この関数は、ed25519_key 構造体内の公開鍵が秘密鍵と一致するかチェックします。
int	wc_ed25519_size (const ed25519_key * key) この関数は、Ed25519 のサイズ (32 バイト) を返します。
int	wc_ed25519_priv_size (const ed25519_key * key) この関数は、秘密鍵のサイズ (秘密 + 公開) をバイト単位で返します。

	Name
int	wc_ed25519_pub_size (const ed25519_key * key) この関数は、圧縮鍵のサイズをバイト単位で返します（公開鍵）。
int	wc_ed25519_sig_size (const ed25519_key * key) この関数は、Ed25519 署名のサイズ（バイト単位で 64）を返します。

C.23.2 Functions Documentation

C.23.2.1 function wc_ed25519_make_public

```
int wc_ed25519_make_public(
    ed25519_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

この関数は、ed25519_key オブジェクトに格納された秘密鍵から Ed25519 公開鍵を生成します。公開鍵をバッファ pubKey に格納します。

Parameters:

- **key** キーを生成する ed25519_key へのポインタ。
- **pubKey** 公開鍵を格納するバッファへのポインタ。
- **pubKeySz** 公開鍵のサイズ。ED25519_PUB_KEY_SIZE である必要があります。

See:

- **wc_ed25519_init**
- **wc_ed25519_import_private_only**
- **wc_ed25519_make_key**

Return:

- 0 公開鍵の作成に成功した場合に返されます。
- BAD_FUNC_ARG key または pubKey が NULL と評価された場合、または指定されたキーサイズが 32 バイトでない場合に返されます（Ed25519 は 32 バイトのキーを持ちます）。
- ECC_PRIV_KEY_E ed25519_key オブジェクトに秘密鍵が含まれていない場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
int ret;

ed25519_key key;
byte priv[] = { 32バイトの秘密鍵で初期化 };
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) {
    // 公開鍵の作成エラー
}
```

C.23.2.2 function wc_ed25519_make_key

```
int wc_ed25519_make_key(
    WC_RNG * rng,
    int keysize,
    ed25519_key * key
)
```

この関数は新しい Ed25519 キーを生成し、それを key に格納します。

Parameters:

- **rng** キーを生成するために使用する初期化済み RNG オブジェクトへのポインタ。
- **keysizes** 生成するキーの長さ。Ed25519 の場合は常に 32 である必要があります。
- **key** キーを生成する ed25519_key へのポインタ。

See: [wc_ed25519_init](#)

Return:

- 0 ed25519_key の作成に成功した場合に返されます。
- BAD_FUNC_ARG rng または key が NULL と評価された場合、または指定されたキーサイズが 32 バイトでない場合に返されます (Ed25519 は 32 バイトのキーを持ちます)。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

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) {
    // キー作成エラー
}
```

C.23.2.3 function wc_ed25519_sign_msg

```
int wc_ed25519_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key
)
```

この関数は、真正性を保証するために ed25519_key オブジェクトを使用してメッセージに署名します。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージの長さ。
- **out** 生成された署名を格納するバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 ed25519_key へのポインタ。

See:

- `wc_ed25519ctx_sign_msg`
- `wc_ed25519ph_sign_hash`
- `wc_ed25519ph_sign_msg`
- `wc_ed25519_verify_msg`

Return:

- 0 メッセージの署名の生成に成功した場合に返されます。
- `BAD_FUNC_ARG` 入力パラメータのいずれかが `NULL` と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- `MEMORY_E` 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // 生成された署名を保持
sigSz = sizeof(sig);
byte message[] = { メッセージで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed25519_init(&key); // key を初期化
wc_ed25519_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed25519_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0) {
    // メッセージ署名の生成エラー
}
```

C.23.2.4 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
)
```

この関数は、真正性を保証するために `ed25519_key` オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部です。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージの長さ。
- **out** 生成された署名を格納するバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 `ed25519_key` へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- `wc_ed25519_sign_msg`

- `wc_ed25519ph_sign_hash`
- `wc_ed25519ph_sign_msg`
- `wc_ed25519_verify_msg`

Return:

- 0 メッセージの署名の生成に成功した場合に返されます。
- `BAD_FUNC_ARG` 入力パラメータのいずれかが `NULL` と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- `MEMORY_E` 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // 生成された署名を保持
sigSz = sizeof(sig);
byte message[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed25519_init(&key); // key を初期化
wc_ed25519_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed25519ctx_sign_msg(message, sizeof(message), sig, &sigSz, &key,
    context, sizeof(context));
if (ret != 0) {
    // メッセージ署名の生成エラー
}
```

C.23.2.5 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
)
```

この関数は、真正性を保証するために `ed25519_key` オブジェクトを使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されます。

Parameters:

- **hash** 署名するメッセージのハッシュを含むバッファへのポインタ。
- **hashLen** 署名するメッセージのハッシュの長さ。
- **out** 生成された署名を格納するバッファ。
- **outLen** 出力バッファの最大長。メッセージ署名の生成に成功した後、`out` に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 `ed25519_key` へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- `wc_ed25519_sign_msg`
- `wc_ed25519ctx_sign_msg`
- `wc_ed25519ph_sign_msg`
- `wc_ed25519_verify_msg`

Return:

- 0 メッセージダイジェストの署名の生成に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // 生成された署名を保持
sigSz = sizeof(sig);
byte hash[] = { メッセージの SHA-512ハッシュで初期化 };
byte context[] = { 署名のコンテキストで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed25519_init(&key); // key を初期化
wc_ed25519_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed25519ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
                             context, sizeof(context));
if (ret != 0) {
    // メッセージ署名の生成エラー
}
```

C.23.2.6 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
)
```

この関数は、真正性を保証するために `ed25519_key` オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されます。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージの長さ。
- **out** 生成された署名を格納するバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 `ed25519_key` へのポインタ。

- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed25519_sign_msg](#)
- [wc_ed25519ctx_sign_msg](#)
- [wc_ed25519ph_sign_hash](#)
- [wc_ed25519_verify_msg](#)

Return:

- 0 メッセージの署名の生成に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // 生成された署名を保持
sigSz = sizeof(sig);
byte message[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed25519_init(&key); // key を初期化
wc_ed25519_make_key(&rng, 32, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed25519ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
                             context, sizeof(context));
if (ret != 0) {
    // メッセージ署名の生成エラー
}
```

C.23.2.7 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
)
```

この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **msg** 検証するメッセージを含むバッファへのポインタ。
- **msgLen** 検証するメッセージの長さ。
- **ret** 検証結果へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するために使用する公開 Ed25519 鍵へのポインタ。

See:

- `wc_ed25519ctx_verify_msg`
- `wc_ed25519ph_verify_hash`
- `wc_ed25519ph_verify_msg`
- `wc_ed25519_sign_msg`

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さとは一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };
byte msg[] = { メッセージで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed25519_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
    &key);
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}
```

C.23.2.8 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
)
```

この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **msg** 検証するメッセージを含むバッファへのポインタ。
- **msgLen** 検証するメッセージの長さ。
- **ret** 検証結果へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するために使用する公開 Ed25519 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- `wc_ed25519_verify_msg`
- `wc_ed25519ph_verify_hash`
- `wc_ed25519ph_verify_msg`
- `wc_ed25519_sign_msg`

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さとは一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };
byte msg[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed25519ctx_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
    &verified, &key, );
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}
```

C.23.2.9 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
)
```

この関数は、真正性を保証するためにメッセージのダイジェストの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。メッセージダイジェストの作成に使用されるハッシュアルゴリズムは SHA-512 である必要があります。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **hash** 検証するメッセージのハッシュを含むバッファへのポインタ。
- **hashLen** 検証するハッシュの長さ。
- **ret** 検証結果へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するために使用する公開 Ed25519 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed25519_verify_msg](#)
- [wc_ed25519ctx_verify_msg](#)
- [wc_ed25519ph_verify_msg](#)
- [wc_ed25519_sign_msg](#)

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さとは一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };
byte hash[] = { メッセージの SHA-512ハッシュで初期化 };
byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed25519ph_verify_hash(sig, sizeof(sig), hash, sizeof(hash),
    &verified, &key, context, sizeof(context));
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}
```

C.23.2.10 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
)
```

この関数は、真正性を保証するためにメッセージの Ed25519 署名を検証します。コンテキストは検証されるデータの一部として含まれます。メッセージは検証前に事前ハッシュ化されます。答えは ret を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **msg** 検証するメッセージを含むバッファへのポインタ。
- **msgLen** 検証するメッセージの長さ。
- **ret** 検証結果へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するために使用する公開 Ed25519 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- [wc_ed25519_verify_msg](#)
- [wc_ed25519ctx_verify_msg](#)
- [wc_ed25519ph_verify_hash](#)
- [wc_ed25519_sign_msg](#)

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さとは一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };
byte msg[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed25519ph_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
    &verified, &key, context, sizeof(context));
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}
```

C.23.2.11 function wc_ed25519_init

```
int wc_ed25519_init(
    ed25519_key * key
)
```

この関数は、メッセージ検証での将来の使用のために ed25519_key オブジェクトを初期化します。

Parameters:

- **key** 初期化する ed25519_key オブジェクトへのポインタ。

See:

- [wc_ed25519_make_key](#)
- [wc_ed25519_free](#)

Return:

- 0 ed25519_key オブジェクトの初期化に成功した場合に返されます。
- BAD_FUNC_ARG key が NULL の場合に返されます。

Example

```
ed25519_key key;
wc_ed25519_init(&key);
```

C.23.2.12 function wc_ed25519_free


```
void wc_ed25519_free(  
    ed25519_key * key  
)
```

この関数は、使用後に Ed25519 オブジェクトを解放します。

Parameters:

- **key** 解放する ed25519_key オブジェクトへのポインタ

See: [wc_ed25519_init](#)

Example

```
ed25519_key key;  
// key を初期化し、安全な交換を実行  
...  
wc_ed25519_free(&key);
```

C.23.2.13 function wc_ed25519_import_public

```
int wc_ed25519_import_public(  
    const byte * in,  
    word32 inLen,  
    ed25519_key * key  
)
```

この関数は、公開鍵を含むバッファから公開 ed25519_key をインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は、秘密鍵が存在する場合にそれと一致するかチェックされます。

Parameters:

- **in** 公開鍵を含むバッファへのポインタ。
- **inLen** 公開鍵を含むバッファの長さ。
- **key** 公開鍵を格納する ed25519_key オブジェクトへのポインタ。

See:

- [wc_ed25519_import_public_ex](#)
- [wc_ed25519_import_private_key](#)
- [wc_ed25519_import_private_key_ex](#)
- [wc_ed25519_export_public](#)

Return:

- 0 ed25519_key のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または inLen が Ed25519 鍵のサイズより小さい場合に返されます。

Example

```
int ret;  
byte pub[] = { Ed25519 公開鍵で初期化 };  
  
ed_25519 key;  
wc_ed25519_init_key(&key);  
ret = wc_ed25519_import_public(pub, sizeof(pub), &key);  
if (ret != 0) {  
    // 鍵のインポートエラー  
}
```


C.23.2.14 function wc_ed25519_import_public_ex

```
int wc_ed25519_import_public_ex(
    const byte * in,
    word32 inLen,
    ed25519_key * key,
    int trusted
)
```

この関数は、公開鍵を含むバッファから公開 ed25519_key をインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、秘密鍵が存在するときに公開鍵が秘密鍵と一致するかチェックします。

Parameters:

- **in** 公開鍵を含むバッファへのポインタ。
- **inLen** 公開鍵を含むバッファの長さ。
- **key** 公開鍵を格納する ed25519_key オブジェクトへのポインタ。
- **trusted** 公開鍵データが信頼されているかどうか。

See:

- [wc_ed25519_import_public](#)
- [wc_ed25519_import_private_key](#)
- [wc_ed25519_import_private_key_ex](#)
- [wc_ed25519_export_public](#)

Return:

- 0 ed25519_key のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または inLen が Ed25519 鍵のサイズより小さい場合に返されます。

Example

```
int ret;
byte pub[] = { Ed25519 公開鍵で初期化 };

ed_25519 key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

C.23.2.15 function wc_ed25519_import_private_only

```
int wc_ed25519_import_private_only(
    const byte * priv,
    word32 privSz,
    ed25519_key * key
)
```

この関数は、バッファから Ed25519 秘密鍵のみをインポートします。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **key** インポートされた秘密鍵を格納する ed25519_key オブジェクトへのポインタ。

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 Ed25519 鍵のインポートに成功した場合に返されます。
- `BAD_FUNC_ARG` `priv` または `key` が `NULL` と評価された場合、または `privSz` が `ED25519_KEY_SIZE` と等しくない場合に返されます。

Example

```
int ret;
byte priv[] = { 32バイトの秘密鍵で初期化 };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // 秘密鍵のインポートエラー
}
```

C.23.2.16 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
)
```

この関数は、一对のバッファから公開/秘密 Ed25519 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は信頼されていないと想定され、秘密鍵に対してチェックされません。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **pub** 公開鍵を含むバッファへのポインタ。
- **pubSz** 公開鍵の長さ。
- **key** インポートされた秘密/公開鍵ペアを格納する `ed25519_key` オブジェクトへのポインタ。

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 `ed25519_key` のインポートに成功した場合に返されます。

- BAD_FUNC_ARG priv または key が NULL と評価された場合、または privSz が ED25519_KEY_SIZE にも ED25519_PRV_KEY_SIZE にも等しくない、もしくは pubSz が ED25519_PUB_KEY_SIZE より小さい場合に返されます。

Example

```
int ret;
byte priv[] = { 32バイトの秘密鍵で初期化 };
byte pub[] = { 対応する公開鍵で初期化 };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
    &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

C.23.2.17 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
)
```

この関数は、一対のバッファから公開/秘密 Ed25519 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、公開鍵が秘密鍵に対してチェックされます。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **pub** 公開鍵を含むバッファへのポインタ。
- **pubSz** 公開鍵の長さ。
- **key** インポートされた秘密/公開鍵ペアを格納する ed25519_key オブジェクトへのポインタ。
- **trusted** 公開鍵データが信頼されているかどうか。

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 ed25519_key のインポートに成功した場合に返されます。
- BAD_FUNC_ARG priv または key が NULL と評価された場合、または privSz が ED25519_KEY_SIZE にも ED25519_PRV_KEY_SIZE にも等しくない、もしくは pubSz が ED25519_PUB_KEY_SIZE より小さい場合に返されます。

Example

```
int ret;
byte priv[] = { 32バイトの秘密鍵で初期化 };
```

```

byte pub[] = { 対応する公開鍵で初期化 };

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) {
    // 鍵のインポートエラー
}

```

C.23.2.18 function wc_ed25519_export_public

```

int wc_ed25519_export_public(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)

```

この関数は、ed25519_key 構造体から公開鍵をエクスポートします。公開鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 公開鍵をエクスポートする ed25519_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。公開鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed25519_import_public](#)
- [wc_ed25519_import_public_ex](#)
- [wc_ed25519_export_private](#)
- [wc_ed25519_export_private_only](#)

Return:

- 0 公開鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが公開鍵を格納するのに十分な大きさでない場合に返されます。このエラーを返す際、関数は outLen に必要なサイズを設定します。

Example

```

int ret;
ed25519_key key;
// key を初期化し、key を作成

char pub[32];
word32 pubSz = sizeof(pub);

ret = wc_ed25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // 公開鍵のエクスポートエラー
}

```

C.23.2.19 function wc_ed25519_export_private_only

```
int wc_ed25519_export_private_only(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed25519_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 秘密鍵をエクスポートする ed25519_key 構造体へのポインタ。
- **out** 秘密鍵を格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。秘密鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed25519_export_public](#)
- [wc_ed25519_export_private](#)
- [wc_ed25519_import_private_key](#)
- [wc_ed25519_import_private_key_ex](#)

Return:

- 0 秘密鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが秘密鍵を格納するのに十分な大きさでない場合に返されます。

Example

```
int ret;
ed25519_key key;
// key を初期化し、key を作成

char priv[32]; // 秘密鍵のみなので 32 バイト
word32 privSz = sizeof(priv);
ret = wc_ed25519_export_private_only(&key, priv, &privSz);
if (ret != 0) {
    // 秘密鍵のエクスポートエラー
}
```

C.23.2.20 function wc_ed25519_export_private

```
int wc_ed25519_export_private(
    const ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed25519_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 鍵ペアをエクスポートする ed25519_key 構造体へのポインタ。
- **out** 鍵ペアを格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。鍵ペアのエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- `wc_ed25519_import_private_key`
- `wc_ed25519_import_private_key_ex`
- `wc_ed25519_export_private_only`

Return:

- 0 鍵ペアのエクスポートに成功した場合に返されます。
- `BAD_FUNC_ARG` 入力値のいずれかが `NULL` と評価された場合に返されます。
- `BUFFER_E` 提供されたバッファが鍵ペアを格納するのに十分な大きさでない場合に返されます。

Example

```
ed25519_key key;
wc_ed25519_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // 32 バイトの Ed25519 鍵を初期化

byte out[64]; // out は十分なバッファサイズである必要があります
word32 outLen = sizeof(out);
int key_size = wc_ed25519_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // 関数が outLen をリセットしたかどうかを確認するため、out のサイズと outLen を比較
}
```

C.23.2.21 function wc_ed25519_export_key

```
int wc_ed25519_export_key(
    const ed25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

この関数は、`ed25519_key` 構造体から秘密鍵と公開鍵を個別にエクスポートします。秘密鍵をバッファ `priv` に格納し、このバッファに書き込まれたバイト数を `privSz` に設定します。公開鍵をバッファ `pub` に格納し、このバッファに書き込まれたバイト数を `pubSz` に設定します。

Parameters:

- **key** 鍵ペアをエクスポートする `ed25519_key` 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** `out` で利用可能なサイズを持つ `word32` オブジェクトへのポインタ。秘密鍵のエクスポートに成功した後、`out` に書き込まれたバイト数が設定されます。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** `out` で利用可能なサイズを持つ `word32` オブジェクトへのポインタ。公開鍵のエクスポートに成功した後、`out` に書き込まれたバイト数が設定されます。

See:

- `wc_ed25519_export_private`
- `wc_ed25519_export_public`

Return:

- 0 鍵ペアのエクスポートに成功した場合に返されます。
- `BAD_FUNC_ARG` 入力値のいずれかが `NULL` と評価された場合に返されます。

- BUFFER_E 提供されたバッファが鍵ペアを格納するのに十分な大きさでない場合に返されます。

Example

```
int ret;
ed25519_key key;
// key を初期化し、key を作成

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) {
    // 公開鍵のエクスポートエラー
}
```

C.23.2.22 function wc_ed25519_check_key

```
int wc_ed25519_check_key(
    ed25519_key * key
)
```

この関数は、ed25519_key 構造体内の公開鍵が秘密鍵と一致するかチェックします。

Parameters:

- **key** 秘密鍵と公開鍵を保持する ed25519_key 構造体へのポインタ。

See:

- `wc_ed25519_import_private_key`
- `wc_ed25519_import_private_key_ex`

Return:

- 0 秘密鍵と公開鍵が一致した場合に返されます。
- BAD_FUNC_ARG 指定された key が NULL の場合に返されます。
- PUBLIC_KEY_E 公開鍵が利用できない、または無効な場合に返されます。

Example

```
int ret;
byte priv[] = { 57バイトの秘密鍵で初期化 };
byte pub[] = { 対応する公開鍵で初期化 };

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) {
    // 鍵のチェックエラー
}
```

C.23.2.23 function wc_ed25519_size

```
int wc_ed25519_size(  
    const ed25519_key * key  
)
```

この関数は、Ed25519 のサイズ（32 バイト）を返します。

Parameters:

- **key** 鍵サイズを取得する ed25519_key 構造体へのポインタ。

See: [wc_ed25519_make_key](#)

Return:

- ED25519_KEY_SIZE 有効な秘密鍵のサイズ（32 バイト）。
- BAD_FUNC_ARG 指定された key が NULL の場合に返されます。

Example

```
int keySz;  
ed25519_key key;  
// key を初期化し、key を作成  
keySz = wc_ed25519_size(&key);  
if (keySz == 0) {  
    // 鍵サイズの決定エラー  
}
```

C.23.2.24 function wc_ed25519_priv_size

```
int wc_ed25519_priv_size(  
    const ed25519_key * key  
)
```

この関数は、秘密鍵のサイズ（秘密 + 公開）をバイト単位で返します。

Parameters:

- **key** 鍵サイズを取得する ed25519_key 構造体へのポインタ。

See: [wc_ed25519_pub_size](#)

Return:

- ED25519_PRIV_KEY_SIZE 秘密鍵のサイズ（64 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```
ed25519_key key;  
wc_ed25519_init(&key);  
  
WC_RNG rng;  
wc_InitRng(&rng);  
  
wc_ed25519_make_key(&rng, 32, &key); // 32 バイトの Ed25519 鍵を初期化  
int key_size = wc_ed25519_priv_size(&key);
```

C.23.2.25 function wc_ed25519_pub_size

```
int wc_ed25519_pub_size(  
    const ed25519_key * key  
)
```


この関数は、圧縮鍵のサイズをバイト単位で返します（公開鍵）。

Parameters:

- **key** 鍵サイズを取得する ed25519_key 構造体へのポインタ。

See: `wc_ed25519_priv_size`

Return:

- ED25519_PUB_KEY_SIZE 圧縮公開鍵のサイズ（32 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```
ed25519_key key;
wc_ed25519_init(&key);
WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // 32 バイトの Ed25519 鍵を初期化
int key_size = wc_ed25519_pub_size(&key);
```

C.23.2.26 function wc_ed25519_sig_size

```
int wc_ed25519_sig_size(
    const ed25519_key * key
)
```

この関数は、Ed25519 署名のサイズ（バイト単位で 64）を返します。

Parameters:

- **key** 署名サイズを取得する ed25519_key 構造体へのポインタ。

See: `wc_ed25519_sign_msg`

Return:

- ED25519_SIG_SIZE Ed25519 署名のサイズ（64 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```
int sigSz;
ed25519_key key;
// key を初期化し、key を作成

sigSz = wc_ed25519_sig_size(&key);
if (sigSz == 0) {
    // 署名サイズの決定エラー
}
```

C.23.3 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);

```

C.24 dox_comments/header_files-ja/ed448.h

C.24.1 Functions

	Name
int	wc_ed448_make_public (ed448_key * key, unsigned char * pubKey, word32 pubKeySz) この関数は、秘密鍵から Ed448 公開鍵を生成します。公開鍵をバッファ pubKey に格納し、このバッファに書き込まれたバイト数を pubKeySz に設定します。
int	wc_ed448_make_key (WC_RNG * rng, int keysize, ed448_key * key) この関数は新しい Ed448 キーを生成し、それを key に格納します。
int	wc_ed448_sign_msg (const byte * in, word32 inLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed448_key オブジェクトを使用してメッセージに署名します。
int	wc_ed448ph_sign_hash (const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed448_key オブジェクトを使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。
int	wc_ed448ph_sign_msg (const byte * in, word32 inlen, byte * out, word32 * outlen, ed448_key * key, const byte * context, byte contextLen) この関数は、真正性を保証するために ed448_key オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されます。

	Name
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) この関数は、真正性を保証するためにメッセージの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。答えは res を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
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) この関数は、真正性を保証するためにメッセージのダイジェストの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。答えは res を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
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) この関数は、真正性を保証するためにメッセージの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。メッセージは検証前に事前ハッシュ化されます。答えは res を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。
int	wc_ed448_init (ed448_key * key) この関数は、メッセージ検証での将来の使用のために ed448_key オブジェクトを初期化します。
void	wc_ed448_free (ed448_key * key) この関数は、使用後に Ed448 オブジェクトを解放します。
int	wc_ed448_import_public (const byte * in, word32 inLen, ed448_key * key) この関数は、公開鍵を含むバッファから公開 ed448_key ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は、秘密鍵が存在する場合にそれと一致するかチェックされます。
int	wc_ed448_import_public_ex (const byte * in, word32 inLen, ed448_key * key, int trusted) この関数は、公開鍵を含むバッファから公開 ed448_key ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、秘密鍵が存在するときに公開鍵が秘密鍵と一致するかチェックします。
int	wc_ed448_import_private_only (const byte * priv, word32 privSz, ed448_key * key) この関数は、バッファから Ed448 秘密鍵のみをインポートします。

	Name
int	wc_ed448_import_private_key (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key) この関数は、一対のバッファから公開/秘密 Ed448 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。
int	wc_ed448_import_private_key_ex (const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key, int trusted) この関数は、一対のバッファから公開/秘密 Ed448 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、公開鍵が秘密鍵に対してチェックされます。
int	wc_ed448_export_public (const ed448_key * key, byte * out, word32 * outLen) この関数は、ed448_key 構造体から秘密鍵をエクスポートします。公開鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ed448_export_private_only (const ed448_key * key, byte * out, word32 * outLen) この関数は、ed448_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ed448_export_private (const ed448_key * key, byte * out, word32 * outLen) この関数は、ed448_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。
int	wc_ed448_export_key (const ed448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、ed448_key 構造体から秘密鍵と公開鍵を個別にエクスポートします。秘密鍵をバッファ priv に格納し、このバッファに書き込まれたバイト数を privSz に設定します。公開鍵をバッファ pub に格納し、このバッファに書き込まれたバイト数を pubSz に設定します。
int	wc_ed448_check_key (ed448_key * key) この関数は、ed448_key 構造体内の公開鍵が秘密鍵と一致するかチェックします。
int	wc_ed448_size (const ed448_key * key) この関数は、Ed448 秘密鍵のサイズ (57 バイト) を返します。
int	wc_ed448_priv_size (const ed448_key * key) この関数は、秘密鍵のサイズ (秘密 + 公開) をバイト単位で返します。
int	wc_ed448_pub_size (const ed448_key * key) この関数は、圧縮鍵のサイズをバイト単位で返します (公開鍵)。

	Name
int	wc_ed448_sig_size (const ed448_key * key) この関数は、Ed448 署名のサイズ (バイト単位で 114) を返します。

C.24.2 Functions Documentation

C.24.2.1 function wc_ed448_make_public

```
int wc_ed448_make_public(
    ed448_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

この関数は、秘密鍵から Ed448 公開鍵を生成します。公開鍵をバッファ pubKey に格納し、このバッファに書き込まれたバイト数を pubKeySz に設定します。

Parameters:

- **key** キーを生成する ed448_key へのポインタ。
- **pubKey** 公開鍵を格納するバッファへのポインタ。
- **pubKeySz** pubKey バッファのサイズ (バイト単位)。

See:

- **wc_ed448_init**
- **wc_ed448_import_private_only**
- **wc_ed448_make_key**

Return:

- 0 公開鍵の作成に成功した場合に返されます。
- BAD_FUNC_ARG key または pubKey が NULL と評価された場合、または指定されたキーサイズが 57 バイトでない場合に返されます (Ed448 は 57 バイトのキーを持ちます)。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
int ret;

ed448_key key;
byte priv[] = { 57バイトの秘密鍵で初期化 };
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) {
    // 公開鍵の作成エラー
}
```

C.24.2.2 function wc_ed448_make_key

```
int wc_ed448_make_key(
    WC_RNG * rng,
    int keysize,
```

```
    ed448_key * key
)
```

この関数は新しい Ed448 キーを生成し、それを key に格納します。

Parameters:

- **rng** キーを生成するために使用する初期化済み RNG オブジェクトへのポインタ。
- **keysize** 生成するキーの長さ。Ed448 の場合は常に 57 である必要があります。
- **key** キーを生成する ed448_key へのポインタ。

See: [wc_ed448_init](#)

Return:

- 0 ed448_key の作成に成功した場合に返されます。
- BAD_FUNC_ARG rng または key が NULL と評価された場合、または指定されたキーサイズが 57 バイトでない場合に返されます (Ed448 は 57 バイトのキーを持ちます)。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

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) {
    // キー作成エラー
}
```

C.24.2.3 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
)
```

この関数は、真正性を保証するために ed448_key オブジェクトを使用してメッセージに署名します。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inLen** 署名するメッセージの長さ。
- **out** 生成された署名を格納するバッファ。
- **outLen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 ed448_key へのポインタ。
- **context** メッセージが署名されるコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- `wc_ed448ph_sign_hash`
- `wc_ed448ph_sign_msg`
- `wc_ed448_verify_msg`

Return:

- 0 メッセージの署名の生成に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // 生成された署名を保持
sigSz = sizeof(sig);
byte message[] = { メッセージで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed448_init(&key); // key を初期化
wc_ed448_make_key(&rng, 57, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed448_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0) {
    // メッセージ署名の生成エラー
}
```

C.24.2.4 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
)
```

この関数は、真正性を保証するために ed448_key オブジェクトを使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。

Parameters:

- **hash** 署名するメッセージのハッシュを含むバッファへのポインタ。
- **hashLen** 署名するメッセージのハッシュの長さ。
- **out** 生成された署名を格納するバッファ。
- **outLen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 ed448_key へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- `wc_ed448_sign_msg`

- `wc_ed448ph_sign_msg`
- `wc_ed448ph_verify_hash`

Return:

- 0 メッセージダイジェストの署名の生成に成功した場合に返されます。
- `BAD_FUNC_ARG` 入力パラメータのいずれかが `NULL` と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- `MEMORY_E` 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // 生成された署名を保持
sigSz = sizeof(sig);
byte hash[] = { メッセージのハッシュで初期化 };
byte context[] = { 署名のコンテキストで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed448_init(&key); // key を初期化
wc_ed448_make_key(&rng, 57, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed448ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
    context, sizeof(context));
if (ret != 0) {
    // メッセージ署名の生成エラー
}
```

C.24.2.5 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
)
```

この関数は、真正性を保証するために `ed448_key` オブジェクトを使用してメッセージに署名します。コンテキストは署名されるデータの一部として含まれます。メッセージは署名計算前に事前ハッシュ化されます。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inLen** 署名するメッセージの長さ。
- **out** 生成された署名を格納するバッファ。
- **outLen** 出力バッファの最大長。メッセージ署名の生成に成功した後、out に書き込まれたバイト数が格納されます。
- **key** 署名を生成するために使用する秘密 `ed448_key` へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- `wc_ed448_sign_msg`
- `wc_ed448ph_sign_hash`
- `wc_ed448ph_verify_msg`

Return:

- 0 メッセージの署名の生成に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または出力バッファが生成された署名を格納するには小さすぎる場合に返されます。
- MEMORY_E 関数実行中にメモリの割り当てエラーが発生した場合に返されます。

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // 生成された署名を保持
sigSz = sizeof(sig);
byte message[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };

wc_InitRng(&rng); // rng を初期化
wc_ed448_init(&key); // key を初期化
wc_ed448_make_key(&rng, 57, &key); // 公開/秘密鍵ペアを作成
ret = wc_ed448ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
                           context, sizeof(context));
if (ret != 0) {
    // メッセージ署名の生成エラー
}
```

C.24.2.6 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
)
```

この関数は、真正性を保証するためにメッセージの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。答えは `res` を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **msg** 検証するメッセージを含むバッファへのポインタ。
- **msgLen** 検証するメッセージの長さ。
- **res** 検証完了後、有効な署名の場合は 1、無効な署名の場合は 0 が設定される int 型へのポインタ。
- **key** 署名を検証するために使用する公開 Ed448 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- `wc_ed448ph_verify_hash`
- `wc_ed448ph_verify_msg`
- `wc_ed448_sign_msg`

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さとは一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };
byte msg[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed448_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
    &key, context, sizeof(context));
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}
```

C.24.2.7 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
)
```

この関数は、真正性を保証するためにメッセージのダイジェストの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。ハッシュは署名計算前の事前ハッシュ化されたメッセージです。答えは res を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **hash** 検証するメッセージのハッシュを含むバッファへのポインタ。
- **hashlen** 検証するハッシュの長さ。
- **res** 検証完了後、有効な署名の場合は 1、無効な署名の場合は 0 が設定される int 型へのポインタ。
- **key** 署名を検証するために使用する公開 Ed448 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- `wc_ed448_verify_msg`
- `wc_ed448ph_verify_msg`
- `wc_ed448ph_sign_hash`

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さとは一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };
byte hash[] = { メッセージのハッシュで初期化 };
byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed448ph_verify_hash(sig, sizeof(sig), hash, sizeof(hash),
                             &verified, &key, context, sizeof(context));
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}
```

C.24.2.8 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
)
```

この関数は、真正性を保証するためにメッセージの Ed448 署名を検証します。コンテキストは検証されるデータの一部として含まれます。メッセージは検証前に事前ハッシュ化されます。答えは res を通じて返され、1 は有効な署名に対応し、0 は無効な署名に対応します。

Parameters:

- **sig** 検証する署名を含むバッファへのポインタ。
- **siglen** 検証する署名の長さ。
- **msg** 検証するメッセージを含むバッファへのポインタ。
- **msgLen** 検証するメッセージの長さ。
- **res** 検証完了後、有効な署名の場合は 1、無効な署名の場合は 0 が設定される int 型へのポインタ。
- **key** 署名を検証するために使用する公開 Ed448 鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファの長さ。

See:

- `wc_ed448_verify_msg`

- `wc_ed448ph_verify_hash`
- `wc_ed448ph_sign_msg`

Return:

- 0 署名の検証と認証の実行に成功した場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL と評価された場合、または siglen が署名の実際の長さとは一致しない場合に返されます。
- SIG_VERIFY_E 検証は完了したが、生成された署名が提供された署名と一致しない場合に返されます。

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { 受信した署名で初期化 };
byte msg[] = { メッセージで初期化 };
byte context[] = { 署名のコンテキストで初期化 };
// 受信した公開鍵で key を初期化
ret = wc_ed448ph_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
                           &key, context, sizeof(context));
if (ret < 0) {
    // 検証実行エラー
} else if (verified == 0)
    // 署名が無効
}
```

C.24.2.9 function wc_ed448_init

```
int wc_ed448_init(
    ed448_key * key
)
```

この関数は、メッセージ検証での将来の使用のために ed448_key オブジェクトを初期化します。

Parameters:

- **key** 初期化する ed448_key オブジェクトへのポインタ。

See:

- `wc_ed448_make_key`
- `wc_ed448_free`

Return:

- 0 ed448_key オブジェクトの初期化に成功した場合に返されます。
- BAD_FUNC_ARG key が NULL の場合に返されます。

Example

```
ed448_key key;
wc_ed448_init(&key);
```

C.24.2.10 function wc_ed448_free

```
void wc_ed448_free(
    ed448_key * key
)
```

この関数は、使用後に Ed448 オブジェクトを解放します。

Parameters:

- **key** 解放する ed448_key オブジェクトへのポインタ

See: [wc_ed448_init](#)

Example

```
ed448_key key;
// key を初期化し、安全な交換を実行
...
wc_ed448_free(&key);
```

C.24.2.11 function wc_ed448_import_public

```
int wc_ed448_import_public(
    const byte * in,
    word32 inLen,
    ed448_key * key
)
```

この関数は、公開鍵を含むバッファから公開 ed448_key ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。公開鍵は、秘密鍵が存在する場合にそれと一致するかチェックされます。

Parameters:

- **in** 公開鍵を含むバッファへのポインタ。
- **inLen** 公開鍵を含むバッファの長さ。
- **key** 公開鍵を格納する ed448_key オブジェクトへのポインタ。

See:

- [wc_ed448_import_public_ex](#)
- [wc_ed448_import_private_key](#)
- [wc_ed448_import_private_key_ex](#)
- [wc_ed448_export_public](#)

Return:

- 0 ed448_key のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または inLen が Ed448 鍵のサイズより小さい場合に返されます。

Example

```
int ret;
byte pub[] = { Ed448 公開鍵で初期化 };

ed_448 key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
    // 鍵のインポートエラー
}
```

C.24.2.12 function wc_ed448_import_public_ex

```
int wc_ed448_import_public_ex(
    const byte * in,
```

```

    word32 inLen,
    ed448_key * key,
    int trusted
)

```

この関数は、公開鍵を含むバッファから公開 ed448_key ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、秘密鍵が存在するときに公開鍵が秘密鍵と一致するかチェックします。

Parameters:

- **in** 公開鍵を含むバッファへのポインタ。
- **inLen** 公開鍵を含むバッファの長さ。
- **key** 公開鍵を格納する ed448_key オブジェクトへのポインタ。
- **trusted** 公開鍵データが信頼されているかどうか。

See:

- `wc_ed448_import_public`
- `wc_ed448_import_private_key`
- `wc_ed448_import_private_key_ex`
- `wc_ed448_export_public`

Return:

- 0 ed448_key のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または inLen が Ed448 鍵のサイズより小さい場合に返されます。

Example

```

int ret;
byte pub[] = { Ed448 公開鍵で初期化 };

ed_448 key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
    // 鍵のインポートエラー
}

```

C.24.2.13 function wc_ed448_import_private_only

```

int wc_ed448_import_private_only(
    const byte * priv,
    word32 privSz,
    ed448_key * key
)

```

この関数は、バッファから Ed448 秘密鍵のみをインポートします。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **key** インポートされた秘密鍵を格納する ed448_key オブジェクトへのポインタ。

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 Ed448 秘密鍵のインポートに成功した場合に返されます。
- `BAD_FUNC_ARG` in または `key` が `NULL` と評価された場合、または `privSz` が `ED448_KEY_SIZE` より小さい場合に返されます。

Example

```
int ret;
byte priv[] = { 57バイトの秘密鍵で初期化 };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // 秘密鍵のインポートエラー
}
```

C.24.2.14 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
)
```

この関数は、一対のバッファから公開/秘密 Ed448 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **pub** 公開鍵を含むバッファへのポインタ。
- **pubSz** 公開鍵の長さ。
- **key** インポートされた秘密/公開鍵ペアを格納する `ed448_key` オブジェクトへのポインタ。

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 Ed448 鍵のインポートに成功した場合に返されます。
- `BAD_FUNC_ARG` in または `key` が `NULL` と評価された場合、または `privSz` が `ED448_KEY_SIZE` より小さい、もしくは `pubSz` が `ED448_PUB_KEY_SIZE` より小さい場合に返されます。

Example

```
int ret;
byte priv[] = { 57バイトの秘密鍵で初期化 };
```



```

byte pub[] = { 対応する公開鍵で初期化 };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
    &key);
if (ret != 0) {
    // 鍵のインポートエラー
}

```

C.24.2.15 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
)

```

この関数は、一対のバッファから公開/秘密 Ed448 鍵ペアをインポートします。この関数は、圧縮キーと非圧縮キーの両方を処理します。信頼されていない場合、公開鍵が秘密鍵に対してチェックされます。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵の長さ。
- **pub** 公開鍵を含むバッファへのポインタ。
- **pubSz** 公開鍵の長さ。
- **key** インポートされた秘密/公開鍵ペアを格納する ed448_key オブジェクトへのポインタ。
- **trusted** 公開鍵データが信頼されているかどうか。

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 Ed448 鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が NULL と評価された場合、または privSz が ED448_KEY_SIZE より小さい、もしくは pubSz が ED448_PUB_KEY_SIZE より小さい場合に返されます。

Example

```

int ret;
byte priv[] = { 57バイトの秘密鍵で初期化 };
byte pub[] = { 対応する公開鍵で初期化 };

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) {

```

```
// 鍵のインポートエラー
}
```

C.24.2.16 function wc_ed448_export_public

```
int wc_ed448_export_public(
    const ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed448_key 構造体から秘密鍵をエクスポートします。公開鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 公開鍵をエクスポートする ed448_key 構造体へのポインタ。
- **out** 公開鍵を格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。公開鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed448_import_public](#)
- [wc_ed448_import_public_ex](#)
- [wc_ed448_export_private_only](#)

Return:

- 0 公開鍵のエクスポートに成功した場合に返されます。
- BAD_FUNC_ARG 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが秘密鍵を格納するのに十分な大きさでない場合に返されます。このエラーを返す際、関数は outLen に必要なサイズを設定します。

Example

```
int ret;
ed448_key key;
// key を初期化し、key を作成

char pub[57];
word32 pubSz = sizeof(pub);

ret = wc_ed448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // 公開鍵のエクスポートエラー
}
```

C.24.2.17 function wc_ed448_export_private_only

```
int wc_ed448_export_private_only(
    const ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed448_key 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 秘密鍵をエクスポートする ed448_key 構造体へのポインタ。
- **out** 秘密鍵を格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。秘密鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed448_export_public](#)
- [wc_ed448_import_private_key](#)
- [wc_ed448_import_private_key_ex](#)

Return:

- 0 秘密鍵のエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが秘密鍵を格納するのに十分な大きさでない場合に返されます。

Example

```
int ret;
ed448_key key;
// key を初期化し、key を作成

char priv[57]; // 秘密鍵のみなので 57 バイト
word32 privSz = sizeof(priv);
ret = wc_ed448_export_private_only(&key, priv, &privSz);
if (ret != 0) {
    // 秘密鍵のエクスポートエラー
}
```

C.24.2.18 function wc_ed448_export_private

```
int wc_ed448_export_private(
    const ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed448_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、このバッファに書き込まれたバイト数を outLen に設定します。

Parameters:

- **key** 鍵ペアをエクスポートする ed448_key 構造体へのポインタ。
- **out** 鍵ペアを格納するバッファへのポインタ。
- **outLen** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。鍵ペアのエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed448_import_private](#)
- [wc_ed448_export_private_only](#)

Return:

- 0 鍵ペアのエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが鍵ペアを格納するのに十分な大きさでない場合に返されます。

Example

```

ed448_key key;
wc_ed448_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // 57 バイトの Ed448 鍵を初期化

byte out[114]; // out は十分なバッファサイズである必要があります
word32 outLen = sizeof(out);
int key_size = wc_ed448_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // 関数が outLen をリセットしたかどうかを確認するため、out のサイズと outLen を比較
}

```

C.24.2.19 function wc_ed448_export_key

```

int wc_ed448_export_key(
    const ed448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)

```

この関数は、ed448_key 構造体から秘密鍵と公開鍵を個別にエクスポートします。秘密鍵をバッファ priv に格納し、このバッファに書き込まれたバイト数を privSz に設定します。公開鍵をバッファ pub に格納し、このバッファに書き込まれたバイト数を pubSz に設定します。

Parameters:

- **key** 鍵ペアをエクスポートする ed448_key 構造体へのポインタ。
- **priv** 秘密鍵を格納するバッファへのポインタ。
- **privSz** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。秘密鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。
- **pub** 公開鍵を格納するバッファへのポインタ。
- **pubSz** out で利用可能なサイズを持つ word32 オブジェクトへのポインタ。公開鍵のエクスポートに成功した後、out に書き込まれたバイト数が設定されます。

See:

- [wc_ed448_export_private](#)
- [wc_ed448_export_public](#)

Return:

- 0 鍵ペアのエクスポートに成功した場合に返されます。
- ECC_BAD_ARG_E 入力値のいずれかが NULL と評価された場合に返されます。
- BUFFER_E 提供されたバッファが鍵ペアを格納するのに十分な大きさでない場合に返されます。

Example

```

int ret;
ed448_key key;
// key を初期化し、key を作成

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) {
    // 秘密鍵と公開鍵のエクスポートエラー
}
```

C.24.2.20 function wc_ed448_check_key

```
int wc_ed448_check_key(
    ed448_key * key
)
```

この関数は、ed448_key 構造体内の公開鍵が秘密鍵と一致するかチェックします。

Parameters:

- **key** 秘密鍵と公開鍵を保持する ed448_key 構造体へのポインタ。

See:

- [wc_ed448_import_private_key](#)
- [wc_ed448_import_private_key_ex](#)

Return:

- 0 秘密鍵と公開鍵が一致した場合に返されます。
- BAD_FUNC_ARGS 指定された key が NULL の場合に返されます。

Example

```
int ret;
byte priv[] = { 57バイトの秘密鍵で初期化 };
byte pub[] = { 対応する公開鍵で初期化 };

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) {
    // 鍵のチェックエラー
}
```

C.24.2.21 function wc_ed448_size

```
int wc_ed448_size(
    const ed448_key * key
)
```

この関数は、Ed448 秘密鍵のサイズ（57 バイト）を返します。

Parameters:

- **key** 鍵サイズを取得する ed448_key 構造体へのポインタ。

See: [wc_ed448_make_key](#)

Return:

- ED448_KEY_SIZE 有効な秘密鍵のサイズ（57 バイト）。
- BAD_FUNC_ARGS 指定された key が NULL の場合に返されます。

Example

```
int keySz;
ed448_key key;
// key を初期化し、key を作成
keySz = wc_ed448_size(&key);
if (keySz == 0) {
    // 鍵サイズの決定エラー
}
```

C.24.2.22 function wc_ed448_priv_size

```
int wc_ed448_priv_size(
    const ed448_key * key
)
```

この関数は、秘密鍵のサイズ（秘密 + 公開）をバイト単位で返します。

Parameters:

- **key** 鍵サイズを取得する ed448_key 構造体へのポインタ。

See: [wc_ed448_pub_size](#)

Return:

- ED448_PRV_KEY_SIZE 秘密鍵のサイズ（114 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```
ed448_key key;
wc_ed448_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // 57 バイトの Ed448 鍵を初期化
int key_size = wc_ed448_priv_size(&key);
```

C.24.2.23 function wc_ed448_pub_size

```
int wc_ed448_pub_size(
    const ed448_key * key
)
```

この関数は、圧縮鍵のサイズをバイト単位で返します（公開鍵）。

Parameters:

- **key** 鍵サイズを取得する ed448_key 構造体へのポインタ。

See: [wc_ed448_priv_size](#)

Return:

- ED448_PUB_KEY_SIZE 圧縮公開鍵のサイズ（57 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```

ed448_key key;
wc_ed448_init(&key);
WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // 57 バイトの Ed448 鍵を初期化
int key_size = wc_ed448_pub_size(&key);

```

C.24.2.24 function wc_ed448_sig_size

```

int wc_ed448_sig_size(
    const ed448_key * key
)

```

この関数は、Ed448 署名のサイズ（バイト単位で 114）を返します。

Parameters:

- **key** 署名サイズを取得する ed448_key 構造体へのポインタ。

See: `wc_ed448_sign_msg`

Return:

- ED448_SIG_SIZE Ed448 署名のサイズ（114 バイト）。
- BAD_FUNC_ARG key 引数が NULL の場合に返されます。

Example

```

int sigSz;
ed448_key key;
// key を初期化し、key を作成

sigSz = wc_ed448_sig_size(&key);
if (sigSz == 0) {
    // 署名サイズの決定エラー
}

```

C.24.3 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);

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.25 dox_comments/header_files-ja/error-crypt.h

C.25.1 Functions

	Name
void	wc_ErrorString (int err, char * buff) この関数は、特定のエラーコードに対するエラー文字列を指定されたバッファに格納します。
const char *	wc_GetErrorString (int error) この関数は、特定のエラーコードに対するエラー文字列を返します。

C.25.2 Functions Documentation

C.25.2.1 function wc_ErrorString

```
void wc_ErrorString(
    int err,
    char * buff
)
```

この関数は、特定のエラーコードに対するエラー文字列を指定されたバッファに格納します。

Parameters:

- **error** 文字列を取得するエラーコード
- **buffer** エラー文字列を格納するバッファ。バッファは少なくとも WOLFSSL_MAX_ERROR_SZ(80 バイト) の長さである必要があります

See: [wc_GetErrorString](#)

Return: none 戻り値なし。

Example

```
char errorMsg[WOLFSSL_MAX_ERROR_SZ];
int err = wc_some_function();

if( err != 0) { // エラーが発生
    wc_ErrorString(err, errorMsg);
}
```

C.25.2.2 function wc_GetErrorString

```
const char * wc_GetErrorString(
    int error
)
```

この関数は、特定のエラーコードに対するエラー文字列を返します。

Parameters:

- **error** 文字列を取得するエラーコード

See: [wc_ErrorString](#)

Return: string エラーコードに対するエラー文字列を文字列リテラルとして返します。

Example

```
char * errorMsg;
int err = wc_some_function();

if( err != 0) { // エラーが発生
    errorMsg = wc_GetErrorString(err);
}
```

C.25.3 Source code

```
void wc_ErrorString(int err, char* buff);

const char* wc_GetErrorString(int error);
```

C.26 dox_comments/header_files-ja/evp.h

C.26.1 Functions

	Name
const WOLFSSL_EVP_CIPHER *	wolfSSL_EVP_des_ede3_ecb (void) それぞれの WOLFSSL_EVP_CIPHER ポインタのゲッター関数。これらの暗号文字列を設定するには、まずプログラムで wolfSSL_EVP_init() を一度呼び出す必要があります。wolfSSL_EVP_des_ede3_ecb() の場合、WOLFSSL_DES_ECB マクロが定義されている必要があります。
const WOLFSSL_EVP_CIPHER *	wolfSSL_EVP_des_cbc (void) それぞれの WOLFSSL_EVP_CIPHER ポインタのゲッター関数。これらの暗号文字列を設定するには、まずプログラムで wolfSSL_EVP_init() を一度呼び出す必要があります。wolfSSL_EVP_des_ecb() の場合、WOLFSSL_DES_ECB マクロが定義されている必要があります。
int	wolfSSL_EVP_DigestInit_ex (WOLFSSL_EVP_MD_CTX * ctx, const WOLFSSL_EVP_MD * type, WOLFSSL_ENGINE * impl) WOLFSSL_EVP_MD_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_DigestInit() のラッパーです。
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) WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_CipherInit() のラッパーです。

	Name
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) WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_CipherInit() のラッパーです。暗号化フラグを暗号化に設定します。
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) WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_CipherInit() のラッパーです。暗号化フラグを復号に設定します。
int	wolfSSL_EVP_CipherUpdate (WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl, const unsigned char * in, int inl) データを暗号化/復号する関数。in バッファが暗号化または復号される対象として追加され、out バッファが結果を保持します。outl は暗号化/復号された情報の長さになります。
int	wolfSSL_EVP_CipherFinal (WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl) この関数は、パディングを追加する最終暗号操作を実行します。WOLFSSL_EVP_CIPHER_CTX 構造体に WOLFSSL_EVP_CIPH_NO_PADDING フラグが設定されている場合、1 が返され、暗号化/復号は実行されません。パディングフラグが設定されている場合、ctx が暗号化に設定されているときにパディングが追加され暗号化され、復号に設定されているときにパディング値がチェックされます。
int	wolfSSL_EVP_CIPHER_CTX_set_key_length (WOLFSSL_EVP_CIPHER_CTX * ctx, int keylen) WOLFSSL_EVP_CIPHER_CTX 構造体の鍵長のセッター関数。
int	wolfSSL_EVP_CIPHER_CTX_block_size (const WOLFSSL_EVP_CIPHER_CTX * ctx) これは ctx ブロックサイズのゲッター関数です。
int	wolfSSL_EVP_CIPHER_block_size (const WOLFSSL_EVP_CIPHER * cipher) これは暗号のブロックサイズのゲッター関数です。
void	wolfSSL_EVP_CIPHER_CTX_set_flags (WOLFSSL_EVP_CIPHER_CTX * ctx, int flags) WOLFSSL_EVP_CIPHER_CTX 構造体のセッター関数。
void	wolfSSL_EVP_CIPHER_CTX_clear_flags (WOLFSSL_EVP_CIPHER_CTX * ctx, int flags) WOLFSSL_EVP_CIPHER_CTX 構造体のクリア関数。

	Name
int	wolfSSL_EVP_CIPHER_CTX_set_padding (WOLFSSL_EVP_CIPHER_CTX * c, int pad) パディングを使用するための WOLFSSL_EVP_CIPHER_CTX 構造体のセッター関数。
unsigned long	wolfSSL_EVP_CIPHER_CTX_flags (const WOLFSSL_EVP_CIPHER_CTX * ctx) WOLFSSL_EVP_CIPHER_CTX 構造体のゲッター関数。v1.1.0 で非推奨

C.26.2 Functions Documentation

C.26.2.1 function wolfSSL_EVP_des_ede3_ecb

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_ede3_ecb(
    void
)
```

それぞれの WOLFSSL_EVP_CIPHER ポインタのゲッター関数。これらの暗号文字列を設定するには、まずプログラムで wolfSSL_EVP_init() を一度呼び出す必要があります。wolfSSL_EVP_des_ede3_ecb() の場合、WOLFSSL_DES_ECB マクロが定義されている必要があります。

Parameters:

- **none** パラメータなし。

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer DES EDE3 操作の WOLFSSL_EVP_CIPHER ポインタを返します。

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.26.2.2 function wolfSSL_EVP_des_cbc

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_cbc(
    void
)
```

それぞれの WOLFSSL_EVP_CIPHER ポインタのゲッター関数。これらの暗号文字列を設定するには、まずプログラムで wolfSSL_EVP_init() を一度呼び出す必要があります。wolfSSL_EVP_des_ecb() の場合、WOLFSSL_DES_ECB マクロが定義されている必要があります。

Parameters:

- **none** パラメータなし。

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer DES 操作の WOLFSSL_EVP_CIPHER ポインタを返します。

Example

```
WOLFSSL_EVP_CIPHER* cipher;
cipher = wolfSSL_EVP_des_cbc();
...
```

C.26.2.3 function wolfSSL_EVP_DigestInit_ex

```
int wolfSSL_EVP_DigestInit_ex(
    WOLFSSL_EVP_MD_CTX * ctx,
    const WOLFSSL_EVP_MD * type,
    WOLFSSL_ENGINE * impl
)
```

WOLFSSL_EVP_MD_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_DigestInit() のラッパーです。

Parameters:

- **ctx** 初期化する構造体。
- **type** 実行するハッシュのタイプ、例えば SHA。
- **impl** 使用するエンジン。wolfSSL では該当なし、NULL でも可。

See:

- wolfSSL_EVP_MD_CTX_new
- [wolfCrypt_Init](#)
- wolfSSL_EVP_MD_CTX_free

Return:

- SSL_SUCCESS 正常に設定された場合。
- SSL_FAILURE 成功しなかった場合。

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));
//リソースを解放
```

C.26.2.4 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
)
```

WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_CipherInit() のラッパーです。

Parameters:

- **ctx** 初期化する構造体。
- **type** 実行する暗号化/復号のタイプ、例えば AES。
- **impl** 使用するエンジン。wolfSSL では該当なし、NULL でも可。
- **key** 設定する鍵。

- **iv** アルゴリズムが必要とする場合の iv。
- **enc** 暗号化 (1) または復号 (0) フラグ。

See:

- wolfSSL_EVP_CIPHER_CTX_new
- **wolfCrypt_Init**
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定された場合。
- SSL_FAILURE 成功しなかった場合。

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_cbc(), e, key, iv, 1));
// リソースを解放
```

C.26.2.5 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
)
```

WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_CipherInit() のラッパーです。暗号化フラグを暗号化に設定します。

Parameters:

- **ctx** 初期化する構造体。
- **type** 実行する暗号化のタイプ、例えば AES。
- **impl** 使用するエンジン。wolfSSL では該当なし、NULL でも可。
- **key** 使用する鍵。
- **iv** 使用する iv。

See:

- wolfSSL_EVP_CIPHER_CTX_new
- **wolfCrypt_Init**
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定された場合。
- SSL_FAILURE 成功しなかった場合。

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));
//リソースを解放
```

C.26.2.6 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
)
```

WOLFSSL_EVP_CIPHER_CTX を初期化する関数。この関数は、wolfSSL が WOLFSSL_ENGINE を使用しないため、wolfSSL_EVP_CipherInit() のラッパーです。暗号化フラグを復号に設定します。

Parameters:

- **ctx** 初期化する構造体。
- **type** 実行する暗号化/復号のタイプ、例えば AES。
- **impl** 使用するエンジン。wolfSSL では該当なし、NULL でも可。
- **key** 設定する鍵。
- **iv** アルゴリズムが必要とする場合の iv。
- **enc** 暗号化 (1) または復号 (0) フラグ。

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定された場合。
- SSL_FAILURE 成功しなかった場合。

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_cbc(), e, key, iv, 1));
// リソースを解放

```

C.26.2.7 function wolfSSL_EVP_CipherUpdate

```

int wolfSSL_EVP_CipherUpdate(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl,
    const unsigned char * in,
    int inl
)

```

データを暗号化/復号する関数。in バッファが暗号化または復号される対象として追加され、out バッファが結果を保持します。outl は暗号化/復号された情報の長さになります。

Parameters:

- **ctx** 暗号タイプを取得する構造体。
- **out** 出力を保持するバッファ。
- **outl** 出力のサイズに調整されます。
- **in** 操作を実行するバッファ。
- **inl** 入力バッファの長さ。

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 成功した場合。
- SSL_FAILURE 成功しなかった場合。

Example

```

WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
unsigned char out[100];
int outl;
unsigned char in[100];
int inl = 100;

ctx = wolfSSL_EVP_CIPHER_CTX_new();
// ctx をセットアップ
ret = wolfSSL_EVP_CipherUpdate(ctx, out, outl, in, inl);
// ret 値をチェック
// バッファ out は outl バイトのデータを保持
// リソースを解放

```


C.26.2.8 function wolfSSL_EVP_CipherFinal

```
int wolfSSL_EVP_CipherFinal(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * out1
)
```

この関数は、パディングを追加する最終暗号操作を実行します。WOLFSSL_EVP_CIPHER_CTX 構造体に WOLFSSL_EVP_CIPH_NO_PADDING フラグが設定されている場合、1 が返され、暗号化/復号は実行されません。パディングフラグが設定されている場合、ctx が暗号化に設定されているときにパディングが追加され暗号化され、復号に設定されているときにパディング値がチェックされます。

Parameters:

- **ctx** 復号/暗号化に使用する構造体。
- **out** 最終復号/暗号化用のバッファ。
- **out1** 関数によってデータが追加されたときの out バッファのサイズ。

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- 1 成功時に返されます。
- 0 失敗が発生した場合。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int out1;
unsigned char out[64];
// ctx を作成
wolfSSL_EVP_CipherFinal(ctx, out, &out1);
```

C.26.2.9 function wolfSSL_EVP_CIPHER_CTX_set_key_length

```
int wolfSSL_EVP_CIPHER_CTX_set_key_length(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int keylen
)
```

WOLFSSL_EVP_CIPHER_CTX 構造体の鍵長のセッター関数。

Parameters:

- **ctx** 鍵長を設定する構造体。
- **keylen** 鍵の長さ。

See: wolfSSL_EVP_CIPHER_flags

Return:

- SSL_SUCCESS 正常に設定された場合。
- SSL_FAILURE 鍵長の設定に失敗した場合。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int keylen;
// ctx を作成
wolfSSL_EVP_CIPHER_CTX_set_key_length(ctx, keylen);
```

C.26.2.10 function wolfSSL_EVP_CIPHER_CTX_block_size

```
int wolfSSL_EVP_CIPHER_CTX_block_size(  
    const WOLFSSL_EVP_CIPHER_CTX * ctx  
)
```

これは ctx ブロックサイズのゲッター関数です。

Parameters:

- **ctx** ブロックサイズを取得する暗号 ctx。

See: [wolfSSL_EVP_CIPHER_block_size](#)

Return: size ctx->block_size を返します。

Example

```
const WOLFSSL_EVP_CIPHER_CTX* ctx;  
//ctx をセットアップ  
printf("block size = %d\n", wolfSSL_EVP_CIPHER_CTX_block_size(ctx));
```

C.26.2.11 function wolfSSL_EVP_CIPHER_block_size

```
int wolfSSL_EVP_CIPHER_block_size(  
    const WOLFSSL_EVP_CIPHER * cipher  
)
```

これは暗号のブロックサイズのゲッター関数です。

Parameters:

- **cipher** ブロックサイズを取得する暗号。

See: [wolfSSL_EVP_aes_256_ctr](#)

Return: size ブロックサイズを返します。

Example

```
printf("block size = %d\n",  
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_aes_256_ecb()));
```

C.26.2.12 function wolfSSL_EVP_CIPHER_CTX_set_flags

```
void wolfSSL_EVP_CIPHER_CTX_set_flags(  
    WOLFSSL_EVP_CIPHER_CTX * ctx,  
    int flags  
)
```

WOLFSSL_EVP_CIPHER_CTX 構造体のセッター関数。

Parameters:

- **ctx** フラグを設定する構造体。
- **flag** 構造体に設定するフラグ。

See:

- [wolfSSL_EVP_CIPHER_flags](#)
- [wolfSSL_EVP_CIPHER_CTX_flags](#)

Return: none 戻り値なし。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;  
int flag;  
// ctx を作成  
wolfSSL_EVP_CIPHER_CTX_set_flags(ctx, flag);
```

C.26.2.13 function wolfSSL_EVP_CIPHER_CTX_clear_flags

```
void wolfSSL_EVP_CIPHER_CTX_clear_flags(  
    WOLFSSL_EVP_CIPHER_CTX * ctx,  
    int flags  
)
```

WOLFSSL_EVP_CIPHER_CTX 構造体のクリア関数。

Parameters:

- **ctx** フラグをクリアする構造体。
- **flag** 構造体でクリアするフラグ値。

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none 戻り値なし。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;  
int flag;  
// ctx を作成  
wolfSSL_EVP_CIPHER_CTX_clear_flags(ctx, flag);
```

C.26.2.14 function wolfSSL_EVP_CIPHER_CTX_set_padding

```
int wolfSSL_EVP_CIPHER_CTX_set_padding(  
    WOLFSSL_EVP_CIPHER_CTX * c,  
    int pad  
)
```

パディングを使用するための WOLFSSL_EVP_CIPHER_CTX 構造体のセッター関数。

Parameters:

- **ctx** パディングフラグを設定する構造体。
- **padding** パディングを設定しない場合は 0、パディングを設定する場合は 1。

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- SSL_SUCCESS 正常に設定された場合。
- BAD_FUNC_ARG null 引数が渡された場合。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;  
// ctx を作成  
wolfSSL_EVP_CIPHER_CTX_set_padding(ctx, 1);
```

C.26.2.15 function wolfSSL_EVP_CIPHER_CTX_flags

```
unsigned long wolfSSL_EVP_CIPHER_CTX_flags(  
    const WOLFSSL_EVP_CIPHER_CTX * ctx  
)
```

WOLFSSL_EVP_CIPHER_CTX 構造体のゲッター関数。v1.1.0 で非推奨

Parameters:

- **ctx** フラグを取得する構造体。

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfSSL_EVP_CIPHER_flags

Return: unsigned long フラグ/モードの unsigned long。

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;  
unsigned long flags;  
ctx = wolfSSL_EVP_CIPHER_CTX_new()  
flags = wolfSSL_EVP_CIPHER_CTX_flags(ctx);
```

C.26.3 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.27 dox_comments/header_files-ja/hash.h

C.27.1 Functions

	Name
int	wc_HashGetOID (enum wc_HashType hash_type) この関数は、提供された wc_HashType の OID を返します。
int	wc_HashGetDigestSize (enum wc_HashType hash_type) この関数は、hash_type のダイジェスト (出力) のサイズを返します。返されるサイズは、wc_Hash に提供される出力バッファが十分な大きさであることを確認するために使用されます。
int	wc_Hash (enum wc_HashType hash_type, const byte * data, word32 data_len, byte * hash, word32 hash_len) この関数は、提供されたデータバッファに対してハッシュを実行し、提供されたハッシュバッファに結果を返します。
int	wc_Md5Hash (const byte * data, word32 len, byte * hash) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。
int	wc_ShaHash (const byte * data, word32 len, byte * hash) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。
int	wc_Sha224Hash (const byte * data, word32 len, byte * hash) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。
int	wc_Sha256Hash (const byte * data, word32 len, byte * hash) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

	Name
int	wc_Sha384Hash (const byte * data, word32 len, byte * hash) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。
int	wc_Sha512Hash (const byte * data, word32 len, byte * hash) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。
int	wc_Sha3_224Hash (const byte * data, word32 len, byte * hash) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。
int	wc_Sha3_256Hash (const byte * data, word32 len, byte * hash) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。
int	wc_Sha3_384Hash (const byte * data, word32 len, byte * hash) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。
int	wc_Sha3_512Hash (const byte * data, word32 len, byte * hash) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。
int	wc_Shake128Hash (const byte * data, word32 len, byte * hash, word32 hashLen) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。
int	wc_Shake256Hash (const byte * data, word32 len, byte * hash, word32 hashLen) 便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

C.27.2 Functions Documentation

C.27.2.1 function wc_HashGetOID

```
int wc_HashGetOID(
    enum wc_HashType hash_type
)
```

この関数は、提供された wc_HashType の OID を返します。

Parameters:

- **hash_type** “WC_HASH_TYPE_SHA256” などの “enum wc_HashType” からのハッシュタイプ。

See:

- [wc_HashGetDigestSize](#)
- [wc_Hash](#)

Return:

- OID 0 より大きい値を返します
- HASH_TYPE_E ハッシュタイプがサポートされていません。
- BAD_FUNC_ARG 提供された引数の 1 つが正しくありません。

Example

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int oid = wc_HashGetOID(hash_type);
if (oid > 0) {
```

```
// 成功  
}
```

C.27.2.2 function wc_HashGetDigestSize

```
int wc_HashGetDigestSize(  
    enum wc_HashType hash_type  
)
```

この関数は、hash_type のダイジェスト (出力) のサイズを返します。返されるサイズは、wc_Hash に提供される出力バッファが十分な大きさであることを確認するために使用されます。

Parameters:

- **hash_type** “WC_HASH_TYPE_SHA256” などの “enum wc_HashType” からのハッシュタイプ。

See: [wc_Hash](#)

Return:

- Success 正の戻り値は、ハッシュのダイジェストサイズを示します。
- Error hash_type がサポートされていない場合、HASH_TYPE_E を返します。
- Failure 無効な hash_type が使用された場合、BAD_FUNC_ARG を返します。

Example

```
int hash_len = wc_HashGetDigestSize(hash_type);  
if (hash_len <= 0) {  
    WOLFSSL_MSG("Invalid hash type/len");  
    return BAD_FUNC_ARG;  
}
```

C.27.2.3 function wc_Hash

```
int wc_Hash(  
    enum wc_HashType hash_type,  
    const byte * data,  
    word32 data_len,  
    byte * hash,  
    word32 hash_len  
)
```

この関数は、提供されたデータバッファに対してハッシュを実行し、提供されたハッシュバッファに結果を返します。

Parameters:

- **hash_type** “WC_HASH_TYPE_SHA256” などの “enum wc_HashType” からのハッシュタイプ。
- **data** ハッシュ化するデータを含むバッファへのポインタ。
- **data_len** データバッファの長さ。
- **hash** 最終ハッシュを出力するために使用されるバッファへのポインタ。
- **hash_len** ハッシュバッファの長さ。

See: [wc_HashGetDigestSize](#)

Return: 0 成功、それ以外はエラー (BAD_FUNC_ARG や BUFFER_E など)。

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) {
        // 成功
    }
}

```

C.27.2.4 function wc_Md5Hash

```

int wc_Md5Hash(
    const byte * data,
    word32 len,
    byte * hash
)

```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Md5Hash](#)
- [wc_Md5Final](#)
- [wc_InitMd5](#)

Return:

- 0 データのハッシュ化に成功した場合に返されます。
- Memory_E メモリエラー、メモリを割り当てられません。これは小さいスタックオプションが有効な場合にのみ可能です。

Example

```

const byte* data;
word32 data_len;
byte* hash;
int ret;
...
ret = wc_Md5Hash(data, data_len, hash);
if (ret != 0) {
    // Md5 ハッシュ失敗のケース。
}

```

C.27.2.5 function wc_ShaHash

```

int wc_ShaHash(
    const byte * data,
    word32 len,
    byte * hash
)

```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ

- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_ShaHash](#)
- [wc_ShaFinal](#)
- [wc_InitSha](#)

Return:

- 0 正常に.... の場合に返されます。
- Memory_E メモリエラー、メモリを割り当てられません。これは小さいスタックオプションが有効な場合にのみ可能です。

Example

none

C.27.2.6 function wc_Sha224Hash

```
int wc_Sha224Hash(  
    const byte * data,  
    word32 len,  
    byte * hash  
)
```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_InitSha224](#)
- [wc_Sha224Update](#)
- [wc_Sha224Final](#)

Return:

- 0 成功
- <0 エラー

Example

none

C.27.2.7 function wc_Sha256Hash

```
int wc_Sha256Hash(  
    const byte * data,  
    word32 len,  
    byte * hash  
)
```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ

- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Sha256Hash](#)
- [wc_Sha256Final](#)
- [wc_InitSha256](#)

Return:

- 0 正常に...の場合に返されます
- Memory_E メモリエラー、メモリを割り当てられません。これは小さいスタックオプションが有効な場合にのみ可能です。

Example

none

C.27.2.8 function wc_Sha384Hash

```
int wc_Sha384Hash(  
    const byte * data,  
    word32 len,  
    byte * hash  
)
```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Sha384Hash](#)
- [wc_Sha384Final](#)
- [wc_InitSha384](#)

Return:

- 0 データのハッシュ化に成功した場合に返されます
- Memory_E メモリエラー、メモリを割り当てられません。これは小さいスタックオプションが有効な場合にのみ可能です。

Example

none

C.27.2.9 function wc_Sha512Hash

```
int wc_Sha512Hash(  
    const byte * data,  
    word32 len,  
    byte * hash  
)
```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ

- **len** データの長さ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Sha512Hash](#)
- [wc_Sha512Final](#)
- [wc_InitSha512](#)

Return:

- 0 入力されたデータのハッシュ化に成功した場合に返されます
- Memory_E メモリエラー、メモリを割り当てられません。これは小さいスタックオプションが有効な場合にのみ可能です。

Example

none

C.27.2.10 function wc_Sha3_224Hash

```
int wc_Sha3_224Hash(  
    const byte * data,  
    word32 len,  
    byte * hash  
)
```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_InitSha3_224](#)
- [wc_Sha3_224_Update](#)
- [wc_Sha3_224_Final](#)

Return:

- 0 データのハッシュ化に成功した場合に返されます
- Memory_E メモリエラー、メモリを割り当てられません。これは小さいスタックオプションが有効な場合にのみ可能です。

Example

none

C.27.2.11 function wc_Sha3_256Hash

```
int wc_Sha3_256Hash(  
    const byte * data,  
    word32 len,  
    byte * hash  
)
```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_InitSha3_256](#)
- [wc_Sha3_256_Update](#)
- [wc_Sha3_256_Final](#)

Return:

- 0 データのハッシュ化に成功した場合に返されます
- Memory_E メモリエラー、メモリを割り当てられません。これは小さいスタックオプションが有効な場合にのみ可能です。

Example

none

C.27.2.12 function wc_Sha3_384Hash

```
int wc_Sha3_384Hash(  
    const byte * data,  
    word32 len,  
    byte * hash  
)
```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_InitSha3_384](#)
- [wc_Sha3_384_Update](#)
- [wc_Sha3_384_Final](#)

Return:

- 0 データのハッシュ化に成功した場合に返されます
- Memory_E メモリエラー、メモリを割り当てられません。これは小さいスタックオプションが有効な場合にのみ可能です。

Example

none

C.27.2.13 function wc_Sha3_512Hash

```
int wc_Sha3_512Hash(  
    const byte * data,  
    word32 len,  
    byte * hash  
)
```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_InitSha3_512](#)
- [wc_Sha3_512_Update](#)
- [wc_Sha3_512_Final](#)

Return:

- 0 入力されたデータのハッシュ化に成功した場合に返されます
- Memory_E メモリエラー、メモリを割り当てられません。これは小さいスタックオプションが有効な場合にのみ可能です。

Example

none

C.27.2.14 function wc_Shake128Hash

```
int wc_Shake128Hash(  
    const byte * data,  
    word32 len,  
    byte * hash,  
    word32 hashLen  
)
```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ
- **hash** ハッシュ値を保持するバイト配列。
- **hashLen** hash に書き込むバイト数。

See:

- [wc_InitShake128](#)
- [wc_Shake128_Update](#)
- [wc_Shake128_Final](#)

Return:

- 0 入力されたデータのハッシュ化に成功した場合に返されます
- Memory_E メモリエラー、メモリを割り当てられません。これは小さいスタックオプションが有効な場合にのみ可能です。

Example

none

C.27.2.15 function wc_Shake256Hash

```
int wc_Shake256Hash(  
    const byte * data,  
    word32 len,  
    byte * hash,  
    word32 hashLen  
)
```

便利な関数で、すべてのハッシュ処理を行い、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ
- **hash** ハッシュ値を保持するバイト配列。
- **hashLen** hash に書き込むバイト数。

See:

- [wc_InitShake256](#)
- [wc_Shake256_Update](#)
- [wc_Shake256_Final](#)

Return:

- 0 入力されたデータのハッシュ化に成功した場合に返されます
- Memory_E メモリエラー、メモリを割り当てられません。これは小さいスタックオプションが有効な場合にのみ可能です。

Example

none

C.27.3 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_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.28 dox_comments/header_files-ja/hmac.h

C.28.1 Functions

	Name
int	wc_HmacSetKey (Hmac * hmac, int type, const byte * key, word32 keySz) この関数は Hmac オブジェクトを初期化し、暗号化タイプ、鍵、および HMAC 長を設定します。
int	wc_HmacUpdate (Hmac * hmac, const byte * in, word32 sz) この関数は、HMAC を使用して認証するメッセージを更新します。wc_HmacSetKey で Hmac オブジェクトが初期化された後に呼び出す必要があります。この関数は、ハッシュするメッセージを更新するために複数回呼び出すことができます。必要に応じて wc_HmacUpdate を呼び出した後、wc_HmacFinal を呼び出して最終的な認証メッセージタグを取得する必要があります。
int	wc_HmacFinal (Hmac * hmac, byte * out) この関数は、Hmac オブジェクトのメッセージの最終ハッシュを計算します。
int	wolfSSL_GetHmacMaxSize (void) この関数は、設定された暗号スイートに基づいて利用可能な最大の HMAC ダイジェストサイズを返します。
int	wc_HKDF (int type, const byte * inKey, word32 inKeySz, const byte * salt, word32 saltSz, const byte * info, word32 infoSz, byte * out, word32 outSz) この関数は、HMAC 鍵導出関数 (HKDF) へのアクセスを提供します。HMAC を利用して、オプションのソルトとオプションの情報を含む inKey を導出鍵に変換し、out に保存します。ハッシュタイプは、0 または NULL が指定された場合、デフォルトで MD5 になります。
int	wc_HKDF_Extract (int type, const byte * salt, word32 saltSz, const byte * inKey, word32 inKeySz, byte * out) この関数は、HMAC 鍵導出関数 (HKDF) へのアクセスを提供します。HMAC を利用して、オプションのソルトを含む inKey を導出鍵に変換し、out に保存します。ハッシュタイプは、0 または NULL が指定された場合、デフォルトで MD5 になります。

	Name
int	wc_HKDF_Extract_ex (int type, const byte * salt, word32 saltSz, const byte * inKey, word32 inKeySz, byte * out, void * heap, int devId) この関数は、HMAC 鍵導出関数 (HKDF) へのアクセスを提供します。HMAC を利用して、オプションのソルトを含む inKey を導出鍵に変換し、out に保存します。ハッシュタイプは、0 または NULL が指定された場合、デフォルトで MD5 になります。これは、ヒープヒントとデバイス識別子を追加する _ex バージョンです。
int	wc_HKDF_Expand (int type, const byte * inKey, word32 inKeySz, const byte * info, word32 infoSz, byte * out, word32 outSz) この関数は、HMAC 鍵導出関数 (HKDF) へのアクセスを提供します。HMAC を利用して、オプションの情報を含む inKey を導出鍵に変換し、out に保存します。ハッシュタイプは、0 または NULL が指定された場合、デフォルトで MD5 になります。
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) この関数は、HMAC 鍵導出関数 (HKDF) へのアクセスを提供します。HMAC を利用して、オプションの情報を含む inKey を導出鍵に変換し、out に保存します。ハッシュタイプは、0 または NULL が指定された場合、デフォルトで MD5 になります。これは、ヒープヒントとデバイス識別子を追加する _ex バージョンです。
int	wc_Tls13_HKDF_Extract (byte * prk, const byte * salt, word32 saltLen, byte * ikm, word32 ikmLen, int digest) この関数は、TLS v1.3 鍵導出のための RFC 5869 HMAC ベース抽出拡張鍵導出関数 (HKDF) へのアクセスを提供します。
int	wc_Tls13_HKDF_Extract_ex (byte * prk, const byte * salt, word32 saltLen, byte * ikm, word32 ikmLen, int digest, void * heap, int devId) この関数は、TLS v1.3 鍵導出のための RFC 5869 HMAC ベース抽出拡張鍵導出関数 (HKDF) へのアクセスを提供します。これは、ヒープヒントとデバイス識別子を追加する _ex バージョンです。
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) HMAC、ソルト、ラベル、および情報を使用してデータを拡張します。TLS v1.3 は鍵導出のためにこの関数を定義しています。これは、ヒープヒントとデバイス識別子を追加する _ex バージョンです。

	Name
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) HMAC、ソルト、ラベル、および情報を使用してデータを拡張します。TLS v1.3 は鍵導出のためにこの関数を定義しています。
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) この関数は wc_Tls13_HKDF_Expand_Label() と非常に似ていますが、通常使用されるスタックスペースが十分でない場合にメモリを割り当てます。HMAC、ソルト、ラベル、および情報を使用してデータを拡張します。TLS v1.3 は鍵導出のためにこの関数を定義しています。これは、ヒープヒントとデバイス識別子を追加する _ex バージョンです。

C.28.2 Functions Documentation

C.28.2.1 function wc_HmacSetKey

```
int wc_HmacSetKey(
    Hmac * hmac,
    int type,
    const byte * key,
    word32 keySz
)
```

この関数は Hmac オブジェクトを初期化し、暗号化タイプ、鍵、および HMAC 長を設定します。

Parameters:

- **hmac** 初期化する Hmac オブジェクトへのポインタ。
- **type** Hmac オブジェクトが使用する暗号化方式を指定するタイプ。有効なオプションは：WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512。
- **key** Hmac オブジェクトを初期化する鍵を含むバッファへのポインタ。
- **length** 鍵の長さ。

See:

- **wc_HmacUpdate**
- **wc_HmacFinal**

Return:

- 0 Hmac オブジェクトの初期化に成功した場合に返されます。
- BAD_FUNC_ARG 入力タイプが無効な場合に返されます (type パラメータを参照)。
- MEMORY_E ハッシュに使用する構造体のメモリ割り当てエラーがある場合に返されます。
- HMAC_MIN_KEYLEN_E FIPS 実装を使用していて、指定された鍵の長さが最小許容 FIPS 標準の 14 バイトより短い場合に返されます。

Example

```
Hmac hmac;
byte key[] = { // 暗号化に使用する鍵で初期化 };
if (wc_HmacSetKey(&hmac, WC_MD5, key, sizeof(key)) != 0) {
    // Hmac オブジェクトの初期化エラー
}
```

C.28.2.2 function wc_HmacUpdate

```
int wc_HmacUpdate(
    Hmac * hmac,
    const byte * in,
    word32 sz
)
```

この関数は、HMAC を使用して認証するメッセージを更新します。wc_HmacSetKey で Hmac オブジェクトが初期化された後に呼び出す必要があります。この関数は、ハッシュするメッセージを更新するために複数回呼び出すことができます。必要に応じて wc_HmacUpdate を呼び出した後、wc_HmacFinal を呼び出して最終的な認証メッセージタグを取得する必要があります。

Parameters:

- **hmac** メッセージを更新する Hmac オブジェクトへのポインタ。
- **msg** 追加するメッセージを含むバッファへのポインタ。
- **length** 追加するメッセージの長さ。

See:

- [wc_HmacSetKey](#)
- [wc_HmacFinal](#)

Return:

- 0 認証するメッセージの更新に成功した場合に返されます。
- MEMORY_E ハッシュアルゴリズムで使用するメモリ割り当てエラーがある場合に返されます。

Example

```
Hmac hmac;
byte msg[] = { // 認証するメッセージで初期化 };
byte msg2[] = { // メッセージの後半で初期化 };
// hmac を初期化
if ( wc_HmacUpdate(&hmac, msg, sizeof(msg)) != 0) {
    // メッセージ更新エラー
}
if ( wc_HmacUpdate(&hmac, msg2, sizeof(msg)) != 0) {
    // 2 番目のメッセージでの更新エラー
}
```

C.28.2.3 function wc_HmacFinal

```
int wc_HmacFinal(
    Hmac * hmac,
    byte * out
)
```

この関数は、Hmac オブジェクトのメッセージの最終ハッシュを計算します。

Parameters:

- **hmac** 最終ハッシュを計算する Hmac オブジェクトへのポインタ。
- **hash** 最終ハッシュを保存するバッファへのポインタ。選択したハッシュアルゴリズムに必要なスペースを確保する必要があります。

See:

- `wc_HmacSetKey`
- `wc_HmacUpdate`

Return:

- 0 最終ハッシュの計算に成功した場合に返されます。
- MEMORY_E ハッシュアルゴリズムで使用するメモリ割り当てエラーがある場合に返されます。

Example

```
Hmac hmac;
byte hash[MD5_DIGEST_SIZE];
// タイプとして MD5 で hmac を初期化
// メッセージで wc_HmacUpdate()

if (wc_HmacFinal(&hmac, hash) != 0) {
    // ハッシュ計算エラー
}
```

C.28.2.4 function wolfSSL_GetHmacMaxSize

```
int wolfSSL_GetHmacMaxSize(
    void
)
```

この関数は、設定された暗号スイートに基づいて利用可能な最大の HMAC ダイジェストサイズを返します。

Parameters:

- **none** パラメータなし。

See: none

Return: Success 設定された暗号スイートに基づいて利用可能な最大の HMAC ダイジェストサイズを返します。

Example

```
int maxDigestSz = wolfSSL_GetHmacMaxSize();
```

C.28.2.5 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
)
```

この関数は、HMAC 鍵導出関数（HKDF）へのアクセスを提供します。HMAC を利用して、オプションのソルトとオプションの情報を含む inKey を導出鍵に変換し、out に保存します。ハッシュタイプは、0 または NULL が指定された場合、デフォルトで MD5 になります。

Parameters:

- **type** HKDF に使用するハッシュタイプ。有効なタイプは：WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512。
- **inKey** KDF に使用する鍵を含むバッファへのポインタ。
- **inKeySz** 入力鍵の長さ。
- **salt** オプションのソルトを含むバッファへのポインタ。ソルトを使用しない場合は NULL を使用します。
- **saltSz** ソルトの長さ。ソルトを使用しない場合は 0 を使用します。
- **info** オプションの追加情報を含むバッファへのポインタ。追加情報を追加しない場合は NULL を使用します。
- **infoSz** 追加情報の長さ。追加情報を使用しない場合は 0 を使用します。
- **out** 導出鍵を保存するバッファへのポインタ。
- **outSz** 生成された鍵を保存する出力バッファで利用可能なスペース。

See: [wc_HmacSetKey](#)

Return:

- 0 指定された入力で鍵の生成に成功した場合に返されます。
- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合に返されます（type パラメータを参照）。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- HMAC_MIN_KEYLEN_E FIPS 実装を使用していて、指定された鍵の長さが最小許容 FIPS 標準より短い場合に返される可能性があります。

HMAC 設定オプションは-enable-hmac（デフォルトでオン）、またはソースを直接ビルドする場合は HAVE_HKDF です。

Example

```
byte key[] = { // 鍵で初期化 };
byte salt[] = { // ソルトで初期化 };
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 ) {
    // 導出鍵の生成エラー
}
```

C.28.2.6 function wc_HKDF_Extract

```
int wc_HKDF_Extract(
    int type,
    const byte * salt,
    word32 saltSz,
    const byte * inKey,
    word32 inKeySz,
    byte * out
)
```

この関数は、HMAC 鍵導出関数（HKDF）へのアクセスを提供します。HMAC を利用して、オプションのソルトを含む inKey を導出鍵に変換し、out に保存します。ハッシュタイプは、0 または NULL が指定された場合、デフォルトで MD5 になります。

Parameters:

- **type** HKDF に使用するハッシュタイプ。有効なタイプは：WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512。
- **salt** オプションのソルトを含むバッファへのポインタ。ソルトを使用しない場合は NULL を使用します。
- **saltSz** ソルトの長さ。ソルトを使用しない場合は 0 を使用します。
- **inKey** KDF に使用する鍵を含むバッファへのポインタ。
- **inKeySz** 入力鍵の長さ。
- **out** 導出鍵を保存するバッファへのポインタ。

See:

- [wc_HKDF](#)
- [wc_HKDF_Extract_ex](#)
- [wc_HKDF_Expand](#)
- [wc_HKDF_Expand_ex](#)

Return:

- 0 指定された入力で鍵の生成に成功した場合に返されます。
- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合に返されます (type パラメータを参照)。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- HMAC_MIN_KEYLEN_E FIPS 実装を使用していて、指定された鍵の長さが最小許容 FIPS 標準より短い場合に返される可能性があります。

HMAC 設定オプションは-enable-hmac (デフォルトでオン)、またはソースを直接ビルドする場合は HAVE_HKDF です。

Example

```
byte key[] = { // 鍵で初期化 };
byte salt[] = { // ソルトで初期化 };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Extract(WC_SHA512, salt, sizeof(salt), key, sizeof(key),
    derivedKey);
if ( ret != 0 ) {
    // 導出鍵の生成エラー
}
```

C.28.2.7 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
)
```

この関数は、HMAC 鍵導出関数 (HKDF) へのアクセスを提供します。HMAC を利用して、オプションのソルトを含む inKey を導出鍵に変換し、out に保存します。ハッシュタイプは、0 または NULL が指定された場合、デフォルトで MD5 になります。これは、ヒープヒントとデバイス識別子を追加する _ex バージョンです。

Parameters:

- **type** HKDF に使用するハッシュタイプ。有効なタイプは：WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512。
- **salt** オプションのソルトを含むバッファへのポインタ。ソルトを使用しない場合は NULL を使用します。
- **saltSz** ソルトの長さ。ソルトを使用しない場合は 0 を使用します。
- **inKey** KDF に使用する鍵を含むバッファへのポインタ。
- **inKeySz** 入力鍵の長さ。
- **out** 導出鍵を保存するバッファへのポインタ。
- **heap** メモリに使用するヒープヒント。NULL にできます。
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定します。

See:

- [wc_HKDF](#)
- [wc_HKDF_Extract](#)
- [wc_HKDF_Expand](#)
- [wc_HKDF_Expand_ex](#)

Return:

- 0 指定された入力で鍵の生成に成功した場合に返されます。
- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合に返されます (type パラメータを参照)。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- HMAC_MIN_KEYLEN_E FIPS 実装を使用していて、指定された鍵の長さが最小許容 FIPS 標準より短い場合に返される可能性があります。

HMAC 設定オプションは-enable-hmac (デフォルトでオン)、またはソースを直接ビルドする場合は HAVE_HKDF です。

Example

```
byte key[] = { // 鍵で初期化 };
byte salt[] = { // ソルトで初期化 };
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 ) {
    // 導出鍵の生成エラー
}
```

C.28.2.8 function wc_HKDF_Expand

```
int wc_HKDF_Expand(
    int type,
    const byte * inKey,
    word32 inKeySz,
    const byte * info,
    word32 infoSz,
    byte * out,
    word32 outSz
)
```

この関数は、HMAC 鍵導出関数 (HKDF) へのアクセスを提供します。HMAC を利用して、オプションの情報を含む inKey を導出鍵に変換し、out に保存します。ハッシュタイプは、0 または NULL が指定された場合、デフォルトで MD5 になります。

Parameters:

- **type** HKDF に使用するハッシュタイプ。有効なタイプは：WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512。
- **inKey** KDF に使用する鍵を含むバッファへのポインタ。
- **inKeySz** 入力鍵の長さ。
- **info** オプションの追加情報を含むバッファへのポインタ。追加情報を追加しない場合は NULL を使用します。
- **infoSz** 追加情報の長さ。追加情報を使用しない場合は 0 を使用します。
- **out** 導出鍵を保存するバッファへのポインタ。
- **outSz** 生成された鍵を保存する出力バッファで利用可能なスペース。

See:

- `wc_HKDF`
- `wc_HKDF_Extract`
- `wc_HKDF_Extract_ex`
- `wc_HKDF_Expand_ex`

Return:

- 0 指定された入力で鍵の生成に成功した場合に返されます。
- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合に返されます（type パラメータを参照）。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- HMAC_MIN_KEYLEN_E FIPS 実装を使用して、指定された鍵の長さが最小許容 FIPS 標準より短い場合に返される可能性があります。

HMAC 設定オプションは `-enable-hmac`（デフォルトでオン）、またはソースを直接ビルドする場合は `HAVE_HKDF` です。

Example

```
byte key[] = { // 鍵で初期化 };
byte salt[] = { // ソルトで初期化 };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Expand(WC_SHA512, key, sizeof(key), NULL, 0,
    derivedKey, sizeof(derivedKey));
if ( ret != 0 ) {
    // 導出鍵の生成エラー
}
```

C.28.2.9 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
)
```

この関数は、HMAC 鍵導出関数（HKDF）へのアクセスを提供します。HMAC を利用して、オプションの情報を含む inKey を導出鍵に変換し、out に保存します。ハッシュタイプは、0 または NULL が指定された場

合、デフォルトで MD5 になります。これは、ヒープヒントとデバイス識別子を追加する `_ex` バージョンです。

Parameters:

- **type** HKDF に使用するハッシュタイプ。有効なタイプは：WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512。
- **inKey** KDF に使用する鍵を含むバッファへのポインタ。
- **inKeySz** 入力鍵の長さ。
- **info** オプションの追加情報を含むバッファへのポインタ。追加情報を追加しない場合は NULL を使用します。
- **infoSz** 追加情報の長さ。追加情報を使用しない場合は 0 を使用します。
- **out** 導出鍵を保存するバッファへのポインタ。
- **outSz** 生成された鍵を保存する出力バッファで利用可能なスペース。
- **heap** メモリに使用するヒープヒント。NULL にできます。
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定します。

See:

- `wc_HKDF`
- `wc_HKDF_Extract`
- `wc_HKDF_Extract_ex`
- `wc_HKDF_Expand`

Return:

- 0 指定された入力で鍵の生成に成功した場合に返されます。
- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合に返されます (type パラメータを参照)。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- HMAC_MIN_KEYLEN_E FIPS 実装を使用していて、指定された鍵の長さが最小許容 FIPS 標準より短い場合に返される可能性があります。

HMAC 設定オプションは `-enable-hmac` (デフォルトでオン)、またはソースを直接ビルドする場合は `HAVE_HKDF` です。

Example

```
byte key[] = { // 鍵で初期化 };
byte salt[] = { // ソルトで初期化 };
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 ) {
    // 導出鍵の生成エラー
}
```

C.28.2.10 function wc_Tls13_HKDF_Extract

```
int wc_Tls13_HKDF_Extract(
    byte * prk,
    const byte * salt,
    word32 saltLen,
    byte * ikm,
    word32 ikmLen,
    int digest
)
```


この関数は、TLS v1.3 鍵導出のための RFC 5869 HMAC ベース抽出拡張鍵導出関数（HKDF）へのアクセスを提供します。

Parameters:

- **prk** 生成された疑似ランダム鍵。
- **salt** ソルト。
- **saltLen** ソルトの長さ。
- **ikm** 鍵材料の出力へのポインタ。
- **ikmLen** 入力鍵材料バッファの長さ。
- **digest** HKDF に使用するハッシュタイプ。有効なタイプは：WC_SHA256、WC_SHA384、または WC_SHA512。

See:

- [wc_HKDF](#)
- [wc_HKDF_Extract](#)
- [wc_HKDF_Extract_ex](#)
- [wc_HKDF_Expand](#)
- [wc_Tls13_HKDF_Extract_ex](#)

Return:

- 0 指定された入力で鍵の生成に成功した場合に返されます。
- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合に返されます（type パラメータを参照）。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- HMAC_MIN_KEYLEN_E FIPS 実装を使用していて、指定された鍵の長さが最小許容 FIPS 標準より短い場合に返される可能性があります。

Example

```
byte secret[] = { // ランダム鍵で初期化 };
byte salt[] = { // オプションのソルトで初期化 };
byte masterSecret[MAX_DIGEST_SIZE];

int ret = wc_Tls13_HKDF_Extract(secret, salt, sizeof(salt), 0,
    masterSecret, sizeof(masterSecret), WC_SHA512);
if ( ret != 0 ) {
    // 導出鍵の生成エラー
}
```

C.28.2.11 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
)
```

この関数は、TLS v1.3 鍵導出のための RFC 5869 HMAC ベース抽出拡張鍵導出関数（HKDF）へのアクセスを提供します。これは、ヒープヒントとデバイス識別子を追加する _ex バージョンです。

Parameters:

- **prk** 生成された疑似ランダム鍵。

- **salt** ソルト。
- **saltLen** ソルトの長さ。
- **ikm** 鍵材料の出力へのポインタ。
- **ikmLen** 入力鍵材料バッファの長さ。
- **digest** HKDF に使用するハッシュタイプ。有効なタイプは：WC_SHA256、WC_SHA384、または WC_SHA512。
- **heap** メモリに使用するヒープヒント。NULL にできます。
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定します。

See:

- [wc_HKDF](#)
- [wc_HKDF_Extract](#)
- [wc_HKDF_Extract_ex](#)
- [wc_HKDF_Expand](#)
- [wc_Tls13_HKDF_Extract](#)

Return:

- 0 指定された入力で鍵の生成に成功した場合に返されます。
- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合に返されます (type パラメータを参照)。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- HMAC_MIN_KEYLEN_E FIPS 実装を使用していて、指定された鍵の長さが最小許容 FIPS 標準より短い場合に返される可能性があります。

Example

```
byte secret[] = { // ランダム鍵で初期化 };
byte salt[] = { // オプションのソルトで初期化 };
byte masterSecret[MAX_DIGEST_SIZE];

int ret = wc_Tls13_HKDF_Extract_ex(secret, salt, sizeof(salt), 0,
    masterSecret, sizeof(masterSecret), WC_SHA512, NULL, INVALID_DEVID);
if ( ret != 0 ) {
    // 導出鍵の生成エラー
}
```

C.28.2.12 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
)
```

HMAC、ソルト、ラベル、および情報を使用してデータを拡張します。TLS v1.3 は鍵導出のためにこの関数を定義しています。これは、ヒープヒントとデバイス識別子を追加する _ex バージョンです。

Parameters:

- **okm** 生成された疑似ランダム鍵 - 出力鍵材料。
- **okmLen** 生成された疑似ランダム鍵の長さ - 出力鍵材料。
- **prk** ソルト - 疑似ランダム鍵。
- **prkLen** ソルトの長さ - 疑似ランダム鍵。
- **protocol** TLS プロトコルラベル。
- **protocolLen** TLS プロトコルラベルの長さ。
- **info** 拡張する情報。
- **infoLen** 情報の長さ。
- **digest** HKDF に使用するハッシュタイプ。有効なタイプは：WC_SHA256、WC_SHA384、または WC_SHA512。
- **heap** メモリに使用するヒープヒント。NULL にできます。
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定します。

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 指定された入力で鍵の生成に成功した場合に返されます。
- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合に返されます (type パラメータを参照)。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- HMAC_MIN_KEYLEN_E FIPS 実装を使用していて、指定された鍵の長さが最小許容 FIPS 標準より短い場合に返される可能性があります。

C.28.2.13 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
)
```

HMAC、ソルト、ラベル、および情報を使用してデータを拡張します。TLS v1.3 は鍵導出のためにこの関数を定義しています。

Parameters:

- **okm** 生成された疑似ランダム鍵 - 出力鍵材料。
- **okmLen** 生成された疑似ランダム鍵の長さ - 出力鍵材料。
- **prk** ソルト - 疑似ランダム鍵。
- **prkLen** ソルトの長さ - 疑似ランダム鍵。

- **protocol** TLS プロトコラベル。
- **protocolLen** TLS プロトコラベルの長さ。
- **info** 拡張する情報。
- **infoLen** 情報の長さ。
- **digest** HKDF に使用するハッシュタイプ。有効なタイプは：WC_SHA256、WC_SHA384、または 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 指定された入力で鍵の生成に成功した場合に返されます。
- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合に返されます (type パラメータを参照)。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- HMAC_MIN_KEYLEN_E FIPS 実装を使用していて、指定された鍵の長さが最小許容 FIPS 標準より短い場合に返される可能性があります。

C.28.2.14 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
)
```

この関数は `wc_Tls13_HKDF_Expand_Label()` と非常に似ていますが、通常使用されるスタックスペースが十分でない場合にメモリを割り当てます。HMAC、ソルト、ラベル、および情報を使用してデータを拡張します。TLS v1.3 は鍵導出のためにこの関数を定義しています。これは、ヒープヒントとデバイス識別子を追加する `_ex` バージョンです。

Parameters:

- **okm** 生成された疑似ランダム鍵 - 出力鍵材料。
- **okmLen** 生成された疑似ランダム鍵の長さ - 出力鍵材料。
- **prk** ソルト - 疑似ランダム鍵。
- **prkLen** ソルトの長さ - 疑似ランダム鍵。
- **protocol** TLS プロトコラベル。
- **protocolLen** TLS プロトコラベルの長さ。
- **info** 拡張する情報。
- **infoLen** 情報の長さ。
- **digest** HKDF に使用するハッシュタイプ。有効なタイプは：WC_SHA256、WC_SHA384、または WC_SHA512。

- **heap** メモリに使用するヒープヒント。NULL にできます。

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 指定された入力で鍵の生成に成功した場合に返されます。
- `BAD_FUNC_ARG` 無効なハッシュタイプが指定された場合に返されます (type パラメータを参照)。
- `MEMORY_E` メモリ割り当てエラーがある場合に返されます。
- `HMAC_MIN_KEYLEN_E` FIPS 実装を使用していて、指定された鍵の長さが最小許容 FIPS 標準より短い場合に返される可能性があります。

C.28.3 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.29 dox_comments/header_files-ja/iotsafe.h

C.29.1 Functions

	Name
int	wolfSSL_CTX_iotsafe_enable (WOLFSSL_CTX * ctx) この関数は、指定されたコンテキストで IoT-Safe サポートを有効にします。

	Name
int	wolfSSL_iotsafe_on (WOLFSSL * ssl, byte privkey_id, byte ecdh_keypair_slot, byte peer_pubkey_slot, byte peer_cert_slot) この関数は、IoT-Safe TLS コールバックを指定された SSL セッションに接続します。
int	**wolfSSL_iotsafe_on_ex と同等ですが、IoT-SAFE スロットの ID を参照渡しでき、ID フィールドの長さをパラメータ "id_size" で指定できる点が異なります。
void	wolfIoTSafe_SetCSIM_read_cb (wolfSSL_IOTSafe_CSIM_read_cb rf)AT+CSIM コマンド用の読み取りコールバックを関連付けます。この入力関数は通常、モデムと通信する UART チャンネルの読み取りイベントに関連付けられます。関連付けられる読み取りコールバックはグローバルで、同時に IoT-safe サポートを使用するすべてのコンテキストに対して変更されます。
void	wolfIoTSafe_SetCSIM_write_cb (wolfSSL_IOTSafe_CSIM_write_cb wf)AT+CSIM コマンド用の書き込みコールバックを関連付けます。この出力関数は通常、モデムと通信する UART チャンネルの書き込みイベントに関連付けられます。関連付けられる書き込みコールバックはグローバルで、同時に IoT-safe サポートを使用するすべてのコンテキストに対して変更されます。
int	wolfIoTSafe_GetRandom (unsigned char * out, word32 sz)IoT-Safe 関数 GetRandom を使用して、指定されたサイズのランダムバッファを生成します。この関数は wolfCrypt RNG オブジェクトによって自動的に使用されます。
int	wolfIoTSafe_GetCert (byte id, unsigned char * output, unsigned long sz)IoT-Safe アプレット上のファイルに保存されている証明書をインポートし、メモリ内にローカルに保存します。1 バイトのファイル ID フィールドで動作します。
int	**wolfIoTSafe_GetCert_ex と同等ですが、2 バイト以上のファイル ID で呼び出すことができます。
int	wc_iotsafe_ecc_import_public (ecc_key * key, byte key_id)IoT-Safe アプレットに保存されている ECC 256 ビット公開鍵を ecc_key オブジェクトにインポートします。
int	wc_iotsafe_ecc_export_public (ecc_key * key, byte key_id)ECC 256 ビット公開鍵を ecc_key オブジェクトから IoT-Safe アプレット内の書き込み可能な公開鍵スロットにエクスポートします。
int	**wc_iotsafe_ecc_import_public_ex と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。
int	wc_iotsafe_ecc_export_private (ecc_key * key, byte key_id)ECC 256 ビット鍵を ecc_key オブジェクトから IoT-Safe アプレット内の書き込み可能な秘密鍵スロットにエクスポートします。

	Name
int	** wc_iotsafe_ecc_export_private_ex と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。
int	wc_iotsafe_ecc_sign_hash (byte * in, word32 inlen, byte * out, word32 * outlen, byte key_id)IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた秘密鍵を使用して、事前計算されたハッシュに署名します。
int	** wc_iotsafe_ecc_sign_hash_ex と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。
int	wc_iotsafe_ecc_verify_hash (byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte key_id)IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた公開鍵を使用して、事前計算されたハッシュに対する ECC 署名を検証します。結果は res に書き込まれます。1 は有効、0 は無効です。注意: 有効性をテストするために戻り値を使用しないでください。res のみを使用してください。
int	** wc_iotsafe_ecc_verify_hash_ex と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。
int	wc_iotsafe_ecc_gen_k (byte key_id)ECC 256 ビット鍵ペアを生成し、IoT-Safe アプレット内の(書き込み可能な)スロットに保存します。

C.29.2 Functions Documentation

C.29.2.1 function wolfSSL_CTX_iotsafe_enable

```
int wolfSSL_CTX_iotsafe_enable(
    WOLFSSL_CTX * ctx
)
```

この関数は、指定されたコンテキストで IoT-Safe サポートを有効にします。

Parameters:

- **ctx** IoT-safe サポートを有効にする必要がある WOLFSSL_CTX オブジェクトへのポインタ

See:

- [wolfSSL_iotsafe_on](#)
- [wolfIoTSafe_SetCSIM_read_cb](#)
- [wolfIoTSafe_SetCSIM_write_cb](#)

Return:

- 0 成功時
- WC_HW_E ハードウェアエラー時

Example

```
WOLFSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
if (!ctx)
    return NULL;
wolfSSL_CTX_iotsafe_enable(ctx);
```


C.29.2.2 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
)
```

この関数は、IoT-Safe TLS コールバックを指定された SSL セッションに接続します。

Parameters:

- **ssl** コールバックが有効になる WOLFSSL オブジェクトへのポインタ
- **privkey_id** ホストの秘密鍵を含む iot-safe アプレットスロットの ID
- **ecdh_keypair_slot** ECDH 鍵ペアを格納する iot-safe アプレットスロットの ID
- **peer_pubkey_slot** ECDH 用に他のエンドポイントの公開鍵を格納する iot-safe アプレットスロットの ID
- **peer_cert_slot** 検証用に他のエンドポイントの公開鍵を格納する iot-safe アプレットスロットの ID

See:

- [wolfSSL_iotsafe_on_ex](#)
- [wolfSSL_CTX_iotsafe_enable](#)

Return:

- 0 成功時
- NOT_COMPILED_IN HAVE_PK_CALLBACKS が無効な場合
- BAD_FUNC_ARG ssl ポインタが無効な場合

これは、スロットの ID が 1 バイト長の場合に SSL セッションを IoT-Safe アプレットに接続するために呼び出す必要があります。IoT-SAFE スロットの ID が 2 バイト以上の場合、代わりに [wolfSSL_iotsafe_on_ex\(\)](#) を使用する必要があります。

Example

```
// IoT-Safe 用の鍵 ID を定義
#define PRIVKEY_ID 0x02
#define ECDH_KEYPAIR_ID 0x03
#define PEER_PUBKEY_ID 0x04
#define PEER_CERT_ID 0x05
// 新しい ssl セッションを作成
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// IoT-Safe を有効にして鍵スロットに関連付け
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.29.2.3 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
)

```

この関数は、IoT-Safe TLS コールバックを指定された SSL セッションに接続します。これは `wolfSSL_iotsafe_on` と同等ですが、IoT-SAFE スロットの ID を参照渡しでき、ID フィールドの長さをパラメータ `id_size` で指定できる点が異なります。

Parameters:

- **ssl** コールバックが有効になる WOLFSSL オブジェクトへのポインタ
- **privkey_id** ホストの秘密鍵を含む iot-safe アプレットスロットの ID へのポインタ
- **ecdh_keypair_slot** ECDH 鍵ペアを格納する iot-safe アプレットスロットの ID へのポインタ
- **peer_pubkey_slot** ECDH 用に他のエンドポイントの公開鍵を格納する iot-safe アプレットスロットの ID へのポインタ
- **peer_cert_slot** 検証用に他のエンドポイントの公開鍵を格納する iot-safe アプレットスロットの ID へのポインタ
- **id_size** 各スロット ID のサイズ

See:

- `wolfSSL_iotsafe_on`
- `wolfSSL_CTX_iotsafe_enable`

Return:

- 0 成功時
- NOT_COMPILED_IN HAVE_PK_CALLBACKS が無効な場合
- BAD_FUNC_ARG ssl ポインタが無効な場合

Example

```

// IoT-Safe 用の鍵 ID を定義 (16 ビット、リトルエンディアン)
#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;

```

```

// 新しい ssl セッションを作成
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// IoT-Safe を有効にして鍵スロットを関連付け
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.29.2.4 function wolfIoTSafe_SetCSIM_read_cb

```
void wolfIoTSafe_SetCSIM_read_cb(  
    wolfSSL_IOTSafe_CSIM_read_cb rf  
)
```

AT+CSIM コマンド用の読み取りコールバックを関連付けます。この入力関数は通常、モデムと通信する UART チャンネルの読み取りイベントに関連付けられます。関連付けられる読み取りコールバックはグローバルで、同時に IoT-safe サポートを使用するすべてのコンテキストに対して変更されます。

Parameters:

- **rf** UART 読み取りイベントに関連付けられる読み取りコールバック。コールバック関数は 2 つの引数 (buf, len) を受け取り、len まで読み取られた文字数を返します。改行文字に遭遇すると、コールバックはこれまでに受信した文字数 (改行文字を含む) を返す必要があります。

See: [wolfIoTSafe_SetCSIM_write_cb](#)

Example

```
// USART 読み取り関数、他の場所で定義  
int usart_read(char *buf, int len);  
  
wolfIoTSafe_SetCSIM_read_cb(usart_read);
```

C.29.2.5 function wolfIoTSafe_SetCSIM_write_cb

```
void wolfIoTSafe_SetCSIM_write_cb(  
    wolfSSL_IOTSafe_CSIM_write_cb wf  
)
```

AT+CSIM コマンド用の書き込みコールバックを関連付けます。この出力関数は通常、モデムと通信する UART チャンネルの書き込みイベントに関連付けられます。関連付けられる書き込みコールバックはグローバルで、同時に IoT-safe サポートを使用するすべてのコンテキストに対して変更されます。

Parameters:

- **rf** UART 書き込みイベントに関連付けられる書き込みコールバック。コールバック関数は 2 つの引数 (buf, len) を受け取り、len まで書き込まれた文字数を返します。

See: [wolfIoTSafe_SetCSIM_read_cb](#)

Example

```
// USART 書き込み関数、他の場所で定義  
int usart_write(const char *buf, int len);  
wolfIoTSafe_SetCSIM_write_cb(usart_write);
```

C.29.2.6 function wolfIoTSafe_GetRandom

```
int wolfIoTSafe_GetRandom(  
    unsigned char * out,  
    word32 sz  
)
```

IoT-Safe 関数 GetRandom を使用して、指定されたサイズのランダムバッファを生成します。この関数は wolfCrypt RNG オブジェクトによって自動的に使用されます。

Parameters:

- **out** ランダムなバイトシーケンスが格納されるバッファ。
- **sz** 生成するランダムシーケンスのサイズ (バイト単位)

Return: 0 成功時

C.29.2.7 function wolfIoTSafe_GetCert

```
int wolfIoTSafe_GetCert(
    byte id,
    unsigned char * output,
    unsigned long sz
)
```

IoT-Safe アプレット上のファイルに保存されている証明書をインポートし、メモリ内にローカルに保存します。1 バイトのファイル ID フィールドで動作します。

Parameters:

- **id** 証明書が保存されている IoT-Safe アプレット内のファイル ID
- **output** 証明書がインポートされるバッファ
- **sz** バッファ output で利用可能な最大サイズ

Return:

- インポートされた証明書の長さ
- < 0 失敗の場合

Example

```
#define CRT_CLIENT_FILE_ID 0x03
unsigned char cert_buffer[2048];
// 証明書をバッファに取得
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);

// TLS クライアントコンテキストのアイデンティティとして証明書バッファを使用
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.29.2.8 function wolfIoTSafe_GetCert_ex

```
int wolfIoTSafe_GetCert_ex(
    uint8_t * id,
    uint16_t id_sz,
    unsigned char * output,
    unsigned long sz
)
```

IoT-Safe アプレット上のファイルに保存されている証明書をインポートし、メモリ内にローカルに保存します。`wolfIoTSafe_GetCert`と同等ですが、2 バイト以上のファイル ID で呼び出すことができます。

Parameters:

- **id** 証明書が保存されている IoT-Safe アプレット内のファイル ID へのポインタ
- **id_sz** ファイル ID のサイズ
- **output** 証明書がインポートされるバッファ
- **sz** バッファ output で利用可能な最大サイズ

Return:

- インポートされた証明書の長さ
- < 0 失敗の場合

*Example***C.29.2.9 function wc_iotsafe_ecc_import_public**

```
int wc_iotsafe_ecc_import_public(  
    ecc_key * key,  
    byte key_id  
)
```

IoT-Safe アプレットに保存されている ECC 256 ビット公開鍵を ecc_key オブジェクトにインポートします。

Parameters:

- **key** IoT-Safe アプレットからインポートされた鍵を含む ecc_key オブジェクト
- **id** 公開鍵が保存されている IoT-Safe アプレット内の鍵 ID

See:

- [wc_iotsafe_ecc_export_public](#)
- [wc_iotsafe_ecc_export_private](#)

Return:

- 0 成功時
- < 0 失敗の場合

C.29.2.10 function wc_iotsafe_ecc_export_public

```
int wc_iotsafe_ecc_export_public(  
    ecc_key * key,  
    byte key_id  
)
```

ECC 256 ビット公開鍵を ecc_key オブジェクトから IoT-Safe アプレット内の書き込み可能な公開鍵スロットにエクスポートします。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 公開鍵が保存される IoT-Safe アプレット内の鍵 ID

See:

- [wc_iotsafe_ecc_import_public_ex](#)
- [wc_iotsafe_ecc_export_private](#)

Return:

- 0 成功時
- < 0 失敗の場合

C.29.2.11 function wc_iotsafe_ecc_import_public_ex

```
int wc_iotsafe_ecc_import_public_ex(  
    ecc_key * key,  
    byte * key_id,  
    word16 id_size  
)
```

ECC 256 ビット公開鍵を ecc_key オブジェクトから IoT-Safe アプレット内の書き込み可能な公開鍵スロットにエクスポートします。wc_iotsafe_ecc_import_public と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 公開鍵が保存される IoT-Safe アプレット内の鍵 ID へのポインタ
- **id_size** 鍵 ID のサイズ

See:

- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_private

Return:

- 0 成功時
- < 0 失敗の場合

C.29.2.12 function wc_iotsafe_ecc_export_private

```
int wc_iotsafe_ecc_export_private(  
    ecc_key * key,  
    byte key_id  
)
```

ECC 256 ビット鍵を ecc_key オブジェクトから IoT-Safe アプレット内の書き込み可能な秘密鍵スロットにエクスポートします。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 秘密鍵が保存される IoT-Safe アプレット内の鍵 ID

See:

- wc_iotsafe_ecc_export_private_ex
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return:

- 0 成功時
- < 0 失敗の場合

C.29.2.13 function wc_iotsafe_ecc_export_private_ex

```
int wc_iotsafe_ecc_export_private_ex(  
    ecc_key * key,  
    byte * key_id,  
    word16 id_size  
)
```

ECC 256 ビット鍵を ecc_key オブジェクトから IoT-Safe アプレット内の書き込み可能な秘密鍵スロットにエクスポートします。`wc_iotsafe_ecc_export_private`と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 秘密鍵が保存される IoT-Safe アプレット内の鍵 ID へのポインタ
- **id_size** 鍵 ID のサイズ

See:

- `wc_iotsafe_ecc_export_private`
- `wc_iotsafe_ecc_import_public`
- `wc_iotsafe_ecc_export_public`

Return:

- 0 成功時
- < 0 失敗の場合

C.29.2.14 function wc_iotsafe_ecc_sign_hash

```
int wc_iotsafe_ecc_sign_hash(  
    byte * in,  
    word32 inlen,  
    byte * out,  
    word32 * outlen,  
    byte key_id  
)
```

IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた秘密鍵を使用して、事前計算されたハッシュに署名します。

Parameters:

- **in** 署名するメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージハッシュの長さ
- **out** 生成された署名を格納するバッファ
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功すると、out に書き込まれたバイト数を格納します
- **id** ペイロードに署名する秘密鍵を含むスロットの IoT-Safe アプレット内の鍵 ID

See:

- `wc_iotsafe_ecc_sign_hash_ex`
- `wc_iotsafe_ecc_verify_hash`
- `wc_iotsafe_ecc_gen_k`

Return:

- 0 成功時
- < 0 失敗の場合

C.29.2.15 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
)

```

IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた秘密鍵を使用して、事前計算されたハッシュに署名します。`wc_iotsafe_ecc_sign_hash`と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。

Parameters:

- **in** 署名するメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージハッシュの長さ
- **out** 生成された署名を格納するバッファ
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功すると、out に書き込まれたバイト数を格納します
- **id** ペイロードに署名する秘密鍵を含むスロットの IoT-Safe アプレット内の鍵 ID へのポインタ
- **id_size** 鍵 ID のサイズ

See:

- `wc_iotsafe_ecc_sign_hash`
- `wc_iotsafe_ecc_verify_hash`
- `wc_iotsafe_ecc_gen_k`

Return:

- 0 成功時
- < 0 失敗の場合

C.29.2.16 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
)

```

IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた公開鍵を使用して、事前計算されたハッシュに対する ECC 署名を検証します。結果は res に書き込まれます。1 は有効、0 は無効です。注意: 有効性をテストするために戻り値を使用しないでください。res のみを使用してください。

Parameters:

- **sig** 検証する署名を含むバッファ
- **hash** 署名されたハッシュ (メッセージダイジェスト)
- **hashlen** ハッシュの長さ (オクテット)
- **res** 署名の結果、1== 有効、0== 無効
- **key_id** IoT-Safe アプレット内に公開 ECC 鍵が保存されているスロットの ID

See:

- `wc_iotsafe_ecc_verify_hash_ex`
- `wc_iotsafe_ecc_sign_hash`
- `wc_iotsafe_ecc_gen_k`

Return:

- 0 成功時 (署名が有効でない場合でも)

- < 0 失敗の場合。

C.29.2.17 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
)
```

IoT-Safe アプレット内に事前に保存または事前プロビジョニングされた公開鍵を使用して、事前計算されたハッシュに対する ECC 署名を検証します。結果は `res` に書き込まれます。1 は有効、0 は無効です。注意: 有効性をテストするために戻り値を使用しないでください。res のみを使用してください。`wc_iotsafe_ecc_verify_hash` と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。

Parameters:

- **sig** 検証する署名を含むバッファ
- **hash** 署名されたハッシュ (メッセージダイジェスト)
- **hashlen** ハッシュの長さ (オクテット)
- **res** 署名の結果、1== 有効、0== 無効
- **key_id** IoT-Safe アプレット内に公開 ECC 鍵が保存されているスロットの ID
- **id_size** 鍵 ID のサイズ

See:

- `wc_iotsafe_ecc_verify_hash`
- `wc_iotsafe_ecc_sign_hash`
- `wc_iotsafe_ecc_gen_k`

Return:

- 0 成功時 (署名が有効でない場合でも)
- < 0 失敗の場合。

C.29.2.18 function wc_iotsafe_ecc_gen_k

```
int wc_iotsafe_ecc_gen_k(
    byte key_id
)
```

ECC 256 ビット鍵ペアを生成し、IoT-Safe アプレット内の (書き込み可能な) スロットに保存します。

Parameters:

- **key_id** IoT-Safe アプレット内に ECC 鍵ペアが保存されているスロットの ID。
- **key_id** IoT-Safe アプレット内に ECC 鍵ペアが保存されているスロットの ID。
- **id_size** 鍵 ID のサイズ

See:

- `wc_iotsafe_ecc_gen_k_ex`
- `wc_iotsafe_ecc_sign_hash`
- `wc_iotsafe_ecc_verify_hash`
- `wc_iotsafe_ecc_gen_k`
- `wc_iotsafe_ecc_sign_hash_ex`

- `wc_iotsafe_ecc_verify_hash_ex`

Return:

- 0 成功時
- < 0 失敗の場合。
- 0 成功時
- < 0 失敗の場合。

ECC 256 ビット鍵ペアを生成し、IoT-Safe アプレット内の (書き込み可能な) スロットに保存します。
`wc_iotsafe_ecc_gen_k`と同等ですが、2 バイト以上の鍵 ID で呼び出すことができます。

C.29.3 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(byte 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_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(byte key_id);

```

C.30 dox_comments/header_files-ja/logging.h

C.30.1 Functions

	Name
int	wolfSSL_SetLoggingCb (wolfSSL_Logging_cb log_function) この関数は、wolfSSL ログメッセージを処理するために使用されるロギングコールバックを登録します。デフォルトでは、システムがサポートしている場合、stderr への fprintf() が使用されますが、この関数を使用することでユーザーは任意の処理を行うことができます。
int	wolfSSL_Debugging_ON (void) ビルド時にロギングが有効になっている場合、この関数は実行時にロギングを有効にします。ビルド時にロギングを有効にするには、-enable-debug を使用するか、DEBUG_WOLFSSL を定義します。
void	wolfSSL_Debugging_OFF (void) この関数は、実行時のロギングメッセージを無効にします。既に無効になっている場合、何も行われません。

C.30.2 Functions Documentation

C.30.2.1 function wolfSSL_SetLoggingCb

```

int wolfSSL_SetLoggingCb(
    wolfSSL_Logging_cb log_function
)

```

この関数は、wolfSSL ログメッセージを処理するために使用されるロギングコールバックを登録します。デフォルトでは、システムがサポートしている場合、stderr への fprintf() が使用されますが、この関数を使用することでユーザーは任意の処理を行うことができます。

Parameters:

- **log_function** ロギングコールバックとして登録する関数。関数シグネチャは上記のプロトタイプに従う必要があります。

See:

- **wolfSSL_Debugging_ON**

- `wolfSSL_Debugging_OFF`

Return:

- Success 成功した場合、この関数は 0 を返します。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラーです。

Example

```
int ret = 0;
// ロギングコールバックのプロトタイプ
void MyLoggingCallback(const int logLevel, const char* const logMessage);
// カスタムロギングコールバックを wolfSSL に登録
ret = wolfSSL_SetLoggingCb(MyLoggingCallback);
if (ret != 0) {
    // ロギングコールバックの設定に失敗
}
void MyLoggingCallback(const int logLevel, const char* const logMessage)
{
    // カスタムロギング関数
}
```

C.30.2.2 function wolfSSL_Debugging_ON

```
int wolfSSL_Debugging_ON(
    void
)
```

ビルド時にロギングが有効になっている場合、この関数は実行時にロギングを有効にします。ビルド時にロギングを有効にするには、`-enable-debug` を使用するか、`DEBUG_WOLFSSL` を定義します。

Parameters:

- **none** パラメータなし。

See:

- `wolfSSL_Debugging_OFF`
- `wolfSSL_SetLoggingCb`

Return:

- 0 成功時。
- NOT_COMPILED_IN このビルドでロギングが有効になっていない場合に返されるエラーです。

Example

```
wolfSSL_Debugging_ON();
```

C.30.2.3 function wolfSSL_Debugging_OFF

```
void wolfSSL_Debugging_OFF(
    void
)
```

この関数は、実行時のロギングメッセージを無効にします。既に無効になっている場合、何も行われません。

Parameters:

- **none** パラメータなし。

See:

- wolfSSL_Debugging_ON
- wolfSSL_SetLoggingCb

Return: none 戻り値なし。

Example

```
wolfSSL_Debugging_OFF();
```

C.30.3 Source code

```
int wolfSSL_SetLoggingCb(wolfSSL_Logging_cb log_function);
```

```
int wolfSSL_Debugging_ON(void);
```

```
void wolfSSL_Debugging_OFF(void);
```

C.31 dox_comments/header_files-ja/md2.h

C.31.1 Functions

	Name
int	wc_InitMd2 (wc_Md2 * md2) この関数は md2 を初期化します。これは wc_Md2Hash によって自動的に呼び出されます。
int	wc_Md2Update (wc_Md2 * md2, const byte * data, word32 len) 長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。
int	wc_Md2Final (wc_Md2 * md2, byte * hash) データのハッシュ化を完了します。結果は hash に格納されます。
int	wc_Md2Hash (const byte * data, word32 len, byte * hash) 便利な関数で、すべてのハッシュ化を処理し、結果を hash に格納します。

C.31.2 Functions Documentation

C.31.2.1 function wc_InitMd2

```
int wc_InitMd2(
    wc_Md2 * md2
)
```

この関数は md2 を初期化します。これは wc_Md2Hash によって自動的に呼び出されます。

Parameters:

- **md2** 暗号化に使用する md2 構造体へのポインタ

See:

- wc_Md2Hash
- wc_Md2Update
- wc_Md2Final

Return:

- 0 初期化に成功した場合に返されます
- BAD_FUNC_ARG md2 が NULL の場合に返されます

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.31.2.2 function wc_Md2Update

```
int wc_Md2Update(
    wc_Md2 * md2,
    const byte * data,
    word32 len
)
```

長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。

Parameters:

- **md2** 暗号化に使用する md2 構造体へのポインタ
- **data** ハッシュ化されるデータ
- **len** ハッシュ化されるデータの長さ

See:

- [wc_Md2Hash](#)
- [wc_Md2Final](#)
- [wc_InitMd2](#)

Return:

- 0 ダイジェストへのデータ追加に成功した場合に返されます。
- BAD_FUNC_ARG md2 が NULL の場合、または data が NULL で len が 0 でない場合に返されます

Example

```
md2 md2[1];
byte data[] = { }; // ハッシュ化されるデータ
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.31.2.3 function wc_Md2Final

```
int wc_Md2Final(
    wc_Md2 * md2,
    byte * hash
)
```

データのハッシュ化を完了します。結果は hash に格納されます。

Parameters:

- **md2** 暗号化に使用する md2 構造体へのポインタ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Md2Hash](#)
- [wc_Md2Final](#)
- [wc_InitMd2](#)

Return:

- 0 完了に成功した場合に返されます。
- BAD_FUNC_ARG md2 または hash が NULL の場合に返されます

Example

```
md2 md2[1];
byte data[] = { }; // ハッシュ化されるデータ
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.31.2.4 function wc_Md2Hash

```
int wc_Md2Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

便利な関数で、すべてのハッシュ化を処理し、結果を hash に格納します。

Parameters:

- **data** ハッシュ化するデータ
- **len** データの長さ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Md2Hash](#)
- [wc_Md2Final](#)
- [wc_InitMd2](#)

Return:

- 0 データのハッシュ化に成功した場合に返されます。

- Memory_E メモリエラー、メモリを割り当てることができません。これはスモールスタックオプションが有効な場合にのみ発生します。

Example

none

C.31.3 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.32 dox_comments/header_files-ja/md4.h

C.32.1 Functions

	Name
int	wc_InitMd4 (wc_Md4 * md4) この関数は md4 を初期化します。これは wc_Md4Hash によって自動的に呼び出されます。
int	wc_Md4Update (wc_Md4 * md4, const byte * data, word32 len) 長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。
int	wc_Md4Final (wc_Md4 * md4, byte * hash) データのハッシュ化を完了します。結果は hash に格納されます。

C.32.2 Functions Documentation

C.32.2.1 function wc_InitMd4

```
int wc_InitMd4(  
    wc_Md4 * md4  
)
```

この関数は md4 を初期化します。これは wc_Md4Hash によって自動的に呼び出されます。

Parameters:

- **md4** 暗号化に使用する md4 構造体へのポインタ

See:

- wc_Md4Hash
- **wc_Md4Update**
- **wc_Md4Final**

Return:

- 0 初期化に成功した場合に返されます

- BAD_FUNC_ARG md4 が NULL の場合に返されます

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.32.2.2 function wc_Md4Update

```
int wc_Md4Update(
    wc_Md4 * md4,
    const byte * data,
    word32 len
)
```

長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。

Parameters:

- **md4** 暗号化に使用する md4 構造体へのポインタ
- **data** ハッシュ化されるデータ
- **len** ハッシュ化されるデータの長さ

See:

- wc_Md4Hash
- wc_Md4Final
- wc_InitMd4

Return:

- 0 ダイジェストへのデータ追加に成功した場合に返されます。
- BAD_FUNC_ARG md4 が NULL の場合、または data が NULL で len が 0 でない場合に返されます

Example

```
md4 md4[1];
byte data[] = { }; // ハッシュ化されるデータ
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.32.2.3 function wc_Md4Final

```
int wc_Md4Final(
    wc_Md4 * md4,
    byte * hash
)
```

データのハッシュ化を完了します。結果は hash に格納されます。

Parameters:

- **md4** 暗号化に使用する md4 構造体へのポインタ
- **hash** ハッシュ値を保持するバイト配列。

See:

- wc_Md4Hash
- wc_Md4Final
- wc_InitMd4

Return:

- 0 完了に成功した場合に返されます。
- BAD_FUNC_ARG md4 または hash が NULL の場合に返されます

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.32.3 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.33 dox_comments/header_files-ja/md5.h

C.33.1 Functions

	Name
int	wc_InitMd5 (wc_Md5 * md5) この関数は md5 を初期化します。これは wc_Md5Hash によって自動的に呼び出されます。
int	wc_Md5Update (wc_Md5 * md5, const byte * data, word32 len) 長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。
int	wc_Md5Final (wc_Md5 * md5, byte * hash) データのハッシュ化を完了します。結果は hash に格納されます。Md5 構造体はリセットされます。注意：この関数は、HAVE_INTEL_QA が定義されている場合に IntelQaSymMd5() を呼び出した結果も返します。

	Name
void	wc_Md5Free (wc_Md5 * md5) Md5 構造体をリセットします。注意：これは WOLFSSL_TI_HASH が定義されている場合にのみサポートされます。
int	wc_Md5GetHash (wc_Md5 * md5, byte * hash) ハッシュデータを取得します。結果は hash に格納されます。Md5 構造体はリセットされません。

C.33.2 Functions Documentation

C.33.2.1 function wc_InitMd5

```
int wc_InitMd5(
    wc_Md5 * md5
)
```

この関数は md5 を初期化します。これは wc_Md5Hash によって自動的に呼び出されます。

Parameters:

- **md5** 暗号化に使用する md5 構造体へのポインタ

See:

- **wc_Md5Hash**
- **wc_Md5Update**
- **wc_Md5Final**

Return:

- 0 正常に初期化された場合に返されます。
- BAD_FUNC_ARG Md5 構造体が NULL 値として渡された場合に返されます。

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 失敗のケース。
    }
    ret = wc_Md5Final(&md5, hash);
    if (ret != 0) {
        // Md5 Final 失敗のケース。
    }
}
```

C.33.2.2 function wc_Md5Update

```
int wc_Md5Update(
    wc_Md5 * md5,
    const byte * data,
    word32 len
)
```

長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。

Parameters:

- **md5** 暗号化に使用する md5 構造体へのポインタ
- **data** ハッシュ化されるデータ
- **len** ハッシュ化されるデータの長さ

See:

- [wc_Md5Hash](#)
- [wc_Md5Final](#)
- [wc_InitMd5](#)

Return:

- 0 ダイジェストへのデータ追加に成功した場合に返されます。
- BAD_FUNC_ARG Md5 構造体が NULL の場合、または data が NULL で len がゼロより大きい場合に返されます。data パラメータが NULL で len がゼロの場合、関数はエラーを返すべきではありません。

Example

```
Md5 md5;
byte data[] = { ハッシュ化されるデータ };
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 エラーのケース。
    }
    ret = wc_Md5Final(&md5, hash);
    if (ret != 0) {
        // Md5 Final エラーのケース。
    }
}
```

C.33.2.3 function wc_Md5Final

```
int wc_Md5Final(
    wc_Md5 * md5,
    byte * hash
)
```

データのハッシュ化を完了します。結果は hash に格納されます。Md5 構造体はリセットされます。注意：この関数は、HAVE_INTEL_QA が定義されている場合に IntelQaSymMd5() を呼び出した結果も返します。

Parameters:

- **md5** 暗号化に使用する md5 構造体へのポインタ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Md5Hash](#)
- [wc_InitMd5](#)
- [wc_Md5GetHash](#)

Return:

- 0 正常に完了した場合に返されます。
- BAD_FUNC_ARG Md5 構造体または hash ポインタが NULL で渡された場合に返されます。

Example

```
md5 md5[1];
byte data[] = { ハッシュ化されるデータ };
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 失敗のケース。
    }
    ret = wc_Md5Final(md5, hash);
    if (ret != 0) {
        // Md5 Final 失敗のケース。
    }
}
```

C.33.2.4 function wc_Md5Free

```
void wc_Md5Free(
    wc_Md5 * md5
)
```

Md5 構造体をリセットします。注意：これは WOLFSSL_TI_HASH が定義されている場合にのみサポートされます。

Parameters:

- **md5** リセットする Md5 構造体へのポインタ。

See:

- [wc_InitMd5](#)
- [wc_Md5Update](#)
- [wc_Md5Final](#)

Return: none 戻り値なし。

Example

```
Md5 md5;
byte data[] = { ハッシュ化されるデータ };
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.33.2.5 function wc_Md5GetHash

```
int wc_Md5GetHash(
    wc_Md5 * md5,
    byte * hash
)
```

ハッシュデータを取得します。結果は hash に格納されます。Md5 構造体はリセットされません。

Parameters:

- **md5** 暗号化に使用する md5 構造体へのポインタ。
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Md5Hash](#)
- [wc_Md5Final](#)
- [wc_InitMd5](#)

Return: none 戻り値なし

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.33.3 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.34 dox_comments/header_files-ja/memory.h**C.34.1 Functions**

	Name
void *	**wolfSSL_Mallocを参照してください。 wolfSSL_Malloc は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ X_MALLOC によって呼び出されることに注意してください。デフォルトのビルドでは、size パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、heap と type パラメータが含まれます。
void	**wolfSSL_Freeを参照してください。 wolfSSL_Free は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ X_FREE によって呼び出されることに注意してください。デフォルトのビルドでは、ptr パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、heap と type パラメータが含まれます。
void *	**wolfSSL_Reallocを参照してください。 wolfSSL_Realloc は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ X_REALLOC によって呼び出されることに注意してください。デフォルトのビルドでは、size パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、heap と type パラメータが含まれます。
int	wolfSSL_SetAllocators(wolfSSL_Malloc_cb mf, wolfSSL_Free_cb ff, wolfSSL_Realloc_cb rf) この関数は、wolfSSL が使用する割り当て関数を登録します。デフォルトでは、システムがサポートしている場合、malloc/free と realloc が使用されます。この関数を使用すると、ユーザーは実行時に独自のメモリハンドラをインストールできます。
int	wolfSSL_StaticBufferSz(byte * buffer, word32 sz, int flag) この関数は、静的メモリ機能が使用されている場合に利用可能です (-enable-staticmemory)。メモリ「バケット」の最適なバッファサイズを提供します。これにより、パーティション化された後に余分な未使用メモリが残らないようにバッファサイズを計算する方法が提供されます。この関数の非_ex バージョンでは、コンパイル時に設定されたデフォルトのバケットと配布リストが使用されます。返される値が正の場合、使用する計算されたバッファサイズです。
int	wolfSSL_MemoryPaddingSz(void) この関数は、静的メモリ機能が使用されている場合に利用可能です (-enable-staticmemory)。メモリの各パーティションに必要なパディングのサイズを提供します。このパディングサイズは、メモリ管理構造体を含むために必要なサイズと、メモリアライメントのための追加分になります。

	Name
int	wolfSSL_CTX_load_static_memory (WOLFSSL_CTX ** ctx, wolfSSL_method_func method, unsigned char * buf, unsigned int sz, int flag, int max) この関数は、CTX のために静的メモリを確保するために使用されます。確保されたメモリは、CTX の存続期間中および CTX から作成されたすべての SSL オブジェクトに使用されます。NULL の ctx ポインタと wolfSSL_method_func 関数を渡すことにより、CTX 自体の作成も静的メモリを使用します。wolfSSL_method_func は、WOLFSSL_METHOD* (wolfSSL_method_func)(void heap); の関数シグネチャを持ちます。max に 0 を渡すと、設定されていないかのように動作し、最大同時使用制限が適用されません。渡される flag 値は、メモリの使用方法と動作中の動作を決定します。利用可能なフラグは次のとおりです。
int	wolfSSL_CTX_is_static_memory (WOLFSSL_CTX * ctx, WOLFSSL_MEM_STATS * mem_stats) この関数は接続の動作を変更せず、静的メモリ使用に関する情報を収集するためにのみ使用されます。
int	wolfSSL_is_static_memory (WOLFSSL * ssl, WOLFSSL_MEM_CONN_STATS * mem_stats)wolfSSL_is_static_memory は、SSL の静的メモリ使用に関する情報を収集するために使用されます。戻り値は、静的メモリが使用されているかどうかを示し、WOLFSSL_MEM_CONN_STATS は、静的メモリをロードする際に親 CTX に WOLFMEM_TRACK_STATS フラグが渡された場合にのみ入力されます。
int	wc_LoadStaticMemory (WOLFSSL_HEAP_HINT ** pHint, unsigned char * buf, unsigned int sz, int flag, int max) この関数は、wolfCrypt 使用のために静的メモリを確保するために使用されます。作成されたヒープヒントを関数に渡すことでメモリを使用できます。この例は、wc_InitRng_ex を呼び出すときです。渡される flag 値は、メモリの使用方法と動作中の動作を決定します。一般的に、wolfCrypt 操作は WOLFMEM_GENERAL プールからメモリを使用します。利用可能なフラグは次のとおりです。
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) この関数は、カスタムバケットサイズと配分を使用して wolfCrypt 使用のために静的メモリを確保するために使用されます。作成されたヒープヒントを関数に渡すことでメモリを使用できます。この拡張バージョンでは、デフォルトの事前定義されたサイズを使用する代わりに、カスタムバケットサイズと配分を使用できます。

	Name
WOLFSSL_HEAP_HINT *	wolfSSL_SetGlobalHeapHint (WOLFSSL_HEAP_HINT * hint) この関数は、NULL ヒープヒントがメモリ割り当て関数に渡されたときに使用されるグローバルヒープヒントを設定します。これにより、アプリケーション全体で使用されるデフォルトのヒープヒントを設定できます。
WOLFSSL_HEAP_HINT *	wolfSSL_GetGlobalHeapHint (void) この関数は、NULL ヒープヒントがメモリ割り当て関数に渡されたときに使用される現在のグローバルヒープヒントを取得します。
void	wolfSSL_SetDebugMemoryCb (DebugMemoryCb cb) この関数は、静的メモリ割り当て追跡用のデバッグコールバック関数を設定します。WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACK ビルドオプションと共に使用されます。コールバック関数は、メモリ割り当ておよび割り当て解除操作中に呼び出され、デバッグ情報を提供します。
void	wc_UnloadStaticMemory (WOLFSSL_HEAP_HINT * heap) この関数は、静的メモリヒープと関連するミューテックスを解放します。静的メモリ割り当ての使用が完了したときに、リソースを適切にクリーンアップするために呼び出す必要があります。
int	wolfSSL_StaticBufferSz_ex (unsigned int listSz, const word32 * sizeList, const word32 * distList, byte * buffer, word32 sz, int flag) この関数は、カスタムバケットサイズと配分を使用した静的メモリ割り当てに必要なバッファサイズを計算します。この拡張バージョンでは、デフォルトの事前定義されたサイズを使用する代わりに、カスタムバケットサイズを使用できます。

C.34.2 Functions Documentation

C.34.2.1 function wolfSSL_Malloc

```
void * wolfSSL_Malloc(
    size_t size,
    void * heap,
    int type
)
```

この関数は malloc() に似ていますが、wolfSSL が使用するよう設定されたメモリ割り当て関数を呼び出します。デフォルトでは、wolfSSL は malloc() を使用します。これは wolfSSL メモリ抽象化レイヤーを使用して変更できます - **wolfSSL_SetAllocators()** を参照してください。wolfSSL_Malloc は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ XMALLOC によって呼び出されることに注意してください。デフォルトのビルドでは、size パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、heap と type パラメータが含まれます。

Parameters:

- **size** 割り当てるメモリのバイト単位のサイズ。
- **heap** メモリに使用するヒープヒント。NULL にできます。

- **type** 動的タイプ (types.h の DYNAMIC_TYPE_ リストを参照)。

See:

- wolfSSL_Free
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer 成功した場合、この関数は割り当てられたメモリへのポインタを返します。
- error エラーがある場合、NULL が返されます。

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
```

C.34.2.2 function wolfSSL_Free

```
void wolfSSL_Free(
    void * ptr,
    void * heap,
    int type
)
```

この関数は free() に似ていますが、wolfSSL が使用するよう設定されたメモリ解放関数を呼び出します。デフォルトでは、wolfSSL は free() を使用します。これは wolfSSL メモリ抽象化レイヤーを使用して変更できます - wolfSSL_SetAllocators() を参照してください。wolfSSL_Free は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ XFREE によって呼び出されることに注意してください。デフォルトのビルドでは、ptr パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、heap と type パラメータが含まれます。

Parameters:

- **ptr** 解放するメモリへのポインタ。
- **heap** メモリに使用するヒープヒント。NULL にできます。
- **type** 動的タイプ (types.h の DYNAMIC_TYPE_ リストを参照)。

See:

- wolfSSL_Alloc
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return: none 戻り値なし。

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
// 必要に応じてデータを処理
...
if(tenInts) {
    wolfSSL_Free(tenInts, NULL, DYNAMIC_TYPE_TMP_BUFFER);
}
```

C.34.2.3 function wolfSSL_Realloc

```
void * wolfSSL_Realloc(
    void * ptr,
    size_t size,
    void * heap,
    int type
)
```

この関数は realloc() に似ていますが、wolfSSL が使用するよう設定されたメモリ再割り当て関数を呼び出します。デフォルトでは、wolfSSL は realloc() を使用します。これは wolfSSL メモリ抽象化レイヤーを使用して変更できます - [wolfSSL_SetAllocators\(\)](#) を参照してください。wolfSSL_Realloc は直接 wolfSSL によって呼び出されるのではなく、代わりにマクロ XREALLOC によって呼び出されることに注意してください。デフォルトのビルドでは、size パラメータのみが存在します。WOLFSSL_STATIC_MEMORY ビルドを使用している場合は、heap と type パラメータが含まれます。

Parameters:

- **ptr** 再割り当てする、以前に割り当てられたメモリへのポインタ。
- **size** 割り当てるバイト数。
- **heap** メモリに使用するヒープヒント。NULL にできます。
- **type** 動的タイプ (types.h の DYNAMIC_TYPE_ リストを参照)。

See:

- [wolfSSL_Free](#)
- [wolfSSL_Malloc](#)
- [wolfSSL_SetAllocators](#)
- [XMALLOC](#)
- [XFREE](#)
- [XREALLOC](#)

Return:

- pointer 成功した場合、この関数は再割り当てされたメモリへのポインタを返します。これは ptr と同じポインタである場合もあれば、新しいポインタの場所である場合もあります。
- Null エラーがある場合、NULL が返されます。

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
int* twentyInts = (int*)wolfSSL_Realloc(tenInts, sizeof(int)*20);
```

C.34.2.4 function wolfSSL_SetAllocators

```
int wolfSSL_SetAllocators(
    wolfSSL_Malloc_cb mf,
    wolfSSL_Free_cb ff,
    wolfSSL_Realloc_cb rf
)
```

この関数は、wolfSSL が使用する割り当て関数を登録します。デフォルトでは、システムがサポートしている場合、malloc/free と realloc が使用されます。この関数を使用すると、ユーザーは実行時に独自のメモリハンドラをインストールできます。

Parameters:

- **malloc_function** wolfSSL が使用するメモリ割り当て関数。関数シグネチャは上記の wolfSSL_Malloc_cb プロトタイプと一致する必要があります。
- **free_function** wolfSSL が使用するメモリ解放関数。関数シグネチャは上記の wolfSSL_Free_cb プロトタイプと一致する必要があります。

- **realloc_function** wolfSSL が使用するメモリ再割り当て関数。関数シグネチャは上記の wolfSSL_Realloc_cb プロトタイプと一致する必要があります。

See: none

Return:

- Success 成功した場合、この関数は 0 を返します。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラー。

Example

```
static void* MyMalloc(size_t size)
{
    // カスタム malloc 関数
}

static void MyFree(void* ptr)
{
    // カスタム free 関数
}

static void* MyRealloc(void* ptr, size_t size)
{
    // カスタム realloc 関数
}

// カスタムメモリ関数を wolfSSL に登録
int ret = wolfSSL_SetAllocators(MyMalloc, MyFree, MyRealloc);
if (ret != 0) {
    // メモリ関数の設定に失敗
}
```

C.34.2.5 function wolfSSL_StaticBufferSz

```
int wolfSSL_StaticBufferSz(
    byte * buffer,
    word32 sz,
    int flag
)
```

この関数は、静的メモリ機能が使用されている場合に利用可能です (-enable-staticmemory)。メモリ「バケット」の最適なバッファサイズを提供します。これにより、パーティション化された後に余分な未使用メモリが残らないようにバッファサイズを計算する方法が提供されます。この関数の非_ex バージョンでは、コンパイル時に設定されたデフォルトのバケットと配布リストが使用されます。返される値が正の場合、使用する計算されたバッファサイズです。

Parameters:

- **buffer** バッファへのポインタ。
- **size** バッファのサイズ。
- **type** 希望するメモリタイプ、つまり WOLFMEM_GENERAL または WOLFMEM_IO_POOL。

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- Success バッファサイズの計算が正常に完了すると、正の値が返されます。この返される値は最適なバッファサイズです。
- Failure すべての負の値はエラーケースと見なされます。

Example

```
byte buffer[1000];
word32 size = sizeof(buffer);
int optimum;

optimum = wolfSSL_StaticBufferSz(buffer, size, WOLFMEM_GENERAL);
if (optimum < 0) { //エラーケースを処理 }
printf(" すべてのメモリを利用するための最適なバッファサイズは %d です\n",
optimum);
...
```

C.34.2.6 function wolfSSL_MemoryPaddingSz

```
int wolfSSL_MemoryPaddingSz(
    void
)
```

この関数は、静的メモリ機能が使用されている場合に利用可能です (-enable-staticmemory)。メモリの各パーティションに必要なパディングのサイズを提供します。このパディングサイズは、メモリ管理構造体を含むために必要なサイズと、メモリアライメントのための追加分になります。

Parameters:

- **none** パラメータなし。

See:

- `wolfSSL_Malloc`
- `wolfSSL_Free`

Return:

- メモリパディングの計算が成功すると、戻り値は正の値になります。
- すべての負の値はエラーケースと見なされます。

Example

```
int padding;
padding = wolfSSL_MemoryPaddingSz();
if (padding < 0) { //エラーケースを処理 }
printf(" メモリの各「バケット」に必要なパディングサイズは %d です\n",
padding);
// IO POOL サイズのバッファの計算は、バケット数
// × (padding + WOLFMEM_IO_SZ)
...
```

C.34.2.7 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
)
```

この関数は、CTX のために静的メモリを確保するために使用されます。確保されたメモリは、CTX の存続期間中および CTX から作成されたすべての SSL オブジェクトに使用されます。NULL の ctx ポインタと wolfSSL_method_func 関数を渡すことにより、CTX 自体の作成も静的メモリを使用します。wolfSSL_method_func は、WOLFSSL_METHOD* (wolfSSL_method_func)(void heap); の関数シグネチャを持ちます。max に 0 を渡すと、設定されていないかのように動作し、最大同時使用制限が適用されません。渡される flag 値は、メモリの使用方法と動作中の動作を決定します。利用可能なフラグは次のとおりです。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタのアドレス。
- **method** プロトコルを作成する関数。(ctx も NULL でない場合は NULL である必要があります)
- **buf** すべての操作に使用するメモリ。
- **sz** 渡されるメモリバッファのサイズ。
- **flag** メモリのタイプ。
- **max** 最大同時操作数。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_is_static_memory](#)
- [wolfSSL_is_static_memory](#)

Return:

- 成功した場合、SSL_SUCCESS が返されます。
- すべての失敗した戻り値は 0 未満または SSL_FAILURE と等しくなります。

0 - デフォルトの一般メモリ

WOLFMEM_IO_POOL - メッセージの送受信時の入出力バッファに使用されます。一般メモリをオーバーライドするため、渡されたバッファ内のすべてのメモリが IO に使用されます。WOLFMEM_IO_FIXED - WOLFMEM_IO_POOL と同じですが、各 SSL は存続期間中に 2 つのバッファを保持します。WOLFMEM_TRACK_STATS - 各 SSL は実行中にメモリ統計を追跡します。

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;
...
// 静的メモリを使用して ctx も作成、使用する一般メモリから開始
ctx = NULL;
ret = wolfSSL_CTX_load_static_memory(&ctx, wolfSSLv23_server_method_ex,
    ↪ memory, memorySz, 0,
    MAX_CONCURRENT_HANDSHAKES);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}
// IO で使用するメモリをロード
ret = wolfSSL_CTX_load_static_memory(&ctx, NULL, IO, IOSz, flag,
    ↪ MAX_CONCURRENT_IO);
if (ret != SSL_SUCCESS) {
```

```
// エラーケースを処理
}
...
```

C.34.2.8 function wolfSSL_CTX_is_static_memory

```
int wolfSSL_CTX_is_static_memory(
    WOLFSSL_CTX * ctx,
    WOLFSSL_MEM_STATS * mem_stats
)
```

この関数は接続の動作を変更せず、静的メモリ使用に関する情報を収集するためにのみ使用されます。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された `WOLFSSL_CTX` 構造体へのポインタ。
- **mem_stats** 静的メモリ使用に関する情報を保持する構造体。

See:

- `wolfSSL_CTX_new`
- `wolfSSL_CTX_load_static_memory`
- `wolfSSL_is_static_memory`

Return:

- `CTX` に静的メモリを使用している場合は 1 の値が返されます。
- 静的メモリを使用していない場合は 0 が返されます。

Example

```
WOLFSSL_CTX* ctx;
int ret;
WOLFSSL_MEM_STATS mem_stats;
...
//CTX での静的メモリに関する情報を取得

ret = wolfSSL_CTX_is_static_memory(ctx, &mem_stats);

if (ret == 1) {
    // 静的メモリを使用しているケースを処理
    // mem_stats の要素を出力または検査
}

if (ret == 0) {
    //ctx が静的メモリを使用していないケースを処理
}
...
```

C.34.2.9 function wolfSSL_is_static_memory

```
int wolfSSL_is_static_memory(
    WOLFSSL * ssl,
    WOLFSSL_MEM_CONN_STATS * mem_stats
)
```

`wolfSSL_is_static_memory` は、SSL の静的メモリ使用に関する情報を収集するために使用されます。戻り値は、静的メモリが使用されているかどうかを示し、`WOLFSSL_MEM_CONN_STATS` は、静的メモリをロードする際に親 `CTX` に `WOLFMEM_TRACK_STATS` フラグが渡された場合にのみ入力されます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **mem_stats** 静的メモリ使用を含む構造体。

See:

- `wolfSSL_new`
- `wolfSSL_CTX_is_static_memory`

Return:

- CTX に静的メモリを使用している場合は 1 の値が返されます。
- 静的メモリを使用していない場合は 0 が返されます。

Example

```
WOLFSSL* ssl;
int ret;
WOLFSSL_MEM_CONN_STATS mem_stats;

...

ret = wolfSSL_is_static_memory(ssl, mem_stats);

if (ret == 1) {
    // 静的メモリの場合のケースを処理
    // WOLFMEM_TRACK_STATS フラグがある場合は mem_stats の要素を調査
}
...
```

C.34.2.10 function wc_LoadStaticMemory

```
int wc_LoadStaticMemory(
    WOLFSSL_HEAP_HINT ** pHint,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

この関数は、wolfCrypt 使用のために静的メモリを確保するために使用されます。作成されたヒープヒントを関数に渡すことでメモリを使用できます。この例は、`wc_InitRng_ex` を呼び出すときです。渡される `flag` 値は、メモリの使用方法と動作中の動作を決定します。一般的に、wolfCrypt 操作は `WOLFMEM_GENERAL` プールからメモリを使用します。利用可能なフラグは次のとおりです。

Parameters:

- **phint** 使用する `WOLFSSL_HEAP_HINT` 構造体。
- **buf** すべての操作に使用するメモリ。
- **sz** 渡されるメモリバッファのサイズ。
- **flag** メモリのタイプ。
- **max** 最大同時操作数（ハンドシェイク、IO）。

See: none

Return: none この関数は値を返しません。

`WOLFMEM_GENERAL` - デフォルトの一般メモリ

WOLFMEM_IO_POOL - メッセージの送受信時の入出力バッファに使用されます。一般メモリをオーバーライドするため、渡されたバッファ内のすべてのメモリが IO に使用されます。WOLFMEM_IO_FIXED - WOLFMEM_IO_POOL と同じですが、各 SSL は存続期間中に 2 つのバッファを保持します。WOLFMEM_TRACK_STATS - 各 SSL は実行中にメモリ統計を追跡します。

Example

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
int flag = WOLFMEM_GENERAL | WOLFMEM_TRACK_STATS;
...

// 使用するメモリをロード

ret = wc_LoadStaticMemory(&hint, memory, memorySz, flag, 0);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}
...

ret = wc_InitRng_ex(&rng, hint, 0);

// ret 値をチェック
```

C.34.2.11 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
)
```

この関数は、カスタムバケットサイズと配分を使用して wolfCrypt 使用のために静的メモリを確保するために使用されます。作成されたヒープヒントを関数に渡すことでメモリを使用できます。この拡張バージョンでは、デフォルトの事前定義されたサイズを使用する代わりに、カスタムバケットサイズと配分を使用できます。

Parameters:

- **pHint** 初期化する WOLFSSL_HEAP_HINT ハンドル。
- **listSz** サイズリストと分布リストのエントリ数。
- **sizeList** 使用するバケットサイズの配列。
- **distList** sizeList に対応する分布リスト。
- **buf** すべての操作に使用するメモリ。
- **sz** 渡されるメモリバッファのサイズ。
- **flag** メモリのタイプ。
- **max** 最大同時操作数（ハンドシェイク、IO）。

See:

- [wc_LoadStaticMemory](#)

- [wc_UnloadStaticMemory](#)

Return: none この関数は値を返しません。

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]));

...

// カスタムバケットサイズで使用するメモリをロード

ret = wc_LoadStaticMemory_ex(&hint, listSz, sizeList, distList,
                             memory, memorySz, flag, 0);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}

...

ret = wc_InitRng_ex(&rng, hint, 0);

// ret 値をチェック
```

C.34.2.12 function wolfSSL_SetGlobalHeapHint

```
WOLFSSL_HEAP_HINT * wolfSSL_SetGlobalHeapHint(
    WOLFSSL_HEAP_HINT * hint
)
```

この関数は、NULL ヒープヒントがメモリ割り当て関数に渡されたときに使用されるグローバルヒープヒントを設定します。これにより、アプリケーション全体で使用されるデフォルトのヒープヒントを設定できます。

Parameters:

- **hint** グローバルヒープヒントとして使用する WOLFSSL_HEAP_HINT 構造体。

See:

- [wolfSSL_GetGlobalHeapHint](#)
- [wc_LoadStaticMemory](#)

Return: 設定されていた以前のグローバルヒープヒントを返します。

Example

```
WOLFSSL_HEAP_HINT hint;
WOLFSSL_HEAP_HINT* prev_hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;

...
```

```
// 使用するメモリをロード
ret = wc_LoadStaticMemory(&hint, memory, memorySz, WOLFMEM_GENERAL, 0);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}

// グローバルヒープヒントとして設定
prev_hint = wolfSSL_SetGlobalHeapHint(&hint);
if (prev_hint != NULL) {
    // 以前のグローバルヒープヒントがありました
}
```

C.34.2.13 function wolfSSL_GetGlobalHeapHint

```
WOLFSSL_HEAP_HINT * wolfSSL_GetGlobalHeapHint(
    void
)
```

この関数は、NULL ヒープヒントがメモリ割り当て関数に渡されたときに使用される現在のグローバルヒープヒントを取得します。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_SetGlobalHeapHint](#)
- [wc_LoadStaticMemory](#)

Return: 現在のグローバルヒープヒントを返します。設定されていない場合は NULL を返します。

Example

```
WOLFSSL_HEAP_HINT* current_hint;
...

current_hint = wolfSSL_GetGlobalHeapHint();
if (current_hint != NULL) {
    // グローバルヒープヒントが設定されています
    // current_hint を操作に使用できます
}
```

C.34.2.14 function wolfSSL_SetDebugMemoryCb

```
void wolfSSL_SetDebugMemoryCb(
    DebugMemoryCb cb
)
```

この関数は、静的メモリ割り当て追跡用のデバッグコールバック関数を設定します。WOLFSSL_STATIC_MEMORY_DEBUG_CALLBACK ビルドオプションと共に使用されます。コールバック関数は、メモリ割り当ておよび割り当て解除操作中に呼び出され、デバッグ情報を提供します。

Parameters:

- **cb** 設定するデバッグコールバック関数。

See: none

Return:

- 成功した場合、0 が返されます。
- すべての失敗した戻り値は 0 未満になります。

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);
}
...

// デバッグコールバックを設定
int ret = wolfSSL_SetDebugMemoryCb(debug_memory_cb);
if (ret != 0) {
    // エラーケースを処理
}
```

C.34.2.15 function wc_UnloadStaticMemory

```
void wc_UnloadStaticMemory(
    WOLFSSL_HEAP_HINT * heap
)
```

この関数は、静的メモリヒープと関連するミューテックスを解放します。静的メモリ割り当ての使用が完了したときに、リソースを適切にクリーンアップするために呼び出す必要があります。

Parameters:

- **hint** アンロードする WOLFSSL_HEAP_HINT 構造体。

See:

- [wc_LoadStaticMemory](#)
- [wc_LoadStaticMemory_ex](#)

Return:

- 成功した場合、0 が返されます。
- すべての失敗した戻り値は 0 未満になります。

Example

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
...

// 使用するメモリをロード
ret = wc_LoadStaticMemory(&hint, memory, memorySz, WOLFMEM_GENERAL, 0);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理
}

// 操作にメモリを使用
...
```

```
// 完了時にクリーンアップ
wc_UnloadStaticMemory(&hint);
```

C.34.2.16 function wolfSSL_StaticBufferSz_ex

```
int wolfSSL_StaticBufferSz_ex(
    unsigned int listSz,
    const word32 * sizeList,
    const word32 * distList,
    byte * buffer,
    word32 sz,
    int flag
)
```

この関数は、カスタムバケットサイズと配分を使用した静的メモリ割り当てに必要なバッファサイズを計算します。この拡張バージョンでは、デフォルトの事前定義されたサイズを使用する代わりに、カスタムバケットサイズを使用できます。

Parameters:

- **bucket_sizes** 使用するバケットサイズの配列。
- **bucket_count** 配列内のバケットサイズの数。
- **flag** 希望するメモリタイプ、つまり WOLFMEM_GENERAL または WOLFMEM_IO_POOL。

See:

- [wolfSSL_StaticBufferSz](#)
- [wc_LoadStaticMemory_ex](#)

Return:

- バッファサイズの計算が正常に完了すると、正の値が返されます。
- すべての負の値はエラーケースと見なされます。

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) { //エラーケースを処理 }
printf(" カスタムバケットでの最適なバッファサイズは %d です\n", optimum);
...
```

C.34.3 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.35 dox_comments/header_files-ja/pem.h

C.35.1 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) この関数は、WOLFSSL_BIO 構造体にキーを PEM 形式で書き込みます。

C.35.2 Functions Documentation

C.35.2.1 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
)

```

この関数は、WOLFSSL_BIO 構造体にキーを PEM 形式で書き込みます。

Parameters:

- **bio** PEM バッファを取得する WOLFSSL_BIO 構造体。
- **key** PEM 形式に変換するキー。
- **cipher** EVP 暗号構造体。
- **passwd** パスワード。
- **len** パスワードの長さ。
- **cb** パスワードコールバック。
- **arg** オプション引数。

See: [wolfSSL_PEM_read_bio_X509_AUX](#)

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE 失敗時。

Example

```

WOLFSSL_BIO* bio;
WOLFSSL_EVP_PKEY* key;
int ret;
// bio を作成してキーをセットアップ
ret = wolfSSL_PEM_write_bio_PrivateKey(bio, key, NULL, NULL, 0, NULL, NULL);
//ret 値を確認

```

C.35.3 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.36 dox_comments/header_files-ja/pkcs11.h

C.36.1 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)

C.36.2 Functions Documentation

C.36.2.1 function wc_Pkcs11_Initialize

```
int wc_Pkcs11_Initialize(
    Pkcs11Dev * dev,
    const char * library,
    void * heap
)
```

C.36.2.2 function wc_Pkcs11_Finalize

```
void wc_Pkcs11_Finalize(
    Pkcs11Dev * dev
)
```

C.36.2.3 function wc_Pkcs11Token_Init

```
int wc_Pkcs11Token_Init(
    Pkcs11Token * token,
    Pkcs11Dev * dev,
    int slotId,
    const char * tokenName,
    const unsigned char * userPin,
    int userPinSz
)
```

C.36.2.4 function wc_Pkcs11Token_Final

```
void wc_Pkcs11Token_Final(
    Pkcs11Token * token
)
```

C.36.2.5 function wc_Pkcs11Token_Open

```
int wc_Pkcs11Token_Open(
    Pkcs11Token * token,
    int readWrite
)
```

C.36.2.6 function wc_Pkcs11Token_Close

```
void wc_Pkcs11Token_Close(
    Pkcs11Token * token
)
```


C.36.2.7 function wc_Pkcs11StoreKey

```
int wc_Pkcs11StoreKey(
    Pkcs11Token * token,
    int type,
    int clear,
    void * key
)
```

C.36.2.8 function wc_Pkcs11_CryptoDevCb

```
int wc_Pkcs11_CryptoDevCb(
    int devId,
    wc_CryptoInfo * info,
    void * ctx
)
```

C.36.3 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.37 dox_comments/header_files-ja/pkcs7.h**C.37.1 Types**

	Name
typedef int()(const byte key, word32 keySz, const byte in, word32 inSz, int wrap, byte out, word32 outSz)	CallbackAESKeyWrapUnwrap カスタム AES 鍵ラップ/アンラップ操作に使用されるコールバック。

C.37.2 Functions

	Name
int	wc_PKCS7_InitWithCert (wc_PKCS7 * pkcs7, byte * der, word32 derSz) この関数は、DER 形式の証明書で PKCS7 構造体を初期化します。空の PKCS7 構造体を初期化するには、cert に NULL を、certSz に 0 を渡すことができます。
void	wc_PKCS7_Free (wc_PKCS7 * pkcs7) この関数は、PKCS7 初期化子によって割り当てられたメモリを解放します。
int	wc_PKCS7_EncodeData (wc_PKCS7 * pkcs7, byte * output, word32 outputSz) この関数は、PKCS7 データコンテンツタイプをビルドし、PKCS7 構造体を解析可能な PKCS7 データパケットを含むバッファにエンコードします。
int	wc_PKCS7_EncodeSignedData (wc_PKCS7 * pkcs7, byte * output, word32 outputSz) この関数は、PKCS7 署名付きデータコンテンツタイプをビルドし、PKCS7 構造体を解析可能な PKCS7 署名付きデータパケットを含むバッファにエンコードします。
int	wc_PKCS7_EncodeSignedData_ex (wc_PKCS7 * pkcs7, const byte * hashBuf, word32 hashSz, byte * outputHead, word32 * outputHeadSz, byte * outputFoot, word32 * outputFootSz) この関数は、PKCS7 署名付きデータコンテンツタイプをビルドし、PKCS7 構造体を解析可能な PKCS7 署名付きデータパケットを含むヘッダーとフッターバッファにエンコードします。これにはコンテンツは含まれません。データのハッシュを計算して提供する必要があります。
int	wc_PKCS7_VerifySignedData (wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz) この関数は、送信された PKCS7 署名付きデータメッセージを受け取り、証明書リストと証明書失効リストを抽出し、署名を検証します。抽出されたコンテンツを与えられた PKCS7 構造体に格納します。
int	wc_PKCS7_VerifySignedData_ex (wc_PKCS7 * pkcs7, const byte * hashBuf, word32 hashSz, byte * pkiMsgHead, word32 pkiMsgHeadSz, byte * pkiMsgFoot, word32 pkiMsgFootSz) この関数は、ハッシュ/ヘッダー/フッターとして送信された PKCS7 署名付きデータメッセージを受け取り、証明書リストと証明書失効リストを抽出し、署名を検証します。抽出されたコンテンツを与えられた PKCS7 構造体に格納します。
int	**wc_PKCS7_SetAESKeyWrapUnwrapCb (aesKeyWrapCb) カスタム AES 鍵ラップ/アンラップ操作を実行するために使用されるコールバック関数を設定します。

	Name
int	wc_PKCS7_EncodeEnvelopedData (wc_PKCS7 * pkcs7, byte * output, word32 outputSz) この関数は、PKCS7 エンベロープデータコンテンツタイプをビルドし、PKCS7 構造体を解析可能な PKCS7 エンベロープデータパケットを含むバッファにエンコードします。
int	wc_PKCS7_DecodeEnvelopedData (wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz) この関数は、PKCS7 エンベロープデータコンテンツタイプをアンラップして復号し、メッセージを output にデコードします。渡された PKCS7 オブジェクトの秘密鍵を使用してメッセージを復号します。
int	wc_PKCS7_GetEnvelopedDataKariRid (const byte * in, word32 inSz, byte * out, word32 * outSz) この関数は、KeyAgreeRecipientInfo RecipientInfo オブジェクトを含む EnvelopedData パッケージから KeyAgreeRecipientIdentifier オブジェクトを抽出します。最初の RecipientInfo で見つかった最初の KeyAgreeRecipientIdentifier のみがコピーされます。この関数は、複数の RecipientInfo オブジェクトや KeyAgreeRecipientInfo 内の複数の RecipientEncryptedKey オブジェクトをサポートしていません。
int	wc_PKCS7_DecodeEncryptedData (wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz) この関数は、PKCS7 暗号化データコンテンツタイプをアンラップして復号し、メッセージを output にデコードします。pkcs7->encryptionKey と pkcs7->encryptionKeySz を介して渡された PKCS7 オブジェクトの暗号化鍵を使用してメッセージを復号します。
int	wc_PKCS7_DecodeEncryptedKeyPackage (wc_PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz) この関数は、PKCS7 暗号化鍵パッケージコンテンツタイプをアンラップして復号し、メッセージを output にデコードします。ラップされたコンテンツタイプが EncryptedData の場合、pkcs7 入力構造体に暗号化鍵を設定する必要があります (pkcs7->encryptionKey と pkcs7->encryptionKeySz を介して)。ラップされたコンテンツタイプが EnvelopedData の場合、pkcs7 入力構造体に秘密鍵を設定する必要があります (pkcs7->privateKey と pkcs7->privateKeySz を介して)。AuthEnvelopedData のラップされたコンテンツタイプは現在サポートされていません。

	Name
int	wc_PKCS7_DecodeSymmetricKeyPackageAttribute (const byte * skp, word32 skpSz, size_t index, const byte ** attr, word32 * attrSz) この関数は、SymmetricKeyPackage 属性へのアクセスを提供します。
int	wc_PKCS7_DecodeSymmetricKeyPackageKey (const byte * skp, word32 skpSz, size_t index, const byte ** key, word32 * keySz) この関数は、SymmetricKeyPackage 鍵へのアクセスを提供します。
int	wc_PKCS7_DecodeOneSymmetricKeyAttribute (const byte * osk, word32 oskSz, size_t index, const byte ** attr, word32 * attrSz) この関数は、OneSymmetricKey 属性へのアクセスを提供します。
int	wc_PKCS7_DecodeOneSymmetricKeyKey (const byte * osk, word32 oskSz, const byte ** key, word32 * keySz) この関数は、OneSymmetricKey 鍵へのアクセスを提供します。

C.37.3 Types Documentation

C.37.3.1 typedef CallbackAESKeyWrapUnwrap

```
typedef int(* CallbackAESKeyWrapUnwrap) (const byte *key, word32 keySz, const
↳ byte *in, word32 inSz, int wrap, byte *out, word32 outSz);
```

カスタム AES 鍵ラップ/アンラップ操作に使用されるコールバック。

Parameters:

- **key** 使用する鍵を指定します。
- **keySz** 使用する鍵のサイズ。
- **in** ラップ/アンラップする入力データを指定します。
- **inSz** 入力データのサイズ。
- **wrap** 要求された操作が鍵ラップの場合は 1、アンラップの場合は 0。
- **out** 出力バッファを指定します。
- **outSz** 出力バッファのサイズ。

Return: 成功時には、出力バッファに書き込まれたラップ/アンラップされた鍵のサイズを返す必要があります。戻り値 0 またはエラーコード (< 0) は失敗を示します。

C.37.4 Functions Documentation

C.37.4.1 function wc_PKCS7_InitWithCert

```
int wc_PKCS7_InitWithCert(
    wc_PKCS7 * pkcs7,
    byte * der,
    word32 derSz
)
```

この関数は、DER 形式の証明書で PKCS7 構造体を初期化します。空の PKCS7 構造体を初期化するには、cert に NULL を、certSz に 0 を渡すことができます。

Parameters:

- **pkcs7** デコードされた証明書を格納する PKCS7 構造体へのポインタ。
- **der** PKCS7 構造体を初期化するための DER 形式 ASN.1 証明書を含むバッファへのポインタ。
- **derSz** 証明書バッファのサイズ。

See: [wc_PKCS7_Free](#)

Return:

- 0 PKCS7 構造体の初期化に成功した場合に返されます。
- MEMORY_E XMMALLOC でメモリ割り当てエラーがある場合に返されます。
- ASN_PARSE_E 証明書ヘッダーの解析エラーがある場合に返されます。
- ASN_OBJECT_ID_E 証明書から暗号化タイプの解析エラーがある場合に返されます。
- ASN_EXPECT_0_E 証明書ファイルの暗号化仕様にフォーマットエラーがある場合に返されます。
- ASN_BEFORE_DATE_E 日付が証明書開始日より前の場合に返されます。
- ASN_AFTER_DATE_E 日付が証明書有効期限より後の場合に返されます。
- ASN_BITSTR_E 証明書からビット文字列の解析エラーがある場合に返されます。
- ECC_CURVE_OID_E 証明書から ECC 鍵の解析エラーがある場合に返されます。
- ASN_UNKNOWN_OID_E 証明書が不明な鍵オブジェクト ID を使用している場合に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS オプションが定義されておらず、証明書が V1 または V2 証明書の場合に返されます。
- BAD_FUNC_ARG 証明書拡張の処理エラーがある場合に返されます。
- ASN_CRIT_EXT_E 証明書の処理中に不明なクリティカル拡張に遭遇した場合に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます。
- ASN_SIG_CONFIRM_E 証明書署名の確認に失敗した場合に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA 名前制約によって許可されていない場合に返されます。
- ASN_NO_SIGNER_E 証明書の真正性を検証する CA 署名者がいない場合に返されます。

Example

```

wc_PKCS7 pkcs7;
byte derBuff[] = { }; // DER エンコードされた証明書で初期化
if ( wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff)) != 0 ) {
    // 証明書の pkcs7 形式への解析エラー
}

```

C.37.4.2 function wc_PKCS7_Free

```

void wc_PKCS7_Free(
    wc_PKCS7 * pkcs7
)

```

この関数は、PKCS7 初期化子によって割り当てられたメモリを解放します。

Parameters:

- **pkcs7** 解放する PKCS7 構造体へのポインタ。

See: [wc_PKCS7_InitWithCert](#)

Return: none 戻り値なし。

Example

```

PKCS7 pkcs7;
// PKCS7 オブジェクトを初期化して使用

wc_PKCS7_Free(pkcs7);

```

C.37.4.3 function wc_PKCS7_EncodeData

```
int wc_PKCS7_EncodeData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

この関数は、PKCS7 データコンテンツタイプをビルドし、PKCS7 構造体を解析可能な PKCS7 データパケットを含むバッファにエンコードします。

Parameters:

- **pkcs7** エンコードする PKCS7 構造体へのポインタ。
- **output** エンコードされた証明書を格納するバッファへのポインタ。
- **outputSz** 出力バッファで利用可能なサイズ。

See: [wc_PKCS7_InitWithCert](#)

Return:

- Success PKCS7 データをバッファに正常にエンコードした場合、PKCS7 構造体で解析されたインデックスまで返します。このインデックスは、出力バッファに書き込まれたバイト数にも対応します。
- BUFFER_E 指定されたバッファがエンコードされた証明書を保持するのに十分な大きさでない場合に返されます。

Example

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { }; // DER エンコードされた証明書で初期化
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// エンコードするメッセージとデータを更新
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... など

ret = wc_PKCS7_EncodeData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // 出力バッファへのエンコードエラー
}
```

C.37.4.4 function wc_PKCS7_EncodeSignedData

```
int wc_PKCS7_EncodeSignedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

この関数は、PKCS7 署名付きデータコンテンツタイプをビルドし、PKCS7 構造体を解析可能な PKCS7 署名付きデータパケットを含むバッファにエンコードします。

Parameters:

- **pkcs7** エンコードする PKCS7 構造体へのポインタ。
- **output** エンコードされた証明書を格納するバッファへのポインタ。
- **outputSz** 出力バッファで利用可能なサイズ。

See:

- [wc_PKCS7_InitWithCert](#)
- [wc_PKCS7_VerifySignedData](#)

Return:

- Success PKCS7 データをバッファに正常にエンコードした場合、PKCS7 構造体で解析されたインデックスまで返します。このインデックスは、出力バッファに書き込まれたバイト数にも対応します。
- BAD_FUNC_ARG 署名付きデータバケットを生成するために必要な 1 つ以上の要素が PKCS7 構造体に欠けている場合に返されます。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- PUBLIC_KEY_E 公開鍵の解析エラーがある場合に返されます。
- RSA_BUFFER_E バッファエラー、出力が小さすぎるか入力が大きすぎる場合に返されます。
- BUFFER_E 指定されたバッファがエンコードされた証明書を保持するのに十分な大きさでない場合に返されます。
- MP_INIT_E 署名の生成エラーがある場合に返される可能性があります。
- MP_READ_E 署名の生成エラーがある場合に返される可能性があります。
- MP_CMP_E 署名の生成エラーがある場合に返される可能性があります。
- MP_INVMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_EXPTMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MUL_E 署名の生成エラーがある場合に返される可能性があります。
- MP_ADD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MULMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_TO_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MEM 署名の生成エラーがある場合に返される可能性があります。

Example

```

PKCS7 pkcs7;
int ret;

byte data[] = {}; // 署名するデータで初期化
byte derBuff[] = { }; // DER エンコードされた証明書で初期化
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// エンコードするメッセージとデータを更新
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
pkcs7.hashOID = SHAh;
pkcs7.rng = &rng;
... など

ret = wc_PKCS7_EncodeSignedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // 出力バッファへのエンコードエラー
}

wc_PKCS7_Free(&pkcs7);

```


C.37.4.5 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
)
```

この関数は、PKCS7 署名付きデータコンテンツタイプをビルドし、PKCS7 構造体を解析可能な PKCS7 署名付きデータパケットを含むヘッダーとフッターバッファにエンコードします。これにはコンテンツは含まれません。データのハッシュを計算して提供する必要があります。

Parameters:

- **pkcs7** エンコードする PKCS7 構造体へのポインタ。
- **hashBuf** コンテンツデータの計算されたハッシュへのポインタ。
- **hashSz** ダイジェストのサイズ。
- **outputHead** エンコードされた証明書ヘッダーを格納するバッファへのポインタ。
- **outputHeadSz** 出力ヘッダーバッファのサイズが入力され、実際のサイズを返します。
- **outputFoot** エンコードされた証明書フッターを格納するバッファへのポインタ。
- **outputFootSz** 出力フッターバッファのサイズが入力され、実際のサイズを返します。

See:

- [wc_PKCS7_InitWithCert](#)
- [wc_PKCS7_VerifySignedData_ex](#)

Return:

- 0=Success
- BAD_FUNC_ARG 署名付きデータパケットを生成するために必要な 1 つ以上の要素が PKCS7 構造体に欠けている場合に返されます。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- PUBLIC_KEY_E 公開鍵の解析エラーがある場合に返されます。
- RSA_BUFFER_E バッファエラー、出力が小さすぎるか入力が大きすぎる場合に返されます。
- BUFFER_E 指定されたバッファがエンコードされた証明書を保持するのに十分な大きさでない場合に返されます。
- MP_INIT_E 署名の生成エラーがある場合に返される可能性があります。
- MP_READ_E 署名の生成エラーがある場合に返される可能性があります。
- MP_CMP_E 署名の生成エラーがある場合に返される可能性があります。
- MP_INVMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_EXPTMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MUL_E 署名の生成エラーがある場合に返される可能性があります。
- MP_ADD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MULMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_TO_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MEM 署名の生成エラーがある場合に返される可能性があります。

Example

```
PKCS7 pkcs7;
int ret;
byte derBuff[] = { }; // DER エンコードされた証明書で初期化
byte data[] = { }; // 署名するデータで初期化
```



```

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));
// エンコードするメッセージとデータを更新
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = NULL;
pkcs7.contentSz = dataSz;
pkcs7.hashOID = SHAh;
pkcs7.rng = &rng;
... など

// コンテンツのハッシュを計算
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) {
    // 出力バッファへのエンコードエラー
}

wc_PKCS7_Free(&pkcs7);

```

C.37.4.6 function wc_PKCS7_VerifySignedData

```

int wc_PKCS7_VerifySignedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz
)

```

この関数は、送信された PKCS7 署名付きデータメッセージを受け取り、証明書リストと証明書失効リストを抽出し、署名を検証します。抽出されたコンテンツを与えられた PKCS7 構造体に格納します。

Parameters:

- **pkcs7** 解析された証明書を格納する PKCS7 構造体へのポインタ。
- **pkiMsg** 検証およびデコードする署名付きメッセージを含むバッファへのポインタ。
- **pkiMsgSz** 署名付きメッセージのサイズ。

See:

- [wc_PKCS7_InitWithCert](#)
- [wc_PKCS7_EncodeSignedData](#)

Return:

- 0 メッセージから情報を正常に抽出した場合に返されます。
- BAD_FUNC_ARG 入力パラメータの 1 つが無効な場合に返されます。
- ASN_PARSE_E 与えられた pkiMsg の解析エラーがある場合に返されます。
- PKCS7_OID_E 与えられた pkiMsg が署名付きデータタイプでない場合に返されます。
- ASN_VERSION_E PKCS7 署名者情報がバージョン 1 でない場合に返されます。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- PUBLIC_KEY_E 公開鍵の解析エラーがある場合に返されます。
- RSA_BUFFER_E バッファエラー、出力が小さすぎるか入力が大きすぎる場合に返されます。
- BUFFER_E 指定されたバッファがエンコードされた証明書を保持するのに十分な大きさでない場合に返されます。
- MP_INIT_E 署名の生成エラーがある場合に返される可能性があります。
- MP_READ_E 署名の生成エラーがある場合に返される可能性があります。
- MP_CMP_E 署名の生成エラーがある場合に返される可能性があります。
- MP_INVMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_EXPTMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MUL_E 署名の生成エラーがある場合に返される可能性があります。
- MP_ADD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MULMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_TO_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MEM 署名の生成エラーがある場合に返される可能性があります。

Example

```
PKCS7 pkcs7;
int ret;
byte pkcs7Buff[] = {}; // PKCS7 署名

wc_PKCS7_InitWithCert(&pkcs7, NULL, 0);
// エンコードするメッセージとデータを更新
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... など

ret = wc_PKCS7_VerifySignedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // 出力バッファへのエンコードエラー
}

wc_PKCS7_Free(&pkcs7);
```

C.37.4.7 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
)
```

この関数は、ハッシュ/ヘッダー/フッターとして送信された PKCS7 署名付きデータメッセージを受け取り、証明書リストと証明書失効リストを抽出し、署名を検証します。抽出されたコンテンツを与えられた PKCS7 構造体に格納します。

Parameters:

- **pkcs7** 解析された証明書を格納する PKCS7 構造体へのポインタ。
- **hashBuf** コンテンツデータの計算されたハッシュへのポインタ。
- **hashSz** ダイジェストのサイズ。
- **pkiMsgHead** 検証およびデコードする署名付きメッセージヘッダーを含むバッファへのポインタ。
- **pkiMsgHeadSz** 署名付きメッセージヘッダーのサイズ。
- **pkiMsgFoot** 検証およびデコードする署名付きメッセージフッターを含むバッファへのポインタ。
- **pkiMsgFootSz** 署名付きメッセージフッターのサイズ。

See:

- [wc_PKCS7_InitWithCert](#)
- [wc_PKCS7_EncodeSignedData_ex](#)

Return:

- 0 メッセージから情報を正常に抽出した場合に返されます。
- BAD_FUNC_ARG 入力パラメータの 1 つが無効な場合に返されます。
- ASN_PARSE_E 与えられた pkiMsg の解析エラーがある場合に返されます。
- PKCS7_OID_E 与えられた pkiMsg が署名付きデータタイプでない場合に返されます。
- ASN_VERSION_E PKCS7 署名者情報がバージョン 1 でない場合に返されます。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- PUBLIC_KEY_E 公開鍵の解析エラーがある場合に返されます。
- RSA_BUFFER_E バッファエラー、出力が小さすぎるか入力が大きすぎる場合に返されます。
- BUFFER_E 指定されたバッファがエンコードされた証明書を保持するのに十分な大きさでない場合に返されます。
- MP_INIT_E 署名の生成エラーがある場合に返される可能性があります。
- MP_READ_E 署名の生成エラーがある場合に返される可能性があります。
- MP_CMP_E 署名の生成エラーがある場合に返される可能性があります。
- MP_INVMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_EXPTMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MUL_E 署名の生成エラーがある場合に返される可能性があります。
- MP_ADD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MULMOD_E 署名の生成エラーがある場合に返される可能性があります。
- MP_TO_E 署名の生成エラーがある場合に返される可能性があります。
- MP_MEM 署名の生成エラーがある場合に返される可能性があります。

Example

```
PKCS7 pkcs7;
int ret;
byte data[] = {}; // 署名するデータで初期化
byte pkcs7HeadBuff[] = {}; // PKCS7 ヘッダーで初期化
byte pkcs7FootBuff[] = {}; // PKCS7 フッターで初期化
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);
// エンコードするメッセージとデータを更新
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
```

```

pkcs7.content = NULL;
pkcs7.contentSz = dataSz;
pkcs7.rng = &rng;
... など

// コンテンツのハッシュを計算
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) {
    // 出力バッファへのエンコードエラー
}

wc_PKCS7_Free(&pkcs7);

```

C.37.4.8 function wc_PKCS7_SetAESKeyWrapUnwrapCb

```

int wc_PKCS7_SetAESKeyWrapUnwrapCb(
    wc_PKCS7 * pkcs7,
    CallbackAESKeyWrapUnwrap aesKeyWrapCb
)

```

カスタム AES 鍵ラップ/アンラップ操作を実行するために使用されるコールバック関数を設定します。

Parameters:

- **pkcs7** PKCS7 構造体へのポインタ。
- **aesKeyWrapCb** カスタム AES 鍵ラップ/アンラップ関数へのポインタ。

Returns:

- **0** コールバック関数が正常に設定されました。
- **BAD_FUNC_ARG** パラメータ **pkcs7** が NULL です。

C.37.4.9 function wc_PKCS7_EncodeEnvelopedData

```

int wc_PKCS7_EncodeEnvelopedData(
    wc_PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)

```

この関数は、PKCS7 エンベロープデータコンテンツタイプをビルドし、PKCS7 構造体を解析可能な PKCS7 エンベロープデータパケットを含むバッファにエンコードします。

Parameters:

- **pkcs7** エンコードする PKCS7 構造体へのポインタ。
- **output** エンコードされた証明書を格納するバッファへのポインタ。
- **outputSz** 出力バッファで利用可能なサイズ。

See:

- [wc_PKCS7_InitWithCert](#)
- [wc_PKCS7_DecodeEnvelopedData](#)

Return:

- Success エンベロープデータ形式でメッセージを正常にエンコードした場合、出力バッファに書き込まれたサイズを返します。
- BAD_FUNC_ARG: 入力パラメータの 1 つが無効な場合、または PKCS7 構造体に必要な要素が欠けている場合に返されます。
- ALGO_ID_E PKCS7 構造体がサポートされていないアルゴリズムタイプを使用している場合に返されます。現在、Desb と DES3b のみがサポートされています。
- BUFFER_E 指定された出力バッファが出力データを格納するのに小さすぎる場合に返されます。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- RNG_FAILURE_E 暗号化用の乱数生成器の初期化エラーがある場合に返されます。
- DRBG_FAILED 暗号化に使用される乱数生成器で数値の生成エラーがある場合に返されます。
- NOT_COMPILED_IN ECC 鍵を使用していて、wolfSSL が HAVE_X963_KDF サポートなしでビルドされている場合に返される可能性があります。

Example

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { }; // DER エンコードされた証明書で初期化
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// エンコードするメッセージとデータを更新
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... など

ret = wc_PKCS7_EncodeEnvelopedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret < 0 ) {
    // 出力バッファへのエンコードエラー
}
```

C.37.4.10 function wc_PKCS7_DecodeEnvelopedData

```
int wc_PKCS7_DecodeEnvelopedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

この関数は、PKCS7 エンベロープデータコンテンツタイプをアンラップして復号し、メッセージを output にデコードします。渡された PKCS7 オブジェクトの秘密鍵を使用してメッセージを復号します。

Parameters:

- **pkcs7** エンベロープデータパッケージをデコードするための秘密鍵を含む PKCS7 構造体へのポインタ。

- **pkiMsg** エンベロープデータパッケージを含むバッファへのポインタ。
- **pkiMsgSz** エンベロープデータパッケージのサイズ。
- **output** デコードされたメッセージを格納するバッファへのポインタ。
- **outputSz** 出力バッファで利用可能なサイズ。

See:

- [wc_PKCS7_InitWithCert](#)
- [wc_PKCS7_EncodeEnvelopedData](#)

Return:

- メッセージから情報を正常に抽出した場合、output に書き込まれたバイト数を返します。
- BAD_FUNC_ARG 入力パラメータの 1 つが無効な場合に返されます。
- ASN_PARSE_E 与えられた pkiMsg の解析エラーがある場合に返されます。
- PKCS7_OID_E 与えられた pkiMsg がエンベロープデータタイプでない場合に返されます。
- ASN_VERSION_E PKCS7 署名者情報がバージョン 0 でない場合に返されます。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- ALGO_ID_E PKCS7 構造体がサポートされていないアルゴリズムタイプを使用している場合に返されます。現在、暗号化には Desb と DES3b のみがサポートされており、署名生成には RSAk がサポートされています。
- PKCS7_RECIP_E エンベロープデータ内に提供された受信者と一致する受信者が見つからない場合に返されます。
- RSA_BUFFER_E バッファエラー、出力が小さすぎるか入力が大きすぎることによる RSA 署名検証中のエラーがある場合に返されます。
- MP_INIT_E 署名検証中にエラーがある場合に返される可能性があります。
- MP_READ_E 署名検証中にエラーがある場合に返される可能性があります。
- MP_CMP_E 署名検証中にエラーがある場合に返される可能性があります。
- MP_INVMOD_E 署名検証中にエラーがある場合に返される可能性があります。
- MP_EXPTMOD_E 署名検証中にエラーがある場合に返される可能性があります。
- MP_MOD_E 署名検証中にエラーがある場合に返される可能性があります。
- MP_MUL_E 署名検証中にエラーがある場合に返される可能性があります。
- MP_ADD_E 署名検証中にエラーがある場合に返される可能性があります。
- MP_MULMOD_E 署名検証中にエラーがある場合に返される可能性があります。
- MP_TO_E 署名検証中にエラーがある場合に返される可能性があります。
- MP_MEM 署名検証中にエラーがある場合に返される可能性があります。
- NOT_COMPILED_IN EnvelopedData が ECC 鍵を使用して暗号化されていて、wolfSSL が HAVE_X963_KDF サポートなしでビルドされている場合に返される可能性があります。

注意: EnvelopedData が ECC 鍵と KeyAgreementRecipientInfo 構造体を使用して暗号化されている場合、wolfcrypt 組み込みの AES 鍵ラップ/アンラップ機能を有効にするために HAVE_AES_KEYWRAP ビルドオプションを有効にするか、wc_PKCS7_SetAESKeyWrapUnwrapCb() を使用してカスタム AES 鍵ラップ/アンラップコールバックを設定する必要があります。これらのいずれも該当しない場合、復号は失敗します。

Example

```
PKCS7 pkcs7;
byte received[] = { }; // 受信したエンベロープデータメッセージで初期化
byte decoded[FOURK_BUF];
int decodedSz;

// 証明書で pkcs7 を初期化
// 鍵を更新
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
```

```
decodedSz = wc_PKCS7_DecodeEnvelopedData(&pkcs7, received, sizeof(received),
    decoded, sizeof(decoded));
```



```

if ( decodedSz < 0 ) {
    // メッセージのデコードエラー
}

```

C.37.4.11 function wc_PKCS7_GetEnvelopedDataKariRid

```

int wc_PKCS7_GetEnvelopedDataKariRid(
    const byte * in,
    word32 inSz,
    byte * out,
    word32 * outSz
)

```

この関数は、KeyAgreeRecipientInfo RecipientInfo オブジェクトを含む EnvelopedData パッケージから KeyAgreeRecipientIdentifier オブジェクトを抽出します。最初の RecipientInfo で見つかった最初の KeyAgreeRecipientIdentifier のみがコピーされます。この関数は、複数の RecipientInfo オブジェクトや KeyAgreeRecipientInfo 内の複数の RecipientEncryptedKey オブジェクトをサポートしていません。

Parameters:

- **in** EnvelopedData ContentInfo メッセージを含む入力バッファ。
- **inSz** 入力バッファのサイズ。
- **out** 出力バッファ。
- **outSz** 入力時の出力バッファサイズ、出力時に書き込まれたサイズ。

Return:

- 成功時に 0 を返します。
- BAD_FUNC_ARG 入力パラメータの 1 つが無効な場合に返されます。
- ASN_PARSE_E 入力メッセージの解析エラーがある場合に返されます。
- PKCS7_OID_E 入力メッセージがエンベロープデータタイプでない場合に返されます。
- BUFFER_E 出力バッファに十分なスペースがない場合に返されます。

C.37.4.12 function wc_PKCS7_DecodeEncryptedData

```

int wc_PKCS7_DecodeEncryptedData(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)

```

この関数は、PKCS7 暗号化データコンテンツタイプをアンラップして復号し、メッセージを output にデコードします。pkcs7->encryptionKey と pkcs7->encryptionKeySz を介して渡された PKCS7 オブジェクトの暗号化鍵を使用してメッセージを復号します。

Parameters:

- **pkcs7** 暗号化データパッケージをデコードするための暗号化鍵を含む PKCS7 構造体へのポインタ。
- **pkiMsg** 暗号化データパッケージを含むバッファへのポインタ。
- **pkiMsgSz** 暗号化データパッケージのサイズ。
- **output** デコードされたメッセージを格納するバッファへのポインタ。
- **outputSz** 出力バッファで利用可能なサイズ。

See: [wc_PKCS7_InitWithCert](#)

Return:

- メッセージから情報を正常に抽出した場合、output に書き込まれたバイト数を返します。

- BAD_FUNC_ARG 入力パラメータの 1 つが無効な場合に返されます。
- ASN_PARSE_E 与えられた pkiMsg の解析エラーがある場合に返されます。
- PKCS7_OID_E 与えられた pkiMsg が暗号化データタイプでない場合に返されます。
- ASN_VERSION_E PKCS7 署名者情報がバージョン 0 でない場合に返されます。
- MEMORY_E メモリ割り当てエラーがある場合に返されます。
- BUFFER_E 暗号化されたコンテンツのサイズが無効な場合に返されます。

Example

```
PKCS7 pkcs7;
byte received[] = { }; // 受信した暗号化データメッセージで初期化
byte decoded[FOURK_BUF];
int decodedSz;

// 証明書で pkcs7 を初期化
// 鍵を更新
pkcs7.encryptionKey = key;
pkcs7.encryptionKeySz = keySz;

decodedSz = wc_PKCS7_DecodeEncryptedData(&pkcs7, received,
    sizeof(received), decoded, sizeof(decoded));
if ( decodedSz < 0 ) {
    // メッセージのデコードエラー
}
```

C.37.4.13 function wc_PKCS7_DecodeEncryptedKeyPackage

```
int wc_PKCS7_DecodeEncryptedKeyPackage(
    wc_PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

この関数は、PKCS7 暗号化鍵パッケージコンテンツタイプをアンラップして復号し、メッセージを output にデコードします。ラップされたコンテンツタイプが EncryptedData の場合、pkcs7 入力構造体に暗号化鍵を設定する必要があります (pkcs7->encryptionKey と pkcs7->encryptionKeySz を介して)。ラップされたコンテンツタイプが EnvelopedData の場合、pkcs7 入力構造体に秘密鍵を設定する必要があります (pkcs7->privateKey と pkcs7->privateKeySz を介して)。AuthEnvelopedData のラップされたコンテンツタイプは現在サポートされていません。

Parameters:

- **pkcs7** 暗号化鍵パッケージをデコードするための秘密鍵または暗号化鍵を含む PKCS7 構造体へのポインタ。
- **pkiMsg** 暗号化鍵パッケージメッセージを含むバッファへのポインタ。
- **pkiMsgSz** 暗号化鍵パッケージメッセージのサイズ。
- **output** デコードされた出力を格納するバッファへのポインタ。
- **outputSz** 出力バッファで利用可能なサイズ。

See: [wc_PKCS7_InitWithCert](#)

Return:

- メッセージから情報を正常に抽出した場合、output に書き込まれたバイト数を返します。
- BAD_FUNC_ARG 入力パラメータの 1 つが無効な場合に返されます。

- ASN_PARSE_E 与えられた pkiMsg の解析エラーがある場合、またはラップされたコンテンツタイプが EncryptedData で、EncryptedData のサポートがコンパイルされていない場合（例: NO_PKCS7_ENCRYPTED_DATA が設定されている場合）に返されます。
- PKCS7_OID_E 与えられた pkiMsg が暗号化鍵パッケージデータタイプでない場合に返されます。

この関数は、ラップされたコンテンツタイプに応じて、wc_PKCS7_DecodeEnvelopedData() または wc_PKCS7_DecodeEncryptedData() を自動的に呼び出します。この関数は、ここにリストされているエラーコードに加えて、これらの関数のいずれかからのエラーコードを返す可能性があります。

Example

```
PKCS7 pkcs7;
byte received[] = { }; // 受信した暗号化データメッセージで初期化
byte decoded[FOURK_BUF];
int decodedSz;

// 証明書で pkcs7 を初期化
// 予想される EnvelopedData の鍵を更新 (例)
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;

decodedSz = wc_PKCS7_DecodeEncryptedKeyPackage(&pkcs7, received,
    sizeof(received), decoded, sizeof(decoded));
if ( decodedSz < 0 ) {
    // メッセージのデコードエラー
}
```

C.37.4.14 function wc_PKCS7_DecodeSymmetricKeyPackageAttribute

```
int wc_PKCS7_DecodeSymmetricKeyPackageAttribute(
    const byte * skip,
    word32 skipSz,
    size_t index,
    const byte ** attr,
    word32 * attrSz
)
```

この関数は、SymmetricKeyPackage 属性へのアクセスを提供します。

Parameters:

- **skip** SymmetricKeyPackage オブジェクトを含む入力バッファ。
- **skipSz** SymmetricKeyPackage オブジェクトのサイズ。
- **index** アクセスする属性のインデックス。
- **attr** 要求された属性オブジェクトへのポインタを格納するバッファ。
- **attrSz** 要求された属性オブジェクトのサイズを格納するバッファ。

Return:

- 0 要求された属性が正常に見つかりました。attr と attrSz 出力変数には、属性のアドレスとサイズが入力されます。属性は、skip 入力ポインタを介して渡されたのと同じバッファ内にあります。
- BAD_FUNC_ARG 入力パラメータの 1 つが無効です。
- ASN_PARSE_E 入力オブジェクトの解析中にエラーが発生しました。
- BAD_INDEX_E 要求された属性インデックスが無効でした。

C.37.4.15 function wc_PKCS7_DecodeSymmetricKeyPackageKey

```
int wc_PKCS7_DecodeSymmetricKeyPackageKey(  
    const byte * skp,  
    word32 skpSz,  
    size_t index,  
    const byte ** key,  
    word32 * keySz  
)
```

この関数は、SymmetricKeyPackage 鍵へのアクセスを提供します。

Parameters:

- **skp** SymmetricKeyPackage オブジェクトを含む入力バッファ。
- **skpSz** SymmetricKeyPackage オブジェクトのサイズ。
- **index** アクセスする鍵のインデックス。
- **key** 要求された鍵オブジェクトへのポインタを格納するバッファ。
- **keySz** 要求された鍵オブジェクトのサイズを格納するバッファ。

Return:

- 0 要求された鍵が正常に見つかりました。key と keySz 出力変数には、鍵のアドレスとサイズが入力されます。鍵は、skp 入力ポインタを介して渡されたのと同じバッファ内にあります。
- BAD_FUNC_ARG 入力パラメータの 1 つが無効です。
- ASN_PARSE_E 入力オブジェクトの解析中にエラーが発生しました。
- BAD_INDEX_E 要求された鍵インデックスが無効でした。

C.37.4.16 function wc_PKCS7_DecodeOneSymmetricKeyAttribute

```
int wc_PKCS7_DecodeOneSymmetricKeyAttribute(  
    const byte * osk,  
    word32 oskSz,  
    size_t index,  
    const byte ** attr,  
    word32 * attrSz  
)
```

この関数は、OneSymmetricKey 属性へのアクセスを提供します。

Parameters:

- **osk** OneSymmetricKey オブジェクトを含む入力バッファ。
- **oskSz** OneSymmetricKey オブジェクトのサイズ。
- **index** アクセスする属性のインデックス。
- **attr** 要求された属性オブジェクトへのポインタを格納するバッファ。
- **attrSz** 要求された属性オブジェクトのサイズを格納するバッファ。

Return:

- 0 要求された属性が正常に見つかりました。attr と attrSz 出力変数には、属性のアドレスとサイズが入力されます。属性は、osk 入力ポインタを介して渡されたのと同じバッファ内にあります。
- BAD_FUNC_ARG 入力パラメータの 1 つが無効です。
- ASN_PARSE_E 入力オブジェクトの解析中にエラーが発生しました。
- BAD_INDEX_E 要求された属性インデックスが無効でした。

C.37.4.17 function wc_PKCS7_DecodeOneSymmetricKeyKey

```
int wc_PKCS7_DecodeOneSymmetricKeyKey(  
    const byte * osk,  
    word32 oskSz,
```

```

    const byte ** key,
    word32 * keySz
)

```

この関数は、OneSymmetricKey 鍵へのアクセスを提供します。

Parameters:

- **osk** OneSymmetricKey オブジェクトを含む入力バッファ。
- **oskSz** OneSymmetricKey オブジェクトのサイズ。
- **key** 要求された鍵オブジェクトへのポインタを格納するバッファ。
- **keySz** 要求された鍵オブジェクトのサイズを格納するバッファ。

Return:

- 0 要求された鍵が正常に見つかりました。key と keySz 出力変数には、鍵のアドレスとサイズが入力されます。鍵は、osk 入力ポインタを介して渡されたのと同じバッファ内にあります。
- BAD_FUNC_ARG 入力パラメータの 1 つが無効です。
- ASN_PARSE_E 入力オブジェクトの解析中にエラーが発生しました。

C.37.5 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);

```

C.38 dox_comments/header_files-ja/poly1305.h

C.38.1 Functions

	Name
int	wc_Poly1305SetKey (Poly1305 * poly1305, const byte * key, word32 kySz) この関数は、Poly1305 コンテキスト構造体のキーを設定し、ハッシュ化のために初期化します。注意: wc_Poly1305Final でメッセージハッシュを生成した後、セキュリティを確保するために新しいキーを設定する必要があります。
int	wc_Poly1305Update (Poly1305 * poly1305, const byte * m, word32 bytes) この関数は、Poly1305 構造体でハッシュ化するメッセージを更新します。
int	wc_Poly1305Final (Poly1305 * poly1305, byte * tag) この関数は、入力メッセージのハッシュを計算し、結果を mac に格納します。この関数を呼び出した後、キーをリセットする必要があります。
int	wc_Poly1305_MAC (Poly1305 * ctx, const byte * additional, word32 addSz, const byte * input, word32 sz, byte * tag, word32 tagSz) キーがロードされた初期化済みの Poly1305 構造体を受け取り、最新の TLS AEAD パディングスキームを使用して MAC(タグ)を作成します。

C.38.2 Functions Documentation

C.38.2.1 function wc_Poly1305SetKey

```
int wc_Poly1305SetKey(
    Poly1305 * poly1305,
    const byte * key,
    word32 kySz
)
```

この関数は、Poly1305 コンテキスト構造体のキーを設定し、ハッシュ化のために初期化します。注意: wc_Poly1305Final でメッセージハッシュを生成した後、セキュリティを確保するために新しいキーを設定する必要があります。

Parameters:

- **ctx** 初期化する Poly1305 構造体へのポインタ
- **key** ハッシュ化に使用するキーを含むバッファへのポインタ
- **keySz** バッファ内のキーのサイズ。32 バイトである必要があります

See:

- [wc_Poly1305Update](#)
- [wc_Poly1305Final](#)

Return:

- 0 キーの設定と Poly1305 構造体の初期化に成功した場合に返されます
- BAD_FUNC_ARG 指定されたキーが 32 バイトでない場合、または Poly1305 コンテキストが NULL の場合に返されます

Example

```
Poly1305 enc;
byte key[] = { ハッシュ化に使用する32バイトのキーで初期化 };
wc_Poly1305SetKey(&enc, key, sizeof(key));
```

C.38.2.2 function wc_Poly1305Update

```
int wc_Poly1305Update(
    Poly1305 * poly1305,
    const byte * m,
    word32 bytes
)
```

この関数は、Poly1305 構造体でハッシュ化するメッセージを更新します。

Parameters:

- **ctx** ハッシュ化するメッセージを更新する Poly1305 構造体へのポインタ
- **m** ハッシュに追加するメッセージを含むバッファへのポインタ
- **bytes** ハッシュ化するメッセージのサイズ

See:

- [wc_Poly1305SetKey](#)
- [wc_Poly1305Final](#)

Return:

- 0 ハッシュ化するメッセージの更新に成功した場合に返されます
- BAD_FUNC_ARG Poly1305 構造体が NULL の場合に返されます

Example

```
Poly1305 enc;
byte key[] = { }; // 暗号化に使用する 32 バイトのキーで初期化
```

```

byte msg[] = { }; // ハッシュ化するメッセージで初期化
wc_Poly1305SetKey(&enc, key, sizeof(key));

if( wc_Poly1305Update(key, msg, sizeof(msg)) != 0 ) {
    // ハッシュ化するメッセージの更新エラー
}

```

C.38.2.3 function wc_Poly1305Final

```

int wc_Poly1305Final(
    Poly1305 * poly1305,
    byte * tag
)

```

この関数は、入力メッセージのハッシュを計算し、結果を mac に格納します。この関数を呼び出した後、キーをリセットする必要があります。

Parameters:

- **ctx** MAC を生成するために使用する Poly1305 構造体へのポインタ
- **mac** MAC を格納するバッファへのポインタ。POLY1305_DIGEST_SIZE(16 バイト) 幅である必要があります

See:

- [wc_Poly1305SetKey](#)
- [wc_Poly1305Update](#)

Return:

- 0 最終 MAC の計算に成功した場合に返されます
- BAD_FUNC_ARG Poly1305 構造体が NULL の場合に返されます

Example

```

Poly1305 enc;
byte mac[POLY1305_DIGEST_SIZE]; // 16 バイトの mac 用のスペース

byte key[] = { }; // 暗号化に使用する 32 バイトのキーで初期化

byte msg[] = { }; // ハッシュ化するメッセージで初期化
wc_Poly1305SetKey(&enc, key, sizeof(key));
wc_Poly1305Update(key, msg, sizeof(msg));

if ( wc_Poly1305Final(&enc, mac) != 0 ) {
    // 最終 MAC の計算エラー
}

```

C.38.2.4 function wc_Poly1305_MAC

```

int wc_Poly1305_MAC(
    Poly1305 * ctx,
    const byte * additional,
    word32 addSz,
    const byte * input,
    word32 sz,
    byte * tag,

```

```
    word32 tagSz
)
```

キーがロードされた初期化済みの Poly1305 構造体を受け取り、最新の TLS AEAD パディングスキームを使用して MAC(タグ)を作成します。

Parameters:

- **ctx** 使用する初期化済みの Poly1305 構造体
- **additional** 使用する追加データ
- **addSz** additional バッファのサイズ
- **input** タグを作成する入力バッファ
- **sz** 入力バッファのサイズ
- **tag** 作成されたタグを保持するバッファ
- **tagSz** 入力タグバッファのサイズ (少なくとも WC_POLY1305_MAC_SZ(16) である必要があります)

See:

- [wc_Poly1305SetKey](#)
- [wc_Poly1305Update](#)
- [wcPoly1305Final](#)

Return:

- 0 成功
- BAD_FUNC_ARG ctx、input、または tag が null の場合、または additional が null で addSz が 0 より大きい場合、または tagSz が WC_POLY1305_MAC_SZ 未満の場合に返されます。

Example

```
Poly1305 ctx;
byte key[] = { }; // ハッシュ化に使用する 32 バイトのキーで初期化
byte additional[] = { }; // 追加データで初期化
byte msg[] = { }; // メッセージで初期化
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)
{
    // エラーを処理
}
```

C.38.3 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);
```


C.39 dox_comments/header_files-ja/psa.h

C.39.1 Functions

	Name
int	wolfSSL_CTX_psa_enable (WOLFSSL_CTX * ctx) この関数は、指定されたコンテキストで PSA サポートを有効にします。
int	wolfSSL_set_psa_ctx (WOLFSSL * ssl, struct psa_ssl_ctx * ctx) この関数は、指定された SSL セッションの PSA コンテキストをセットアップします
void	wolfSSL_free_psa_ctx (struct psa_ssl_ctx * ctx) この関数は、PSA コンテキストによって使用されるリソースを解放します
int	wolfSSL_psa_set_private_key_id (struct psa_ssl_ctx * ctx, psa_key_id_t id) この関数は、SSL セッションで使用される秘密鍵を設定します

C.39.2 Functions Documentation

C.39.2.1 function wolfSSL_CTX_psa_enable

```
int wolfSSL_CTX_psa_enable(
    WOLFSSL_CTX * ctx
)
```

この関数は、指定されたコンテキストで PSA サポートを有効にします。

Parameters:

- **ctx** PSA サポートを有効にする必要がある WOLFSSL_CTX オブジェクトへのポインタ

See: **wolfSSL_set_psa_ctx**

Return:

- WOLFSSL_SUCCESS 成功時
- BAD_FUNC_ARG ctx == NULL の場合

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.39.2.2 function wolfSSL_set_psa_ctx

```
int wolfSSL_set_psa_ctx(
    WOLFSSL * ssl,
    struct psa_ssl_ctx * ctx
)
```


この関数は、指定された SSL セッションの PSA コンテキストをセットアップします

Parameters:

- **ssl** ctx を有効にする WOLFSSL へのポインタ
- **ctx** struct psa_ssl_ctx へのポインタ (SSL セッションごとに一意である必要があります)

See:

- [wolfSSL_psa_set_private_key_id](#)
- [wolfSSL_psa_free_psa_ctx](#)

Return:

- WOLFSSL_SUCCESS 成功時
- BAD_FUNC_ARG ssl または ctx が NULL の場合

この関数は、指定された SSL セッションへの TLS コールバック用の PSA コンテキストをセットアップします。セッションの終了時に、コンテキストによって使用されたリソースは、[wolfSSL_free_psa_ctx\(\)](#) を使用して解放する必要があります。

Example

```
// 新しい SSL セッションを作成
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// PSA コンテキストをセットアップ
ret = wolfSSL_set_psa_ctx(ssl, ctx);
```

C.39.2.3 function wolfSSL_free_psa_ctx

```
void wolfSSL_free_psa_ctx(
    struct psa_ssl_ctx * ctx
)
```

この関数は、PSA コンテキストによって使用されるリソースを解放します

Parameters:

- **ctx** struct psa_ssl_ctx へのポインタ

See: [wolfSSL_set_psa_ctx](#)

C.39.2.4 function wolfSSL_psa_set_private_key_id

```
int wolfSSL_psa_set_private_key_id(
    struct psa_ssl_ctx * ctx,
    psa_key_id_t id
)
```

この関数は、SSL セッションで使用される秘密鍵を設定します

Parameters:

- **ctx** struct psa_ssl_ctx へのポインタ
- **id** 秘密鍵として使用されるキーの PSA ID

See: [wolfSSL_set_psa_ctx](#)

Example

```
// 新しい SSL セッションを作成
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
psa_key_id_t key_id;

// キープロビジョニングは既に完了
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.39.3 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);
```

C.40 dox_comments/header_files-ja/pwdbased.h

C.40.1 Functions

	Name
int	wc_PBKDF1 (byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int hashType) この関数は、パスワードベース鍵導出関数 1(PBKDF1) を実装し、連結されたソルトを持つ入力パスワードをより安全な鍵に変換し、それを output に格納します。ユーザーはハッシュ関数として SHA と MD5 のいずれかを選択できます。
int	wc_PBKDF2 (byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int hashType) この関数は、パスワードベース鍵導出関数 2(PBKDF2) を実装し、連結されたソルトを持つ入力パスワードをより安全な鍵に変換し、それを output に格納します。ユーザーは、WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512 を含む、サポートされている HMAC ハッシュ関数のいずれかを選択できます。

	Name
int	wc_PKCS12_PBKDF (byte * output, const byte * passwd, int passLen, const byte * salt, int saltLen, int iterations, int kLen, int hashType, int id) この関数は、RFC 7292 付録 B に記述されているパスワードベース鍵導出関数 (PBKDF) を実装します。この関数は、連結されたソルトを持つ入力パスワードをより安全な鍵に変換し、それを output に格納します。ユーザーは、WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512 を含む、サポートされている HMAC ハッシュ関数のいずれかを選択できます。

C.40.2 Functions Documentation

C.40.2.1 function wc_PBKDF1

```
int wc_PBKDF1(  
    byte * output,  
    const byte * passwd,  
    int pLen,  
    const byte * salt,  
    int sLen,  
    int iterations,  
    int kLen,  
    int hashType  
)
```

この関数は、パスワードベース鍵導出関数 1(PBKDF1) を実装し、連結されたソルトを持つ入力パスワードをより安全な鍵に変換し、それを output に格納します。ユーザーはハッシュ関数として SHA と MD5 のいずれかを選択できます。

Parameters:

- **output** 生成された鍵を格納するバッファへのポインタ。少なくとも kLen の長さである必要があります
- **passwd** 鍵導出に使用するパスワードを含むバッファへのポインタ
- **pLen** 鍵導出に使用するパスワードの長さ
- **salt** 鍵導出に使用するソルトを含むバッファへのポインタ
- **sLen** ソルトの長さ
- **iterations** ハッシュを処理する回数
- **kLen** 導出される鍵の希望の長さ。選択されたハッシュのダイジェストサイズより長くしないでください
- **hashType** 使用するハッシュアルゴリズム。有効な選択肢は WC_MD5 と WC_SHA です

See:

- **wc_PBKDF2**
- **wc_PKCS12_PBKDF**

Return:

- 0 入力パスワードから鍵の導出に成功した場合に返されます

- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合 (有効なタイプは MD5 と SHA)、iterations が 1 未満の場合、または要求された鍵の長さ (kLen) が提供されたハッシュのハッシュ長より大きい場合に返されます
- MEMORY_E SHA または MD5 オブジェクトのメモリ割り当て中にエラーが発生した場合に返されます

Example

```
int ret;
byte key[WC_MD5_DIGEST_SIZE];
byte pass[] = { }; // パスワードで初期化
byte salt[] = { }; // ソルトで初期化

ret = wc_PBKDF1(key, pass, sizeof(pass), salt, sizeof(salt), 1000,
sizeof(key), WC_MD5);
if ( ret != 0 ) {
    // パスワードからの鍵導出エラー
}
```

C.40.2.2 function wc_PBKDF2

```
int wc_PBKDF2(
    byte * output,
    const byte * passwd,
    int plen,
    const byte * salt,
    int slen,
    int iterations,
    int klen,
    int hashType
)
```

この関数は、パスワードベース鍵導出関数 2(PBKDF2) を実装し、連結されたソルトを持つ入力パスワードをより安全な鍵に変換し、それを output に格納します。ユーザーは、WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512 を含む、サポートされている HMAC ハッシュ関数のいずれかを選択できます。

Parameters:

- **output** 生成された鍵を格納するバッファへのポインタ。klen の長さである必要があります
- **passwd** 鍵導出に使用するパスワードを含むバッファへのポインタ
- **plen** 鍵導出に使用するパスワードの長さ
- **salt** 鍵導出に使用するソルトを含むバッファへのポインタ
- **slen** ソルトの長さ
- **iterations** ハッシュを処理する回数
- **klen** 導出される鍵の希望の長さ
- **hashType** 使用するハッシュアルゴリズム。有効な選択肢は、WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512 です

See:

- [wc_PBKDF1](#)
- [wc_PKCS12_PBKDF](#)

Return:

- 0 入力パスワードから鍵の導出に成功した場合に返されます

- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合、または iterations が 1 未満の場合に返されます
- MEMORY_E HMAC オブジェクトのメモリ割り当て中にエラーが発生した場合に返されます

Example

```
int ret;
byte key[64];
byte pass[] = { }; // パスワードで初期化
byte salt[] = { }; // ソルトで初期化

ret = wc_PBKDF2(key, pass, sizeof(pass), salt, sizeof(salt), 2048, sizeof(key),
WC_SHA512);
if ( ret != 0 ) {
    // パスワードからの鍵導出エラー
}
```

C.40.2.3 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
)
```

この関数は、RFC 7292 付録 B に記述されているパスワードベース鍵導出関数 (PBKDF) を実装します。この関数は、連結されたソルトを持つ入力パスワードをより安全な鍵に変換し、それを output に格納します。ユーザーは、WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512 を含む、サポートされている HMAC ハッシュ関数のいずれかを選択できます。

Parameters:

- **output** 生成された鍵を格納するバッファへのポインタ。kLen の長さである必要があります
- **passwd** 鍵導出に使用するパスワードを含むバッファへのポインタ
- **passLen** 鍵導出に使用するパスワードの長さ
- **salt** 鍵導出に使用するソルトを含むバッファへのポインタ
- **saltLen** ソルトの長さ
- **iterations** ハッシュを処理する回数
- **kLen** 導出される鍵の希望の長さ
- **hashType** 使用するハッシュアルゴリズム。有効な選択肢は、WC_MD5、WC_SHA、WC_SHA256、WC_SHA384、WC_SHA512、WC_SHA3_224、WC_SHA3_256、WC_SHA3_384、または WC_SHA3_512 です
- **id** 鍵生成の目的を示すバイト識別子。鍵出力を多様化するために使用され、次のように割り当てる必要があります。ID=1: 疑似乱数ビットは、暗号化または復号を実行するための鍵材料として使用されます。ID=2: 疑似乱数ビットは、暗号化または復号のための IV(初期値) として使用されます。ID=3: 疑似乱数ビットは、MAC 処理のための完全性鍵として使用されます。

See:

- [wc_PBKDF1](#)
- [wc_PBKDF2](#)

Return:

- 0 入力パスワードから鍵の導出に成功した場合に返されます
- BAD_FUNC_ARG 無効なハッシュタイプが指定された場合、iterations が 1 未満の場合、または要求された鍵の長さ (kLen) が提供されたハッシュのハッシュ長より大きい場合に返されます
- MEMORY_E メモリ割り当て中にエラーが発生した場合に返されます
- MP_INIT_E 鍵生成中にエラーが発生した場合に返される可能性があります
- MP_READ_E 鍵生成中にエラーが発生した場合に返される可能性があります
- MP_CMP_E 鍵生成中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E 鍵生成中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E 鍵生成中にエラーが発生した場合に返される可能性があります
- MP_MOD_E 鍵生成中にエラーが発生した場合に返される可能性があります
- MP_MUL_E 鍵生成中にエラーが発生した場合に返される可能性があります
- MP_ADD_E 鍵生成中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E 鍵生成中にエラーが発生した場合に返される可能性があります
- MP_TO_E 鍵生成中にエラーが発生した場合に返される可能性があります
- MP_MEM 鍵生成中にエラーが発生した場合に返される可能性があります

Example

```
int ret;
byte key[64];
byte pass[] = { }; // パスワードで初期化
byte salt[] = { }; // ソルトで初期化

ret = wc_PKCS12_PBKDF(key, pass, sizeof(pass), salt, sizeof(salt), 2048,
    sizeof(key), SHA512, 1);
if ( ret != 0 ) {
    // パスワードからの鍵導出エラー
}
```

C.40.3 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);
```

C.41 dox_comments/header_files-ja/quic.h**C.41.1 Functions**

	Name
int	wolfSSL_CTX_set_quic_method (WOLFSSL_CTX * ctx, const WOLFSSL_QUIC_METHOD * quic_method) 必要な 4 つのコールバックを提供することにより、WOLFSSL_CTX およびすべての派生 WOLFSSL インスタンスに対して QUIC プロトコルを有効化します。CTX は TLSv1.3 である必要があります。
int	wolfSSL_set_quic_method (WOLFSSL * ssl, const WOLFSSL_QUIC_METHOD * quic_method) 必要な 4 つのコールバックを提供することにより、WOLFSSL インスタンスに対して QUIC プロトコルを有効化します。WOLFSSL は TLSv1.3 である必要があります。
int	wolfSSL_is_quic (WOLFSSL * ssl) WOLFSSL インスタンスで QUIC が有効化されているかどうかを確認します。
WOLFSSL_ENCRYPTION_LEVEL	wolfSSL_quic_read_level (const WOLFSSL * ssl) 現在使用中の読み取り用の暗号化レベルを決定します。WOLFSSL インスタンスが QUIC を使用している場合にのみ意味があります。
WOLFSSL_ENCRYPTION_LEVEL	wolfSSL_quic_write_level (const WOLFSSL * ssl) 現在使用中の書き込み用の暗号化レベルを決定します。WOLFSSL インスタンスが QUIC を使用している場合にのみ意味があります。
void	wolfSSL_set_quic_use_legacy_codepoint (WOLFSSL * ssl, int use_legacy) 使用する QUIC バージョンを設定します。これ呼び出さない場合、WOLFSSL は両方 (draft-27 と v1) をサーバーに提供し、またはクライアントから両方を受け入れて最新のものをネゴシエートします。
void	wolfSSL_set_quic_transport_version (WOLFSSL * ssl, int version) 使用する QUIC バージョンを設定します。
int	wolfSSL_get_quic_transport_version (const WOLFSSL * ssl) 設定された QUIC バージョンを取得します。
int	wolfSSL_set_quic_transport_params (WOLFSSL * ssl, const uint8_t * params, size_t params_len) 使用する QUIC トランスポートパラメータを設定します。
int	wolfSSL_get_peer_quic_transport_version (const WOLFSSL * ssl) ネゴシエートされた QUIC トランスポートバージョンを取得します。これは、ピアからの該当する TLS 拡張が確認された後に呼び出された場合にのみ意味のある結果を提供します。
void	wolfSSL_get_peer_quic_transport_params (const WOLFSSL * ssl, const uint8_t ** out_params, size_t * out_params_len) ネゴシエートされた QUIC トランスポートパラメータを取得します。これは、ピアからの該当する TLS 拡張が確認された後に呼び出された場合にのみ意味のある結果を提供します。

	Name
void	wolfSSL_set_quic_early_data_enabled (WOLFSSL * ssl, int enabled) Early Data が有効かどうかを設定します。サーバーがクライアントにこれを通知することを目的としています。
size_t	wolfSSL_quic_max_handshake_flight_len (const WOLFSSL * ssl, WOLFSSL_ENCRYPTION_LEVEL level) 与えられた暗号化レベルで「インフライト」であるべき、つまり未確認であるべきデータの量についてのアドバイスを取得します。これは、WOLFSSL インスタンスがバッファリングする準備ができていないデータの量です。
int	wolfSSL_provide_quic_data (WOLFSSL * ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t * data, size_t len) 復号された CRYPTO データを WOLFSSL インスタンスに渡してさらに処理します。呼び出し間の暗号化レベルは増加することのみが許可され、暗号化レベルの変更が受け入れられる前にデータレコードが完全であることもチェックされます。
WOLFSSL_API int	wolfSSL_process_quic_post_handshake (WOLFSSL * ssl) ハンドシェイクが完了した後に提供された CRYPTO レコードを処理します。それ以前に呼び出された場合は失敗します。
int	wolfSSL_quic_read_write (WOLFSSL * ssl) ハンドシェイク中またはハンドシェイク後に提供された CRYPTO レコードを処理します。ハンドシェイクがまだ完了していない場合はハンドシェイクを進行させ、そうでない場合は wolfSSL_process_quic_post_handshake() のように動作します。
const WOLFSSL_EVP_CIPHER *	wolfSSL_quic_get_aead (WOLFSSL * ssl) TLS ハンドシェイクでネゴシエートされた AEAD 暗号を取得します。
int	wolfSSL_quic_aead_is_gcm (const WOLFSSL_EVP_CIPHER * aead_cipher) AEAD 暗号が GCM であるかどうかを確認します。
int	wolfSSL_quic_aead_is_ccm (const WOLFSSL_EVP_CIPHER * aead_cipher) AEAD 暗号が CCM であるかどうかを確認します。
int	wolfSSL_quic_aead_is_chacha20 (const WOLFSSL_EVP_CIPHER * aead_cipher) AEAD 暗号が CHACHA20 であるかどうかを確認します。
WOLFSSL_API size_t	wolfSSL_quic_get_aead_tag_len (const WOLFSSL_EVP_CIPHER * aead_cipher) AEAD 暗号のタグ長を決定します。
WOLFSSL_API const WOLFSSL_EVP_MD *	wolfSSL_quic_get_md (WOLFSSL * ssl) TLS ハンドシェイクでネゴシエートされたメッセージダイジェストを決定します。
const WOLFSSL_EVP_CIPHER *	wolfSSL_quic_get_hp (WOLFSSL * ssl) TLS ハンドシェイクでネゴシエートされたヘッダー保護暗号を決定します。

	Name
WOLFSSL_EVP_CIPHER_CTX *	wolfSSL_quic_crypt_new (const WOLFSSL_EVP_CIPHER * cipher, const uint8_t * key, const uint8_t * iv, int encrypt) 暗号化/復号用の暗号コンテキストを作成します。
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) 与えられたコンテキストで平文を暗号化します。
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) 与えられたコンテキストで暗号文を復号します。
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) 疑似ランダム鍵を抽出します。
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) 疑似ランダム鍵を新しい鍵に拡張します。
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) 疑似ランダム鍵を拡張および抽出します。

C.41.2 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 ハンドシェイク中にシークレットが生成されたときに呼び出されるコールバック。QUIC プロトコルハンドラはパケットの暗号化/復号を実行するため、early_data/handshake/application レベルのネゴシエートされたシークレットが必要です。
int()(WOLFSSL ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t *data, size_t len)	add_handshake_data ハンドシェイク CRYPTO データをピアに転送するために呼び出されるコールバック。この方法で転送されるデータは暗号化されていません。これを行うのは QUIC プロトコル実装の役割です。使用するシークレットは、指定された暗号化レベルによって決定されます。
int()(WOLFSSL ssl)	flush_flight 送信するデータのアドバイザリーフラッシュのために呼び出されるコールバック。
int()(WOLFSSL ssl, WOLFSSL_ENCRYPTION_LEVEL level, uint8_t *alert)	send_alert 処理中に SSL アラートが発生したときに呼び出されるコールバック。

C.41.3 Functions Documentation

C.41.3.1 function wolfSSL_CTX_set_quic_method

```
int wolfSSL_CTX_set_quic_method(  
    WOLFSSL_CTX * ctx,  
    const WOLFSSL_QUIC_METHOD * quic_method  
)
```

必要な4つのコールバックを提供することにより、WOLFSSL_CTX およびすべての派生 WOLFSSL インスタンスに対して QUIC プロトコルを有効化します。CTX は TLSv1.3 である必要があります。

Parameters:

- **ctx** - [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **quic_method** - コールバック構造体。

See:

- [wolfSSL_is_quic](#)
- [wolfSSL_set_quic_method](#)

Return: WOLFSSL_SUCCESS 成功した場合。

渡された quic_method は、SSL インスタンスよりも長い寿命を持つ必要があります。コピーされません。すべてのコールバックを提供する必要があります。

C.41.3.2 function wolfSSL_set_quic_method

```
int wolfSSL_set_quic_method(  
    WOLFSSL * ssl,  
    const WOLFSSL_QUIC_METHOD * quic_method  
)
```

必要な4つのコールバックを提供することにより、WOLFSSL インスタンスに対して QUIC プロトコルを有効化します。WOLFSSL は TLSv1.3 である必要があります。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **quic_method** - コールバック構造体。

See:

- [wolfSSL_is_quic](#)
- [wolfSSL_CTX_set_quic_method](#)

Return: WOLFSSL_SUCCESS 成功した場合。

渡された quic_method は、SSL インスタンスよりも長い寿命を持つ必要があります。コピーされません。すべてのコールバックを提供する必要があります。

C.41.3.3 function wolfSSL_is_quic

```
int wolfSSL_is_quic(  
    WOLFSSL * ssl  
)
```

WOLFSSL インスタンスで QUIC が有効化されているかどうかを確認します。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_CTX_quic_method`
- `wolfSSL_CTX_set_quic_method`

Return: 1 WOLFSSL が QUIC を使用している場合。

C.41.3.4 function `wolfSSL_quic_read_level`

```
WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_read_level(  
    const WOLFSSL * ssl  
)
```

現在使用中の読み取り用の暗号化レベルを決定します。WOLFSSL インスタンスが QUIC を使用している場合にのみ意味があります。

Parameters:

- `ssl` - `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_quic_write_level`

Return: 暗号化レベル。

有効なレベルは、データをやり取りする際に常にパラメータです。ピアからのデータは、この関数で報告されるレベル以外のレベルで到着する可能性があります。

C.41.3.5 function `wolfSSL_quic_write_level`

```
WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_write_level(  
    const WOLFSSL * ssl  
)
```

現在使用中の書き込み用の暗号化レベルを決定します。WOLFSSL インスタンスが QUIC を使用している場合にのみ意味があります。

Parameters:

- `ssl` - `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_quic_read_level`

Return: 暗号化レベル。

有効なレベルは、データをやり取りする際に常にパラメータです。ピアからのデータは、この関数で報告されるレベル以外のレベルで到着する可能性があります。

C.41.3.6 function `wolfSSL_set_quic_use_legacy_codepoint`

```
void wolfSSL_set_quic_use_legacy_codepoint(  
    WOLFSSL * ssl,  
    int use_legacy  
)
```

使用する QUIC バージョンを設定します。これ呼び出さない場合、WOLFSSL は両方 (draft-27 と v1) をサーバーに提供し、またはクライアントから両方を受け入れて最新のものをネゴシエートします。

Parameters:

- `ssl` - `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- `use_legacy` - draft-27 を使用する場合は true、QUICv1 のみを使用する場合は 0。

See: [wolfSSL_set_quic_transport_version](#)

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.7 function wolfSSL_set_quic_transport_version

```
void wolfSSL_set_quic_transport_version(  
    WOLFSSL * ssl,  
    int version  
)
```

使用する QUIC バージョンを設定します。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **version** - QUIC バージョンに定義された TLS 拡張。

See: [wolfSSL_set_quic_use_legacy_codepoint](#)

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.8 function wolfSSL_get_quic_transport_version

```
int wolfSSL_get_quic_transport_version(  
    const WOLFSSL * ssl  
)
```

設定された QUIC バージョンを取得します。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_set_quic_use_legacy_codepoint](#)
- [wolfSSL_set_quic_transport_version](#)

Return: 設定されたバージョンの TLS 拡張。

C.41.3.9 function wolfSSL_set_quic_transport_params

```
int wolfSSL_set_quic_transport_params(  
    WOLFSSL * ssl,  
    const uint8_t * params,  
    size_t params_len  
)
```

使用する QUIC トランスポートパラメータを設定します。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **params** - 使用するパラメータバイト。
- **params_len** - パラメータの長さ。

See:

- [wolfSSL_set_quic_use_legacy_codepoint](#)
- [wolfSSL_set_quic_transport_version](#)

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.10 function wolfSSL_get_peer_quic_transport_version

```
int wolfSSL_get_peer_quic_transport_version(  
    const WOLFSSL * ssl  
)
```

ネゴシエートされた QUIC トランスポートバージョンを取得します。これは、ピアからの該当する TLS 拡張が確認された後に呼び出された場合にのみ意味のある結果を提供します。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_set_quic_use_legacy_codepoint](#)
- [wolfSSL_set_quic_transport_version](#)

Return: ネゴシエートされたバージョンまたは-1。

C.41.3.11 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  
)
```

ネゴシエートされた QUIC トランスポートパラメータを取得します。これは、ピアからの該当する TLS 拡張が確認された後に呼び出された場合にのみ意味のある結果を提供します。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **out_params** - ピアによって送信されたパラメータ、利用できない場合は NULL に設定されます。
- **out_params_len** - ピアによって送信されたパラメータの長さ、利用できない場合は 0 に設定されます。

See: [wolfSSL_get_peer_quic_transport_version](#)

C.41.3.12 function wolfSSL_set_quic_early_data_enabled

```
void wolfSSL_set_quic_early_data_enabled(  
    WOLFSSL * ssl,  
    int enabled  
)
```

Early Data が有効かどうかを設定します。サーバーがクライアントにこれを通知することを目的としています。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **enabled** - early data が有効な場合は!= 0。

C.41.3.13 function wolfSSL_quic_max_handshake_flight_len

```
size_t wolfSSL_quic_max_handshake_flight_len(  
    const WOLFSSL * ssl,  
    WOLFSSL_ENCRYPTION_LEVEL level  
)
```

与えられた暗号化レベルで「インフライト」であるべき、つまり未確認であるべきデータの量についてのアドバイスを取得します。これは、WOLFSSL インスタンスがバッファリングする準備ができていないデータの量です。

Parameters:

- **ssl** - `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **level** - 問い合わせる暗号化レベル。

Return: 推奨される最大インフライトデータ。

C.41.3.14 function wolfSSL_provide_quic_data

```
int wolfSSL_provide_quic_data(  
    WOLFSSL * ssl,  
    WOLFSSL_ENCRYPTION_LEVEL level,  
    const uint8_t * data,  
    size_t len  
)
```

復号された CRYPTO データを WOLFSSL インスタンスに渡してさらに処理します。呼び出し間の暗号化レベルは増加することのみが許可され、暗号化レベルの変更が受け入れられる前にデータレコードが完全であることもチェックされます。

Parameters:

- **ssl** - `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **level** - データが暗号化されていたレベル。
- **data** - データ自体。
- **len** - データの長さ。

See:

- `wolfSSL_process_quic_post_handshake`
- `wolfSSL_quic_read_write`
- `wolfSSL_accept`
- `wolfSSL_connect`

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.15 function wolfSSL_process_quic_post_handshake

```
WOLFSSL_API int wolfSSL_process_quic_post_handshake(  
    WOLFSSL * ssl  
)
```

ハンドシェイクが完了した後に提供された CRYPTO レコードを処理します。それ以前に呼び出された場合は失敗します。

Parameters:

- **ssl** - `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_provide_quic_data`
- `wolfSSL_quic_read_write`
- `wolfSSL_accept`
- `wolfSSL_connect`

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.16 function wolfSSL_quic_read_write

```
int wolfSSL_quic_read_write(  
    WOLFSSL * ssl  
)
```

ハンドシェイク中またはハンドシェイク後に提供された CRYPTO レコードを処理します。ハンドシェイクがまだ完了していない場合はハンドシェイクを進行させ、そうでない場合は wolfSSL_process_quic_post_handshake() のように動作します。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_provide_quic_data
- wolfSSL_quic_read_write
- wolfSSL_accept
- wolfSSL_connect

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.17 function wolfSSL_quic_get_aead

```
const WOLFSSL_EVP_CIPHER * wolfSSL_quic_get_aead(  
    WOLFSSL * ssl  
)
```

TLS ハンドシェイクでネゴシエートされた AEAD 暗号を取得します。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

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: ネゴシエートされた暗号、または決定されていない場合は NULL。

C.41.3.18 function wolfSSL_quic_aead_is_gcm

```
int wolfSSL_quic_aead_is_gcm(  
    const WOLFSSL_EVP_CIPHER * aead_cipher  
)
```

AEAD 暗号が GCM であるかどうかを確認します。

Parameters:

- **cipher** - 暗号。

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: AEAD 暗号が GCM の場合は!= 0。

C.41.3.19 function `wolfSSL_quic_aead_is_ccm`

```
int wolfSSL_quic_aead_is_ccm(  
    const WOLFSSL_EVP_CIPHER * aead_cipher  
)
```

AEAD 暗号が CCM であるかどうかを確認します。

Parameters:

- **cipher** - 暗号。

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: AEAD 暗号が CCM の場合は!= 0。

C.41.3.20 function `wolfSSL_quic_aead_is_chacha20`

```
int wolfSSL_quic_aead_is_chacha20(  
    const WOLFSSL_EVP_CIPHER * aead_cipher  
)
```

AEAD 暗号が CHACHA20 であるかどうかを確認します。

Parameters:

- **cipher** - 暗号。

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: AEAD 暗号が CHACHA20 の場合は!= 0。

C.41.3.21 function wolfSSL_quic_get_aead_tag_len

```
WOLFSSL_API size\_t wolfSSL_quic_get_aead_tag_len(  
    const WOLFSSL_EVP_CIPHER * aead_cipher  
)
```

AEAD 暗号のタグ長を決定します。

Parameters:

- **cipher** - 暗号。

See: [wolfSSL_quic_get_aead](#)

Return: AEAD 暗号のタグ長。

C.41.3.22 function wolfSSL_quic_get_md

```
WOLFSSL_API const WOLFSSL_EVP_MD * wolfSSL_quic_get_md(  
    WOLFSSL * ssl  
)
```

TLS ハンドシェイクでネゴシエートされたメッセージダイジェストを決定します。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_quic_get_aead](#)
- [wolfSSL_quic_get_hp](#)

Return: TLS ハンドシェイクでネゴシエートされたメッセージダイジェスト。

C.41.3.23 function wolfSSL_quic_get_hp

```
const WOLFSSL_EVP_CIPHER * wolfSSL_quic_get_hp(  
    WOLFSSL * ssl  
)
```

TLS ハンドシェイクでネゴシエートされたヘッダー保護暗号を決定します。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_quic_get_aead](#)
- [wolfSSL_quic_get_md](#)

Return: TLS ハンドシェイクでネゴシエートされたヘッダー保護暗号。

C.41.3.24 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  
)
```

暗号化/復号用の暗号コンテキストを作成します。

Parameters:

- **cipher** - コンテキストで使用する暗号。
- **key** - コンテキストで使用する鍵。
- **iv** - コンテキストで使用する iv。
- **encrypt** - 暗号化の場合は!= 0、そうでない場合は復号。

See:

- [wolfSSL_quic_get_aead](#)
- [wolfSSL_quic_get_hp](#)
- [wolfSSL_quic_aead_encrypt](#)
- [wolfSSL_quic_aead_decrypt](#)

Return: 作成されたコンテキスト、またはエラーの場合は NULL。

C.41.3.25 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  
)
```

与えられたコンテキストで平文を暗号化します。

Parameters:

- **dest** - 暗号化されたデータを書き込む宛先。
- **aead_ctx** - 使用する暗号コンテキスト。
- **plain** - 暗号化する平文データ。
- **plainlen** - 平文データの長さ。
- **iv** - 使用する iv。
- **aad** - 使用する aad。
- **aadlen** - aad の長さ。

See:

- [wolfSSL_quic_get_aead](#)
- [wolfSSL_quic_get_hp](#)
- [wolfSSL_quic_crypt_new](#)
- [wolfSSL_quic_aead_decrypt](#)

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.26 function wolfSSL_quic_aead_decrypt

```
int wolfSSL_quic_aead_decrypt(  
    uint8_t * dest,  
    WOLFSSL_EVP_CIPHER_CTX * ctx,  
    const uint8_t * enc,  
    size_t encrlen,  
    const uint8_t * iv,  
    const uint8_t * aad,  
    size_t aadrlen  
)
```

与えられたコンテキストで暗号文を復号します。

Parameters:

- **dest** - 平文を書き込む宛先。
- **ctx** - 使用する暗号コンテキスト。
- **enc** - 復号する暗号化データ。
- **encrlen** - 暗号化データの長さ。
- **iv** - 使用する iv。
- **aad** - 使用する aad。
- **aadrlen** - aad の長さ。

See:

- [wolfSSL_quic_get_aead](#)
- [wolfSSL_quic_get_hp](#)
- [wolfSSL_quic_crypt_new](#)
- [wolfSSL_quic_aead_encrypt](#)

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.27 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  
)
```

疑似ランダム鍵を抽出します。

Parameters:

- **dest** - 鍵を書き込む宛先。
- **md** - 使用するメッセージダイジェスト。
- **secret** - 使用するシークレット。
- **secretlen** - シークレットの長さ。
- **salt** - 使用するソルト。
- **saltlen** - ソルトの長さ。

See:

- [wolfSSL_quic_hkdf_expand](#)
- [wolfSSL_quic_hkdf](#)

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.28 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  
)
```

疑似ランダム鍵を新しい鍵に拡張します。

Parameters:

- **dest** - 鍵を書き込む宛先。
- **destlen** - 拡張する鍵の長さ。
- **md** - 使用するメッセージダイジェスト。
- **secret** - 使用するシークレット。
- **secretlen** - シークレットの長さ。
- **info** - 使用する情報。
- **infolen** - 情報の長さ。

See:

- [wolfSSL_quic_hkdf_extract](#)
- [wolfSSL_quic_hkdf](#)

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.29 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  
)
```

疑似ランダム鍵を拡張および抽出します。

Parameters:

- **dest** - 鍵を書き込む宛先。
- **destlen** - 鍵の長さ。
- **md** - 使用するメッセージダイジェスト。
- **secret** - 使用するシークレット。
- **secretlen** - シークレットの長さ。
- **salt** - 使用するソルト。
- **saltlen** - ソルトの長さ。
- **info** - 使用する情報。
- **infolen** - 情報の長さ。

See:

- [wolfSSL_quic_hkdf_extract](#)
- [wolfSSL_quic_hkdf_expand](#)

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.4 Attributes Documentation

C.41.4.1 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;
```

ハンドシェイク中にシークレットが生成されたときに呼び出されるコールバック。QUIC プロトコルハンドラはパケットの暗号化/復号を実行するため、early_data/handshake/application レベルのネゴシエートされたシークレットが必要です。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **level** - シークレットが対応する暗号化レベル。
- **read_secret** - 与えられたレベルでの復号に使用されるシークレット、NULL の場合があります。
- **write_secret** - 与えられたレベルでの暗号化に使用されるシークレット、NULL の場合があります。
- **secret_len** - シークレットの長さ。

See: [wolfSSL_set_quic_method](#)

Return: 1 成功時、0 失敗時。

このコールバックはハンドシェイク中に数回呼び出されます。読み取りまたは書き込みシークレットの両方、または一方のみが提供される場合があります。これは、与えられた暗号化レベルがすでに有効になっていることを意味するものではありません。

C.41.4.2 variable add_handshake_data

```
int(*) (WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t *data,
↪ size_t len) add_handshake_data;
```

ハンドシェイク CRYPTO データをピアに転送するために呼び出されるコールバック。この方法で転送されるデータは暗号化されていません。これを行うのは QUIC プロトコル実装の役割です。使用するシークレットは、指定された暗号化レベルによって決定されます。

Parameters:

- **ssl** - [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **level** - データの暗号化に使用する暗号化レベル。
- **data** - データ自体。
- **len** - データの長さ。

See: [wolfSSL_set_quic_method](#)

Return: 1 成功時、0 失敗時。

このコールバックは、ハンドシェイクまたはポストハンドシェイク処理中に数回呼び出される場合があります。データは完全な CRYPTO レコードをカバーする場合もありますが、部分的な場合もあります。ただし、別の暗号化レベルを使用する前に、コールバックはすべてのレコードデータを受信します。

C.41.4.3 variable flush_flight

```
int(*) (WOLFSSL *ssl) flush_flight;
```

送信するデータのアドバイザリーフラッシュのために呼び出されるコールバック。

Parameters:

- **ssl** - `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_set_quic_method`

Return: 1 成功時、0 失敗時。

C.41.4.4 variable send_alert

```
int (*)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level, uint8_t alert) send_alert;
```

処理中に SSL アラートが発生したときに呼び出されるコールバック。

Parameters:

- **ssl** - `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **level** - アラートが発生したときに有効だった暗号化レベル。
- **alert** - エラー。

See: `wolfSSL_set_quic_method`

Return: 1 成功時、0 失敗時。

C.41.5 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);

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.42 dox_comments/header_files-ja/random.h

C.42.1 Functions

	Name
int	wc_InitNetRandom (const char * configFile, wnr_hmac_key hmac_cb, int timeout) グローバル Whitewood netRandom コンテキストを初期化します
int	wc_FreeNetRandom (void) グローバル Whitewood netRandom コンテキストを解放します。
int	wc_InitRng (WC_RNG * rng)rng 用のシード (OS から) と鍵暗号を取得します。rng->drbg(決定論的乱数ビット生成器) が割り当てられます (wc_FreeRng で割り当て解除する必要があります)。これはブロッキング操作です。
int	wc_RNG_GenerateBlock (WC_RNG * rng, byte * b, word32 sz) 疑似乱数データの sz バイトを output にコピーします。必要に応じて rng を再シードします (ブロッキング)。
WC_RNG *	wc_rng_new (byte * nonce, word32 nonceSz, void * heap) 新しい WC_RNG 構造体を作成します。
int	wc_FreeRng (WC_RNG * rng)drbg を安全に解放するために、RNG が不要になったときに呼び出す必要があります。rng->drbg をゼロ化し XFREE します。
WC_RNG *	wc_rng_free (WC_RNG * rng)rng を安全に解放するために、RNG が不要になったときに呼び出す必要があります。
int	wc_RNG_HealthTest (int reseed, const byte * seedA, word32 seedASz, const byte * seedB, word32 seedBSz, byte * output, word32 outputSz)drbg の機能を作成してテストします。

C.42.2 Attributes

Name
WC_RNG byte * b

C.42.3 Functions Documentation

C.42.3.1 function wc_InitNetRandom

```
int wc_InitNetRandom(
    const char * configFile,
    wnr_hmac_key hmac_cb,
    int timeout
)
```

グローバル Whitewood netRandom コンテキストを初期化します

Parameters:

- **configFile** 設定ファイルへのパス
- **hmac_cb** HMAC コールバックを作成するためのオプション。
- **timeout** タイムアウト期間。

See: [wc_FreeNetRandom](#)

Return:

- 0 成功
- BAD_FUNC_ARG configFile が null または timeout が負の値の場合。
- RNG_FAILURE_E rng の初期化に失敗しました。

Example

```
char* config = "path/to/config/example.conf";
int time = // 十分なタイムアウト値;

if (wc_InitNetRandom(config, NULL, time) != 0)
{
    // エラーが発生しました
}
```

C.42.3.2 function wc_FreeNetRandom

```
int wc_FreeNetRandom(
    void
)
```

グローバル Whitewood netRandom コンテキストを解放します。

Parameters:

- **none** 戻り値なし。

See: [wc_InitNetRandom](#)

Return:

- 0 成功
- BAD_MUTEX_E wnr_mutex のミューテックスロックエラー

Example

```
int ret = wc_FreeNetRandom();
if(ret != 0)
{
    // エラーを処理
}
```

C.42.3.3 function wc_InitRng

```
int wc_InitRng(
    WC_RNG * rng
)
```

rng 用のシード (OS から) と鍵暗号を取得します。rng->drbg(決定論的乱数ビット生成器) が割り当てられます (wc_FreeRng で割り当て解除する必要があります)。これはブロッキング操作です。

Parameters:

- **rng** シードと鍵暗号で使用するために初期化される乱数生成器

See:

- wc_InitRngCavium
- **wc_RNG_GenerateBlock**
- wc_RNG_GenerateByte
- **wc_FreeRng**
- **wc_RNG_HealthTest**

Return:

- 0 成功時。
- MEMORY_E XMALLOC が失敗しました
- WINCRYPT_E wc_GenerateSeed: コンテキストの取得に失敗しました
- CRYPTGEN_E wc_GenerateSeed: ランダムの取得に失敗しました
- BAD_FUNC_ARG wc_RNG_GenerateBlock 入力が null または sz が MAX_REQUEST_LEN を超えています
- DRBG_CONT_FIPS_E wc_RNG_GenerateBlock: Hash_gen が DRBG_CONT_FAILURE を返しました
- RNG_FAILURE_E wc_RNG_GenerateBlock: デフォルトエラー。rng のステータスが元々 ok でないか、DRBG_FAILED に設定されています

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.42.3.4 function wc_RNG_GenerateBlock

```
int wc_RNG_GenerateBlock(
    WC_RNG * rng,
    byte * b,
    word32 sz
)
```

疑似乱数データの sz バイトを output にコピーします。必要に応じて rng を再シードします (ブロッキング)。

Parameters:

- **rng** wc_InitRng で初期化された乱数生成器
- **output** ブロックがコピーされるバッファ
- **sz** 出力のサイズ (バイト単位)

See:

- wc_InitRngCavium, [wc_InitRng](#)
- wc_RNG_GenerateByte
- [wc_FreeRng](#)
- [wc_RNG_HealthTest](#)

Return:

- 0 成功時
- BAD_FUNC_ARG 入力が null または sz が MAX_REQUEST_LEN を超えています
- DRBG_CONT_FIPS_E Hash_gen が DRBG_CONT_FAILURE を返しました
- RNG_FAILURE_E デフォルトエラー。rng のステータスが元々 ok でないか、DRBG_FAILED に設定されています

Example

```
RNG rng;
int sz = 32;
byte block[sz];

int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //rng の初期化失敗!
}

ret = wc_RNG_GenerateBlock(&rng, block, sz);
if (ret != 0) {
    return -1; //ブロック生成失敗!
}
```

C.42.3.5 function wc_rng_new

```
WC_RNG * wc_rng_new(
    byte * nonce,
    word32 nonceSz,
    void * heap
)
```

新しい WC_RNG 構造体を作成します。

Parameters:

- **heap** ヒープ識別子へのポインタ

- **nonce** nonce を含むバッファへのポインタ
- **nonceSz** nonce の長さ
- **rng** wc_InitRng で初期化された乱数生成器
- **b** ブロックがコピーされる 1 バイトのバッファ

See:

- [wc_InitRng](#)
- [wc_rng_free](#)
- [wc_FreeRng](#)
- [wc_RNG_HealthTest](#)
- [wc_InitRngCavium](#)
- [wc_InitRng](#)
- [wc_RNG_GenerateBlock](#)
- [wc_FreeRng](#)
- [wc_RNG_HealthTest](#)

Return:

- WC_RNG 成功時の構造体
- NULL エラー時
- 0 成功時
- BAD_FUNC_ARG 入力が null または sz が MAX_REQUEST_LEN を超えています
- DRBG_CONT_FIPS_E Hash_gen が DRBG_CONT_FAILURE を返しました
- RNG_FAILURE_E デフォルトエラー。rng のステータスが元々 ok でないか、DRBG_FAILED に設定されています

Example

```
RNG rng;
byte nonce[] = { nonce を初期化 };
word32 nonceSz = sizeof(nonce);
```

```
wc_rng_new(&nonce, nonceSz, &heap);
```

疑似乱数データの 1 バイトを b にコピーするために wc_RNG_GenerateBlock を呼び出します。必要に応じて rng を再シードします。

Example

```
RNG rng;
int sz = 32;
byte b[1];

int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //rng の初期化失敗!
}

ret = wc_RNG_GenerateByte(&rng, b);
if (ret != 0) {
    return -1; //ブロック生成失敗!
}
```

C.42.3.6 function wc_FreeRng

```
int wc_FreeRng(
    WC_RNG * rng
)
```

drbg を安全に解放するために、RNG が不要になったときに呼び出す必要があります。rng-drbg をゼロ化し XFREE します。

Parameters:

- **rng** wc_InitRng で初期化された乱数生成器

See:

- wc_InitRngCavium
- **wc_InitRng**
- **wc_RNG_GenerateBlock**
- wc_RNG_GenerateByte,
- **wc_RNG_HealthTest**

Return:

- 0 成功時
- BAD_FUNC_ARG rng または rng->drbg が null
- RNG_FAILURE_E drbg の割り当て解除に失敗しました

Example

```
RNG rng;
int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //rng の初期化失敗!
}

int ret = wc_FreeRng(&rng);
if (ret != 0) {
    return -1; //rng の解放失敗!
}
```

C.42.3.7 function wc_rng_free

```
WC_RNG * wc_rng_free(
    WC_RNG * rng
)
```

rng を安全に解放するために、RNG が不要になったときに呼び出す必要があります。

Parameters:

- **rng** wc_InitRng で初期化された乱数生成器

See:

- **wc_InitRng**
- **wc_rng_new**
- **wc_FreeRng**
- **wc_RNG_HealthTest**

Example

```
RNG rng;
byte nonce[] = { nonce を初期化 };
word32 nonceSz = sizeof(nonce);

rng = wc_rng_new(&nonce, nonceSz, &heap);

// rng を使用
```

```
wc_rng_free(&rng);
```

C.42.3.8 function wc_RNG_HealthTest

```
int wc_RNG_HealthTest(
    int reseed,
    const byte * seedA,
    word32 seedASz,
    const byte * seedB,
    word32 seedBSz,
    byte * output,
    word32 outputSz
)
```

drbg の機能を作成してテストします。

Parameters:

- **int** reseed: 設定されている場合、再シード機能をテストします
- **seedA** drbg をインスタンス化するシード
- **seedASz** seedA のサイズ (バイト単位)
- **seedB** reseed が設定されている場合、drbg は seedB で再シードされます
- **seedBSz** seedB のサイズ (バイト単位)
- **output** seedrandom が設定されている場合は seedB でシードされたランダムデータに初期化され、それ以外の場合は seedA でシードされます
- **outputSz** output の長さ (バイト単位)

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng

Return:

- 0 成功時
- BAD_FUNC_ARG seedA と output は null であってはなりません。reseed が設定されている場合、seedB は null であってはなりません
- -1 テスト失敗

Example

```
byte output[SHA256_DIGEST_SIZE * 4];
const byte test1EntropyB[] = ....; // reseed false のテスト入力
const byte test1Output[] = ....; // テストベクター: reseed false の期待出力
ret = wc_RNG_HealthTest(0, test1Entropy, sizeof(test1Entropy), NULL, 0,
                        output, sizeof(output));
if (ret != 0)
    return -1; //再シードなしのヘルステスト失敗

if (XMEMCMP(test1Output, output, sizeof(output)) != 0)
    return -1; //テストベクターとの比較失敗: 予期しない出力

const byte test2EntropyB[] = ....; // reseed のテスト入力
const byte test2Output[] = ....; // テストベクター reseed の期待出力
ret = wc_RNG_HealthTest(1, test2EntropyA, sizeof(test2EntropyA),
```

```

        test2EntropyB, sizeof(test2EntropyB),
        output, sizeof(output));

if (XMEMCMP(test2Output, output, sizeof(output)) != 0)
    return -1; //テストベクターとの比較失敗

```

C.42.4 Attributes Documentation

C.42.4.1 variable b

WC_RNG byte * b;

C.42.5 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* b, word32 sz);

WC_RNG* wc_rng_new(byte* nonce, word32 nonceSz, void* heap)

int wc_RNG_GenerateByte(WC_RNG* rng, byte* b);

int wc_FreeRng(WC_RNG* rng);

WC_RNG* wc_rng_free(WC_RNG* rng);

int wc_RNG_HealthTest(int reseed, const byte* seedA, word32 seedASz,
    const byte* seedB, word32 seedBSz,
    byte* output, word32 outputSz);

```

C.43 dox_comments/header_files-ja/ripemd.h

C.43.1 Functions

	Name
int	wc_InitRipeMd (RipeMd * ripemd) この関数は、ripemd のダイジェスト、バッファ、loLen および hiLen を初期化することで、ripemd 構造体を初期化します。
int	wc_RipeMdUpdate (RipeMd * ripemd, const byte * data, word32 len) この関数は、入力データの RipeMd ダイジェストを生成し、結果を ripemd->digest バッファに格納します。wc_RipeMdUpdate を実行した後、生成された ripemd->digest を既知の認証タグと比較して、メッセージの真正性を検証する必要があります。

	Name
int	wc_RipeMdFinal (RipeMd * ripemd, byte * hash) この関数は、計算されたダイジェストを hash にコピーします。部分的にハッシュ化されていないブロックがある場合、このメソッドはブロックを 0 でパディングし、hash にコピーする前にそのブロックのラウンドをダイジェストに含めます。ripemd の状態はリセットされます。

C.43.2 Functions Documentation

C.43.2.1 function wc_InitRipeMd

```
int wc_InitRipeMd(
    RipeMd * ripemd
)
```

この関数は、ripemd のダイジェスト、バッファ、loLen および hiLen を初期化することで、ripemd 構造体を初期化します。

Parameters:

- **ripemd** 初期化する ripemd 構造体へのポインタ

See:

- **wc_RipeMdUpdate**
- **wc_RipeMdFinal**

Return:

- 0 関数の実行が成功した場合に返されます。RipeMd 構造体が初期化されます。
- BAD_FUNC_ARG RipeMd 構造体が NULL の場合に返されます。

Example

```
RipeMd md;
int ret;
ret = wc_InitRipeMd(&md);
if (ret != 0) {
    // 失敗ケース
}
```

C.43.2.2 function wc_RipeMdUpdate

```
int wc_RipeMdUpdate(
    RipeMd * ripemd,
    const byte * data,
    word32 len
)
```

この関数は、入力データの RipeMd ダイジェストを生成し、結果を ripemd->digest バッファに格納します。wc_RipeMdUpdate を実行した後、生成された ripemd->digest を既知の認証タグと比較して、メッセージの真正性を検証する必要があります。

Parameters:

- **ripemd** wc_InitRipeMd で初期化される ripemd 構造体へのポインタ
- **data** ハッシュ化されるデータ

- **len** データのサイズ (バイト単位)

See:

- `wc_InitRipeMd`
- `wc_RipeMdFinal`

Return:

- 0 関数の実行が成功した場合に返されます。
- BAD_FUNC_ARG RipeMd 構造体が NULL の場合、または data が NULL で len がゼロでない場合に返されます。この関数は、data が NULL で len が 0 の場合は実行されるべきです。

Example

```
const byte* data; // ハッシュ化されるデータ
....
RipeMd md;
int ret;
ret = wc_InitRipeMd(&md);
if (ret == 0) {
ret = wc_RipeMdUpdate(&md, plain, sizeof(plain));
if (ret != 0) {
// 失敗ケース ...
```

C.43.2.3 function wc_RipeMdFinal

```
int wc_RipeMdFinal(
    RipeMd * ripemd,
    byte * hash
)
```

この関数は、計算されたダイジェストを hash にコピーします。部分的にハッシュ化されていないブロックがある場合、このメソッドはブロックを 0 でパディングし、hash にコピーする前にそのブロックのラウンドをダイジェストに含めます。ripemd の状態はリセットされます。

Parameters:

- **ripemd** wc_InitRipeMd で初期化され、wc_RipeMdUpdate からのハッシュを含む ripemd 構造体へのポインタ。状態はリセットされます
- **hash** ダイジェストをコピーするバッファ。RIPEMD_DIGEST_SIZE バイトである必要があります

See: none

Return:

- 0 関数の実行が成功した場合に返されます。RipeMd 構造体の状態がリセットされました。
- BAD_FUNC_ARG RipeMd 構造体または hash パラメータが NULL の場合に返されます。

Example

```
RipeMd md;
int ret;
byte digest[RIPEMD_DIGEST_SIZE];
const byte* data; // ハッシュ化されるデータ
...
ret = wc_InitRipeMd(&md);
if (ret == 0) {
ret = wc_RipeMdUpdate(&md, plain, sizeof(plain));
if (ret != 0) {
// RipeMd 更新失敗ケース
```

```

}
ret = wc_RipeMdFinal(&md, digest);
if (ret != 0) {
    // RipeMd Final 失敗ケース
}...

```

C.43.3 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.44 dox_comments/header_files-ja/rsa.h

C.44.1 Functions

	Name
int	wc_InitRsaKey (RsaKey * key, void * heap) この関数は、提供された RsaKey 構造体を初期化します。また、ユーザー定義のメモリオーバーライド (XMMALLOC、XFREE、XREALLOC を参照) で使用するために、ヒープ識別子も受け取ります。
int	wc_InitRsaKey_Id (RsaKey * key, unsigned char * id, int len, void * heap, int devId) この関数は、提供された RsaKey 構造体を初期化します。id と len は、デバイス上のキーを識別するために使用され、devId はデバイスを識別します。また、ユーザー定義のメモリオーバーライド (XMMALLOC、XFREE、XREALLOC を参照) で使用するために、ヒープ識別子も受け取ります。
int	wc_RsaSetRNG (RsaKey * key, WC_RNG * rng) この関数は、RNG をキーに関連付けます。WC_RSA_BLINDING が有効になっている場合に必要です。
int	wc_FreeRsaKey (RsaKey * key) この関数は、mp_clear を使用して提供された RsaKey 構造体を解放します。
int	wc_RsaDirect (const byte * in, word32 inLen, byte * out, word32 * outSz, RsaKey * key, int type, WC_RNG * rng) パディングなしで RSA 操作を直接実行する関数。入力サイズはキーサイズと一致する必要があります。通常、これは RSA 入力に既にパディングが行われている場合に使用されます。

	Name
int	wc_RsaPublicEncrypt (const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng) この関数は、in からメッセージを暗号化し、結果を out に格納します。初期化された公開鍵と乱数ジェネレータが必要です。副作用として、この関数は out に書き込まれたバイト数を outLen で返します。
int	wc_RsaPrivateDecryptInline (byte * in, word32 inLen, byte ** out, RsaKey * key) この関数は、復号のために wc_RsaPrivateDecrypt 関数によって使用されます。
int	wc_RsaPrivateDecrypt (const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key) この関数は、プライベート RSA 復号を提供します。
int	wc_RsaSSL_Sign (const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng) 秘密鍵で提供された配列に署名します。
int	wc_RsaSSL_VerifyInline (byte * in, word32 inLen, byte ** out, RsaKey * key) メッセージが RSA キーによって署名されたことを検証するために使用されます。出力は入力と同じバイト配列を使用します。
int	wc_RsaSSL_Verify (const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key) メッセージがキーによって署名されたことを検証するために使用されます。
int	wc_RsaPSS_Sign (const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, RsaKey * key, WC_RNG * rng) 秘密鍵で提供された配列に署名します。
int	wc_RsaPSS_Verify (const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, RsaKey * key) メッセージがキーによって署名されたことを検証するために入力署名を復号します。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。
int	wc_RsaPSS_VerifyInline (byte * in, word32 inLen, byte ** out, enum wc_HashType hash, int mgf, RsaKey * key) メッセージが RSA キーによって署名されたことを検証するために入力署名を復号します。出力は入力と同じバイト配列を使用します。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。

	Name
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)RSA-PSS で署名されたメッセージを検証します。ソルトの長さはハッシュの長さと同じになります。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。
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)RSA-PSS で署名されたメッセージを検証します。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。
int	wc_RsaPSS_VerifyCheckInline (byte * in, word32 inLen, byte ** out, const byte * digest, word32 digestLen, enum wc_HashType hash, int mgf, RsaKey * key)RSA-PSS で署名されたメッセージを検証します。入力バッファは出力バッファとして再利用されます。ソルトの長さはハッシュの長さと同じになります。
int	wc_RsaPSS_VerifyCheckInline_ex (byte * in, word32 inLen, byte ** out, const byte * digest, word32 digestLen, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key)RSA-PSS で署名されたメッセージを検証します。入力バッファは出力バッファとして再利用されます。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。
int	wc_RsaPSS_CheckPadding (const byte * in, word32 inLen, const byte * sig, word32 sigSz, enum wc_HashType hashType) 署名が一致することを確認するために PSS データをチェックします。ソルトの長さはハッシュの長さと同じになります。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。
int	wc_RsaPSS_CheckPadding_ex (const byte * in, word32 inLen, const byte * sig, word32 sigSz, enum wc_HashType hashType, int saltLen, int bits) 署名が一致することを確認するために PSS データをチェックします。ソルトの長さはハッシュの長さと同じになります。
int	wc_RsaEncryptSize (const RsaKey * key) 提供されたキー構造体の暗号化サイズを返します。

	Name
int	wc_RsaPrivateKeyDecode (const byte * input, word32 * inOutIdx, RsaKey * key, word32 inSz) この関数は、DER 形式の RSA 秘密鍵を解析し、秘密鍵を抽出して、指定された RsaKey 構造体に格納します。また、idx に解析された距離を設定します。
int	wc_RsaPublicKeyDecode (const byte * input, word32 * inOutIdx, RsaKey * key, word32 inSz) この関数は、DER 形式の RSA 公開鍵を解析し、公開鍵を抽出して、指定された RsaKey 構造体に格納します。また、idx に解析された距離を設定します。
int	wc_RsaPublicKeyDecodeRaw (const byte * n, word32 nSz, const byte * e, word32 eSz, RsaKey * key) この関数は、RSA 公開鍵の生の要素をデコードし、公開モジュラス (n) と指数 (e) を受け取ります。これらの生の要素を提供された RsaKey 構造体に格納し、暗号化/復号プロセスでそれらを使用できるようにします。
int	wc_RsaKeyToDer (RsaKey * key, byte * output, word32 inLen) この関数は、RsaKey キーを DER 形式に変換します。結果は output に書き込まれ、書き込まれたバイト数を返します。
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) この関数は、使用するパディングを選択できるようにしながら、RSA 暗号化を実行します。
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) この関数は、RSA を使用してメッセージを復号し、どのパディングタイプを使用するかオプションを提供します。
int	wc_RsaPrivateDecryptInline_ex (byte * in, word32 inLen, byte ** out, RsaKey * key, int type, enum wc_HashType hash, int mgf, byte * label, word32 labelSz) この関数は、RSA を使用してメッセージをインラインで復号し、どのパディングタイプを使用するかオプションを提供します。in バッファは呼び出し後に復号されたメッセージを含み、out バイトポインタは平文がある「in」バッファ内の場所を指します。
int	wc_RsaFlattenPublicKey (const RsaKey * key, byte * e, word32 * eSz, byte * n, word32 * nSz) RsaKey 構造体を RSA アルゴリズムに使用される個々の要素 (e、n) に展開します。

	Name
int	wc_RsaKeyToPublicDer (RsaKey * key, byte * output, word32 inLen)RSA 公開鍵を DER 形式に変換します。output に書き込み、書き込まれたバイト数を返します。
int	wc_RsaKeyToPublicDer_ex (RsaKey * key, byte * output, word32 inLen, int with_header)RSA 公開鍵を DER 形式に変換します。output に書き込み、書き込まれたバイト数を返します。with_header が 0 の場合、(seq + n + e) のみが ASN.1 DER 形式で返され、ヘッダーは除外されます。
int	wc_MakeRsaKey (RsaKey * key, int size, long e, WC_RNG * rng) この関数は、長さ size (ビット単位) と指定された指数 (e) の RSA 秘密鍵を生成します。その後、このキーを提供された RsaKey 構造体に格納し、暗号化/復号に使用できるようにします。e に使用する安全な数値は 65537 です。size は RSA_MIN_SIZE 以上かつ RSA_MAX_SIZE 以下である必要があります。この関数を使用するには、コンパイル時にオプション WOLFSSL_KEY_GEN を有効にする必要があります。./configure を使用する場合は、-enable-keygen で実現できます。
int	wc_RsaSetNonBlock (RsaKey * key, RsaNb * nb) この関数は、ノンブロッキング RSA コンテキストを設定します。RsaNb コンテキストが設定されると、RSA 関数を多くの小さな操作に分割する、高速数学ベースのノンブロッキング exptmod が有効になります。WC_RSA_NONBLOCK が定義されている場合に有効になります。
int	wc_RsaSetNonBlockTime (RsaKey * key, word32 maxBlockUs, word32 cpuMHz) この関数は、最大ブロッキング時間をマイクロ秒単位で設定します。CPU 速度 (メガヘルツ単位) とともに事前計算されたテーブル (tfm.c exptModNbInst を参照) を使用して、次の操作が提供された最大ブロッキング時間内に完了できるかどうかを判断します。WC_RSA_NONBLOCK_TIME が定義されている場合に有効になります。

C.44.2 Functions Documentation

C.44.2.1 function wc_InitRsaKey

```
int wc_InitRsaKey(
    RsaKey * key,
    void * heap
)
```

この関数は、提供された RsaKey 構造体を初期化します。また、ユーザー定義のメモリオーバーライド (XMALLOCF、XFREE、XREALLOC を参照) で使用するために、ヒープ識別子も受け取ります。

Parameters:

- **key** 初期化する RsaKey 構造体へのポインタ

- **heap** メモリオーバーライドで使用するヒープ識別子へのポインタ。メモリ割り当てのカスタム処理を可能にします。このヒープは、この RSA オブジェクトで使用するメモリを割り当てる際にデフォルトで使用されます

See:

- [wc_FreeRsaKey](#)
- [wc_RsaSetRNG](#)

Return:

- 0 暗号化と復号に使用する RSA 構造体の初期化に成功した場合に返されます
- BAD_FUNC_ARGS RSA キーポインタが NULL と評価された場合に返されます

WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。

Example

```
RsaKey enc;
int ret;
ret = wc_InitRsaKey(&enc, NULL); // ヒープヒントを使用しない。カスタムメモリなし
if ( ret != 0 ) {
    // RSA キーの初期化エラー
}
```

C.44.2.2 function wc_InitRsaKey_Id

```
int wc_InitRsaKey_Id(
    RsaKey * key,
    unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

この関数は、提供された RsaKey 構造体を初期化します。id と len は、デバイス上のキーを識別するために使用され、devId はデバイスを識別します。また、ユーザー定義のメモリオーバーライド（XMALLOC、XFREE、XREALLOC を参照）で使用するために、ヒープ識別子も受け取ります。

Parameters:

- **key** 初期化する RsaKey 構造体へのポインタ
- **id** デバイス上のキーの識別子
- **len** 識別子の長さ（バイト単位）
- **heap** メモリオーバーライドで使用するヒープ識別子へのポインタ。メモリ割り当てのカスタム処理を可能にします。このヒープは、この RSA オブジェクトで使用するメモリを割り当てる際にデフォルトで使用されます
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定

See:

- [wc_InitRsaKey](#)
- [wc_FreeRsaKey](#)
- [wc_RsaSetRNG](#)

Return:

- 0 暗号化と復号に使用する RSA 構造体の初期化に成功した場合に返されます
- BAD_FUNC_ARGS RSA キーポインタが NULL と評価された場合に返されます

- BUFFER_E len が 0 未満または RSA_MAX_ID_LEN より大きい場合に返されます。

WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。

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 ) {
    // コールバックとトークンをデバイス ID に関連付けるエラー
}
ret = wc_InitRsaKey_Id(&enc, id, len, NULL, devId); // ヒープヒントを使用しない
if ( ret != 0 ) {
    // RSA キーの初期化エラー
}
```

C.44.2.3 function wc_RsaSetRNG

```
int wc_RsaSetRNG(
    RsaKey * key,
    WC_RNG * rng
)
```

この関数は、RNG をキーに関連付けます。WC_RSA_BLINDING が有効になっている場合に必要です。

Parameters:

- **key** 関連付ける RsaKey 構造体へのポインタ
- **rng** 関連付ける WC_RNG 構造体へのポインタ

See:

- [wc_InitRsaKey](#)
- [wc_RsaSetRNG](#)

Return:

- 0 成功時に返されます
- BAD_FUNC_ARGS RSA キー、rng ポインタが NULL と評価された場合に返されます

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.44.2.4 function wc_FreeRsaKey

```
int wc_FreeRsaKey(
    RsaKey * key
)
```


この関数は、mp_clear を使用して提供された RsaKey 構造体を解放します。

Parameters:

- **key** 解放する RsaKey 構造体へのポインタ

See: [wc_InitRsaKey](#)

Return: 0 キーの解放に成功した場合に返されます

Example

```
RsaKey enc;
wc_InitRsaKey(&enc, NULL); // ヒープヒントを使用しない。カスタムメモリなし
... キーを設定し、暗号化を実行

wc_FreeRsaKey(&enc);
```

C.44.2.5 function wc_RsaDirect

```
int wc_RsaDirect(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outSz,
    RsaKey * key,
    int type,
    WC_RNG * rng
)
```

パディングなしで RSA 操作を直接実行する関数。入力サイズはキーサイズと一致する必要があります。通常、これは RSA 入力に既にパディングが行われている場合に使用されます。

Parameters:

- **in** 操作を行うバッファ
- **inLen** 入力バッファの長さ
- **out** 結果を保持するバッファ
- **outSz** 結果バッファのサイズに設定されます。出力バッファの長さとして渡す必要があります。ポインタ「out」が null の場合、outSz は必要な予想バッファサイズに設定され、LENGTH_ONLY_E が返されます。
- **key** 暗号化/復号に使用する初期化された RSA キー
- **type** 秘密鍵または公開鍵を使用する場合 (RSA_PUBLIC_ENCRYPT、RSA_PUBLIC_DECRYPT、RSA_PRIVATE_ENCRYPT、RSA_PRIVATE_DECRYPT)
- **rng** 初期化された WC_RNG 構造体

See:

- [wc_RsaPublicEncrypt](#)
- [wc_RsaPrivateDecrypt](#)

Return:

- size 暗号化に成功した場合、暗号化されたバッファのサイズが返されます
- RSA_BUFFER_E RSA バッファエラー、出力が小さすぎるか入力が大きすぎます

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) {
    // エラーを処理
}

```

C.44.2.6 function wc_RsaPublicEncrypt

```

int wc_RsaPublicEncrypt(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng
)

```

この関数は、in からメッセージを暗号化し、結果を out に格納します。初期化された公開鍵と乱数ジェネレータが必要です。副作用として、この関数は out に書き込まれたバイト数を outLen で返します。

Parameters:

- **in** 暗号化する入力メッセージを含むバッファへのポインタ
- **inLen** 暗号化するメッセージの長さ
- **out** 出力暗号文を格納するバッファへのポインタ
- **outLen** 出力バッファの長さ
- **key** 暗号化に使用する公開鍵を含む RsaKey 構造体へのポインタ
- **rng** ランダムブロックパディングを生成する RNG 構造体

See: [wc_RsaPrivateDecrypt](#)

Return:

- Success 入力メッセージの暗号化に成功した場合、成功時に書き込まれたバイト数を返し、失敗の場合はゼロ未満を返します。
- BAD_FUNC_ARG いずれかの入力パラメータが無効な場合に返されます
- RSA_BUFFER_E 出力バッファが暗号文を格納するには小さすぎる場合に返されます
- RNG_FAILURE_E 提供された RNG 構造体を使用してランダムブロックを生成する際にエラーがある場合に返されます
- MP_INIT_E メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_READ_E メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_CMP_E メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_INVMOD_E メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_EXPTMOD_E メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_MOD_E メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_MUL_E メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります

- MP_ADD_E メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_MULMOD_E メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_TO_E メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_MEM メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_ZERO_E メッセージの暗号化中に使用される数学ライブラリにエラーがある場合に返される可能性があります

Example

```
RsaKey pub;
int ret = 0;
byte n[] = { // 受信した公開鍵の n 成分で初期化 };
byte e[] = { // 受信した公開鍵の e 成分で初期化 };
byte msg[] = { // 暗号化するメッセージの平文で初期化 };
byte cipher[256]; // 256 バイトは 2048 ビット RSA 暗号文を格納するのに十分な大きさ

wc_InitRsaKey(&pub, NULL); // ヒープヒントを使用しない。カスタムメモリなし
wc_RsaPublicKeyDecodeRaw(n, sizeof(n), e, sizeof(e), &pub);
// 受信した公開鍵パラメータで初期化
ret = wc_RsaPublicEncrypt(msg, sizeof(msg), out, sizeof(out), &pub, &rng);
if ( ret != 0 ) {
    // メッセージの暗号化エラー
}
```

C.44.2.7 function wc_RsaPrivateDecryptInline

```
int wc_RsaPrivateDecryptInline(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key
)
```

この関数は、復号のために wc_RsaPrivateDecrypt 関数によって使用されます。

Parameters:

- **in** 復号されるバイト配列。
- **inLen** in の長さ。
- **out** 復号されたデータを格納するバイト配列。
- **key** 復号に使用するキー。

See: [wc_RsaPrivateDecrypt](#)

Return:

- Success 復号されたデータの長さ。
- RSA_PAD_E RsaUnPad エラー、フォーマットが不正

Example

なし

C.44.2.8 function wc_RsaPrivateDecrypt

```
int wc_RsaPrivateDecrypt(  
    const byte * in,  
    word32 inLen,  
    byte * out,  
    word32 outLen,  
    RsaKey * key  
)
```

この関数は、プライベート RSA 復号を提供します。

Parameters:

- **in** 復号されるバイト配列。
- **inLen** in の長さ。
- **out** 復号されたデータを格納するバイト配列。
- **outLen** out の長さ。
- **key** 復号に使用するキー。

See:

- RsaUnPad
- wc_RsaFunction
- [wc_RsaPrivateDecryptInline](#)

Return:

- Success 復号されたデータの長さ。
- MEMORY_E -125、メモリ不足エラー
- BAD_FUNC_ARG -173、不正な関数引数が提供されました

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.44.2.9 function wc_RsaSSL_Sign

```
int wc_RsaSSL_Sign(  
    const byte * in,  
    word32 inLen,  
    byte * out,  
    word32 outLen,  
    RsaKey * key,  
    WC_RNG * rng  
)
```

秘密鍵で提供された配列に署名します。

Parameters:

- **in** 暗号化されるバイト配列。
- **inLen** in の長さ。
- **out** 暗号化されたデータを格納するバイト配列。
- **outLen** out の長さ。

- **key** 暗号化に使用するキー。
- **RNG** 乱数の目的で使用する RNG 構造体。

See: wc_RsaPad

Return: RSA_BUFFER_E: -131、RSA バッファエラー、出力が小さすぎるか入力が大きすぎます

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.44.2.10 function wc_RsaSSL_VerifyInline

```
int wc_RsaSSL_VerifyInline(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key
)
```

メッセージが RSA キーによって署名されたことを検証するために使用されます。出力は入力と同じバイト配列を使用します。

Parameters:

- **in** 復号されるバイト配列。
- **inLen** 入力バッファの長さ。
- **out** 復号された情報へのポインタへのポインタ。
- **key** 使用する RsaKey。

See:

- [wc_RsaSSL_Verify](#)
- [wc_RsaSSL_Sign](#)

Return:

- >0 ダイジェストの長さ。
- <0 エラーが発生しました。

Example

```
RsaKey key;
WC_RNG rng;
int ret = 0;
long e = 65537; // 指数に使用する標準値
wc_InitRsaKey(&key, NULL); // ヒープヒントを使用しない。カスタムメモリなし
```

```

wc_InitRng(&rng);
wc_MakeRsaKey(&key, 2048, e, &rng);

byte in[] = { // RSA 暗号化情報で初期化 }
byte* out;
if(wc_RsaSSL_VerifyInline(in, sizeof(in), &out, &key) < 0)
{
    // エラーを処理
}

```

C.44.2.11 function wc_RsaSSL_Verify

```

int wc_RsaSSL_Verify(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key
)

```

メッセージがキーによって署名されたことを検証するために使用されます。

Parameters:

- **in** 復号されるバイト配列。
- **inLen** in の長さ。
- **out** 復号されたデータを格納するバイト配列。
- **outLen** out の長さ。
- **key** 検証に使用するキー。

See: [wc_RsaSSL_Sign](#)

Return:

- Success エラーがない場合のダイジェストの長さ。
- MEMORY_E メモリ例外。

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.44.2.12 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
)

```

秘密鍵で提供された配列に署名します。

Parameters:

- **in** 暗号化されるバイト配列。
- **inLen** in の長さ。
- **out** 暗号化されたデータを格納するバイト配列。
- **outLen** out の長さ。
- **hash** メッセージに含まれるハッシュタイプ
- **mgf** マスク生成関数識別子
- **key** 検証に使用するキー。

See:

- [wc_RsaPSS_Verify](#)
- [wc_RsaSetRNG](#)

Return: RSA_BUFFER_E: -131、RSA バッファエラー、出力が小さすぎるか入力が大きすぎます

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.44.2.13 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
)
```

メッセージがキーによって署名されたことを検証するために入力署名を復号します。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。

Parameters:

- **in** 復号されるバイト配列。
- **inLen** in の長さ。
- **out** 復号されたデータを格納するバイト配列。
- **outLen** out の長さ。
- **hash** メッセージに含まれるハッシュタイプ
- **mgf** マスク生成関数識別子
- **key** 検証に使用するキー。

See:

- [wc_RsaPSS_Sign](#)
- [wc_RsaPSS_VerifyInline](#)
- [wc_RsaPSS_CheckPadding](#)
- [wc_RsaSetRNG](#)

Return:

- Success エラーがない場合のテキストの長さ。
- MEMORY_E メモリ例外。
- MP_EXPTMOD_E - fastmath を使用していて、FP_MAX_BITS がキーサイズの少なくとも 2 倍に設定されていない場合（例：4096 ビットキーを使用する場合、FP_MAX_BITS を 8192 以上の値に設定）

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.44.2.14 function wc_RsaPSS_VerifyInline

```
int wc_RsaPSS_VerifyInline(
    byte * in,
    word32 inLen,
    byte ** out,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

メッセージが RSA キーによって署名されたことを検証するために入力署名を復号します。出力は入力と同じバイト配列を使用します。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。

Parameters:

- **in** 復号されるバイト配列。
- **inLen** 入力バッファの長さ。
- **out** PSS データを含むアドレスへのポインタ。
- **hash** メッセージに含まれるハッシュタイプ
- **mgf** マスク生成関数識別子
- **key** 使用する RsaKey。

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 テキストの長さ。
- <0 エラーが発生しました。

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.44.2.15 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
)

```

RSA-PSS で署名されたメッセージを検証します。ソルトの長さはハッシュの長さと同じになります。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。

Parameters:

- **in** 復号されるバイト配列。
- **inLen** in の長さ。
- **out** PSS データを含むアドレスへのポインタ。
- **outLen** out の長さ。
- **digest** 検証されるデータのハッシュ。
- **digestLen** ハッシュの長さ。
- **hash** ハッシュアルゴリズム。
- **mgf** マスク生成関数。
- **key** 公開 RSA キー。

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 成功時は PSS データの長さ、失敗を示す負の値。
- MEMORY_E メモリ例外。

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.44.2.16 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
)

```

RSA-PSS で署名されたメッセージを検証します。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。

Parameters:

- **in** 復号されるバイト配列。
- **inLen** in の長さ。

- **out** PSS データを含むアドレスへのポインタ。
- **outLen** out の長さ。
- **digest** 検証されるデータのハッシュ。
- **digestLen** ハッシュの長さ。
- **hash** ハッシュアルゴリズム。
- **mgf** マスク生成関数。
- **saltLen** 使用されるソルトの長さ。RSA_PSS_SALT_LEN_DEFAULT (-1) は、ソルトの長さがハッシュの長さと同じであることを示します。RSA_PSS_SALT_LEN_DISCOVER は、ソルトの長さがデータから決定されることを示します。
- **key** 公開 RSA キー。

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 成功時は PSS データの長さ、失敗を示す負の値。
- MEMORY_E メモリ例外。

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.44.2.17 function wc_RsaPSS_VerifyCheckInline

```

int wc_RsaPSS_VerifyCheckInline(
    byte * in,
    word32 inLen,
    byte ** out,
    const byte * digest,
    word32 digestLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)

```

RSA-PSS で署名されたメッセージを検証します。入力バッファは出力バッファとして再利用されます。ソルトの長さはハッシュの長さと同じになります。

Parameters:

- **in** 復号されるバイト配列。
- **inLen** in の長さ。
- **out** 復号されたデータを格納するバイト配列。
- **digest** 検証されるデータのハッシュ。
- **digestLen** ハッシュの長さ。
- **hash** メッセージに含まれるハッシュタイプ
- **mgf** マスク生成関数識別子
- **key** 検証に使用するキー。

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 成功時は PSS データの長さ、失敗を示す負の値。

WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。

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.44.2.18 function wc_RsaPSS_VerifyCheckInline_ex

```

int wc_RsaPSS_VerifyCheckInline_ex(
    byte * in,
    word32 inLen,
    byte ** out,
    const byte * digest,
    word32 digestLen,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key
)

```

RSA-PSS で署名されたメッセージを検証します。入力バッファは出力バッファとして再利用されます。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。

Parameters:

- **in** 復号されるバイト配列。
- **inLen** in の長さ。
- **out** 復号されたデータを格納するバイト配列。
- **digest** 検証されるデータのハッシュ。
- **digestLen** ハッシュの長さ。
- **hash** メッセージに含まれるハッシュタイプ
- **mgf** マスク生成関数識別子
- **saltLen** 使用されるソルトの長さ。RSA_PSS_SALT_LEN_DEFAULT (-1) は、ソルトの長さがハッシュの長さと同じであることを示します。RSA_PSS_SALT_LEN_DISCOVER は、ソルトの長さがデータから決定されることを示します。
- **key** 検証に使用するキー。

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 成功時は PSS データの長さ、失敗を示す負の値。

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.44.2.19 function `wc_RsaPSS_CheckPadding`

```
int wc_RsaPSS_CheckPadding(
    const byte * in,
    word32 inLen,
    const byte * sig,
    word32 sigSz,
    enum wc_HashType hashType
)
```

署名が一致することを確認するために PSS データをチェックします。ソルトの長さはハッシュの長さと同じになります。WC_RSA_BLINDING が有効になっている場合、キーは wc_RsaSetRNG によって RNG と関連付ける必要があります。

Parameters:

- **in** 検証されるデータのハッシュ。
- **inSz** ハッシュの長さ。
- **sig** PSS データを保持するバッファ。
- **sigSz** PSS データのサイズ。
- **hashType** ハッシュアルゴリズム。

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 PSS データが無効な場合、BAD_FUNC_ARG in または sig に NULL が渡された場合、または inSz がハッシュアルゴリズムの長さと同じでない場合、成功時は 0。
- MEMORY_E メモリ例外。

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.44.2.20 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
)
```

署名が一致することを確認するために PSS データをチェックします。ソルトの長さはハッシュの長さと同じになります。

Parameters:

- **in** 検証されるデータのハッシュ。
- **inSz** ハッシュの長さ。
- **sig** PSS データを保持するバッファ。
- **sigSz** PSS データのサイズ。
- **hashType** ハッシュアルゴリズム。
- **saltLen** 使用されるソルトの長さ。RSA_PSS_SALT_LEN_DEFAULT (-1) は、ソルトの長さがハッシュの長さと同じであることを示します。RSA_PSS_SALT_LEN_DISCOVER は、ソルトの長さがデータから決定されることを示します。
- **bits** FIPS の場合、ソルトサイズの計算に使用できます

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 PSS データが無効な場合、BAD_FUNC_ARG in または sig に NULL が渡された場合、または inSz がハッシュアルゴリズムの長さと同じでない場合、成功時は 0。
- MEMORY_E メモリ例外。

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.44.2.21 function wc_RsaEncryptSize

```

int wc_RsaEncryptSize(
    const RsaKey * key
)

```

提供されたキー構造体の暗号化サイズを返します。

Parameters:

- **key** 検証に使用するキー。

See:

- [wc_InitRsaKey](#)
- [wc_InitRsaKey_ex](#)
- [wc_MakeRsaKey](#)

Return: Success 提供されたキー構造体の暗号化サイズ。

Example

```
int sz = wc_RsaEncryptSize(&key);
```

C.44.2.22 function wc_RsaPrivateKeyDecode

```

int wc_RsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    RsaKey * key,
    word32 inSz
)

```

この関数は、DER 形式の RSA 秘密鍵を解析し、秘密鍵を抽出して、指定された RsaKey 構造体に格納します。また、idx に解析された距離を設定します。

Parameters:

- **input** デコードする DER 形式の秘密鍵を含むバッファへのポインタ

- **inOutIdx** キーが始まるバッファ内のインデックスへのポインタ（通常は 0）。この関数の副作用として、inOutIdx は入力バッファを通じて解析された距離を格納します
- **key** デコードされた秘密鍵を格納する RsaKey 構造体へのポインタ
- **inSz** 入力バッファのサイズ

See:

- [wc_RsaPublicKeyDecode](#)
- [wc_MakeRsaKey](#)

Return:

- 0 DER エンコードされた入力から秘密鍵を正常に解析した場合に返されます
- ASN_PARSE_E 入力バッファから秘密鍵を解析する際にエラーがある場合に返されます。これは、入力秘密鍵が ASN.1 標準に従って適切にフォーマットされていない場合に発生する可能性があります
- ASN_RSA_KEY_E RSA キー入力の秘密鍵要素を読み取る際にエラーがある場合に返されます

Example

```
RsaKey enc;
word32 idx = 0;
int ret = 0;
byte der[] = { // DER エンコードされた RSA 秘密鍵で初期化 };

wc_InitRsaKey(&enc, NULL); // ヒープヒントを使用しない。カスタムメモリなし
ret = wc_RsaPrivateKeyDecode(der, &idx, &enc, sizeof(der));
if( ret != 0 ) {
    // 秘密鍵の解析エラー
}
```

C.44.2.23 function wc_RsaPublicKeyDecode

```
int wc_RsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    RsaKey * key,
    word32 inSz
)
```

この関数は、DER 形式の RSA 公開鍵を解析し、公開鍵を抽出して、指定された RsaKey 構造体に格納します。また、idx に解析された距離を設定します。

Parameters:

- **input** デコードする入力 DER エンコードされた RSA 公開鍵を含むバッファへのポインタ
- **inOutIdx** キーが始まるバッファ内のインデックスへのポインタ（通常は 0）。この関数の副作用として、inOutIdx は入力バッファを通じて解析された距離を格納します
- **key** デコードされた公開鍵を格納する RsaKey 構造体へのポインタ
- **inSz** 入力バッファのサイズ

See: [wc_RsaPublicKeyDecodeRaw](#)

Return:

- 0 DER エンコードされた入力から公開鍵を正常に解析した場合に返されます
- ASN_PARSE_E 入力バッファから公開鍵を解析する際にエラーがある場合に返されます。これは、入力公開鍵が ASN.1 標準に従って適切にフォーマットされていない場合に発生する可能性があります
- ASN_OBJECT_ID_E ASN.1 オブジェクト ID が RSA 公開鍵のものとは一致しない場合に返されます
- ASN_EXPECT_0_E 入力キーが ASN.1 標準に従って正しくフォーマットされていない場合に返されます

- ASN_BITSTR_E 入力キーが ASN.1 標準に従って正しくフォーマットされていない場合に返されます
- ASN_RSA_KEY_E RSA キー入力の公開鍵要素を読み取る際にエラーがある場合に返されます

Example

```
RsaKey pub;
word32 idx = 0;
int ret = 0;
byte der[] = { // DER エンコードされた RSA 公開鍵で初期化 };

wc_InitRsaKey(&pub, NULL); // ヒープヒントを使用しない。カスタムメモリなし
ret = wc_RsaPublicKeyDecode(der, &idx, &pub, sizeof(der));
if( ret != 0 ) {
    // 公開鍵の解析エラー
}
```

C.44.2.24 function wc_RsaPublicKeyDecodeRaw

```
int wc_RsaPublicKeyDecodeRaw(
    const byte * n,
    word32 nSz,
    const byte * e,
    word32 eSz,
    RsaKey * key
)
```

この関数は、RSA 公開鍵の生の要素をデコードし、公開モジュラス (n) と指数 (e) を受け取ります。これらの生の要素を提供された RsaKey 構造体に格納し、暗号化/復号プロセスでそれらを使用できるようにします。

Parameters:

- **n** 公開 RSA キーの生のモジュラスパラメータを含むバッファへのポインタ
- **nSz** n を含むバッファのサイズ
- **e** 公開 RSA キーの生の指数パラメータを含むバッファへのポインタ
- **eSz** e を含むバッファのサイズ
- **key** 提供された公開鍵要素で初期化する RsaKey 構造体へのポインタ

See: [wc_RsaPublicKeyDecode](#)

Return:

- 0 公開鍵の生の要素を RsaKey 構造体に正常にデコードした場合に返されます
- BAD_FUNC_ARG いずれかの入力引数が NULL と評価された場合に返されます
- MP_INIT_E 多精度整数 (mp_int) ライブラリで使用するために整数を初期化の際にエラーがある場合に返されます
- ASN_GETINT_E 提供された RSA キー要素 (n または e) のいずれかを読み取る際にエラーがある場合に返されます

Example

```
RsaKey pub;
int ret = 0;
byte n[] = { // 受信した公開鍵の n 成分で初期化 };
byte e[] = { // 受信した公開鍵の e 成分で初期化 };

wc_InitRsaKey(&pub, NULL); // ヒープヒントを使用しない。カスタムメモリなし
ret = wc_RsaPublicKeyDecodeRaw(n, sizeof(n), e, sizeof(e), &pub);
if( ret != 0 ) {
```

```

    // 公開鍵要素の解析エラー
}

```

C.44.2.25 function wc_RsaKeyToDer

```

int wc_RsaKeyToDer(
    RsaKey * key,
    byte * output,
    word32 inLen
)

```

この関数は、RsaKey キーを DER 形式に変換します。結果は output に書き込まれ、書き込まれたバイト数を返します。

Parameters:

- **key** 初期化された RsaKey 構造体。
- **output** 出力バッファへのポインタ。
- **inLen** 出力バッファのサイズ。

See:

- [wc_RsaKeyToPublicDer](#)
- [wc_InitRsaKey](#)
- [wc_MakeRsaKey](#)
- [wc_InitRng](#)

Return:

- 0 成功、書き込まれたバイト数。
- BAD_FUNC_ARG key または output が null の場合、または key->type が RSA_PRIVATE でない場合、または inLen が出力バッファに対して十分な大きさでない場合に返されます。
- MEMORY_E メモリの割り当て中にエラーがある場合に返されます。

Example

```

byte* der;
// der にメモリを割り当て
int derSz = // der に割り当てられたメモリの量;
RsaKey key;
WC_RNG rng;
long e = 65537; // 指数に使用する標準値
ret = wc_MakeRsaKey(&key, 2048, e, &rng); // 2048 ビット長の秘密鍵を生成
wc_InitRsaKey(&key, NULL);
wc_InitRng(&rng);
if(wc_RsaKeyToDer(&key, der, derSz) != 0)
{
    // スローされたエラーを処理
}

```

C.44.2.26 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
)

```

この関数は、使用するパディングを選択できるようにしながら、RSA 暗号化を実行します。

Parameters:

- **in** 暗号化するバッファへのポインタ
- **inLen** 暗号化するバッファの長さ
- **out** 作成された暗号化されたメッセージ
- **outLen** 暗号化されたメッセージを保持するために利用可能なバッファの長さ
- **key** 初期化された RSA キー構造体
- **rng** 初期化された WC_RNG 構造体
- **type** 使用するパディングのタイプ (WC_RSA_OAEP_PAD または WC_RSA_PKCSV15_PAD)
- **hash** 使用するハッシュのタイプ (選択肢は hash.h にあります)
- **mgf** 使用するマスク生成関数のタイプ
- **label** 暗号化されたメッセージに関連付けるオプションのラベル
- **labelSz** 使用されるオプションのラベルのサイズ

See:

- [wc_RsaPublicEncrypt](#)
- [wc_RsaPrivateDecrypt_ex](#)

Return:

- size 暗号化に成功した場合、暗号化されたバッファのサイズが返されます
- RSA_BUFFER_E RSA バッファエラー、出力が小さすぎるか入力が大きすぎます

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) {
    // エラーを処理
}

```

C.44.2.27 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
)

```

この関数は、RSA を使用してメッセージを復号し、どのパディングタイプを使用するかオプションを提供します。

Parameters:

- **in** 復号するバッファへのポインタ
- **inLen** 復号するバッファの長さ
- **out** 作成された復号されたメッセージ
- **outLen** 復号されたメッセージを保持するために利用可能なバッファの長さ
- **key** 初期化された RSA キー構造体
- **type** 使用するパディングのタイプ (WC_RSA_OAEP_PAD または WC_RSA_PKCSV15_PAD)
- **hash** 使用するハッシュのタイプ (選択肢は hash.h にあります)
- **mgf** 使用するマスク生成関数のタイプ
- **label** 暗号化されたメッセージに関連付けるオプションのラベル
- **labelSz** 使用されるオプションのラベルのサイズ

See: なし

Return:

- size 復号に成功した場合、復号されたメッセージのサイズが返されます。
- MEMORY_E 必要な配列を malloc するのに十分なメモリがシステムにない場合に返されます。
- BAD_FUNC_ARG 関数に不正な引数が渡された場合に返されます。

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) {
    // エラーを処理
}
...
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) {
    // エラーを処理
}

```

C.44.2.28 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
)

```

この関数は、RSA を使用してメッセージをインラインで復号し、どのパディングタイプを使用するかオプションを提供します。in バッファは呼び出し後に復号されたメッセージを含み、out バイトポインタは平文がある「in」バッファ内の場所を指します。

Parameters:

- **in** 復号するバッファへのポインタ
- **inLen** 復号するバッファの長さ
- **out** 「in」バッファ内の復号されたメッセージの場所へのポインタ
- **key** 初期化された RSA キー構造体
- **type** 使用するパディングのタイプ (WC_RSA_OAEP_PAD または WC_RSA_PKCSV15_PAD)
- **hash** 使用するハッシュのタイプ (選択肢は hash.h にあります)
- **mgf** 使用するマスク生成関数のタイプ
- **label** 暗号化されたメッセージに関連付けるオプションのラベル
- **labelSz** 使用されるオプションのラベルのサイズ

See: なし

Return:

- size 復号に成功した場合、復号されたメッセージのサイズが返されます。
- MEMORY_E: 必要な配列を malloc するのに十分なメモリがシステムにない場合に返されます。
- RSA_PAD_E: パディングにエラーがあった場合に返されます。
- BAD_PADDING_E: パディングを解析中にエラーが発生した場合に返されます。
- BAD_FUNC_ARG: 関数に不正な引数が渡された場合に返されます。

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) {
    // エラーを処理
}
...

ret = wc_RsaPrivateDecryptInline_ex(out, ret, &plain, &key,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
    // エラーを処理
}

```

C.44.2.29 function wc_RsaFlattenPublicKey


```
int wc_RsaFlattenPublicKey(
    const RsaKey * key,
    byte * e,
    word32 * eSz,
    byte * n,
    word32 * nSz
)
```

RsaKey 構造体を RSA アルゴリズムに使用される個々の要素 (e、n) に展開します。

Parameters:

- **key** 検証に使用するキー。
- **e** e の値のバッファ。e は RSA モジュラー演算における大きな正の整数です。
- **eSz** e バッファのサイズ。
- **n** n の値のバッファ。n は RSA モジュラー演算における大きな正の整数です。
- **nSz** n バッファのサイズ。

See:

- [wc_InitRsaKey](#)
- [wc_InitRsaKey_ex](#)
- [wc_MakeRsaKey](#)

Return:

- 0 関数がエラーなく正常に実行された場合に返されます。
- BAD_FUNC_ARG: いずれかのパラメータが null 値で渡された場合に返されます。
- RSA_BUFFER_E: 渡された e または n バッファが正しいサイズでない場合に返されます。
- MP_MEM: 内部関数にメモリエラーがある場合に返されます。
- MP_VAL: 内部関数の引数が無効な場合に返されます。

Example

```
Rsa key; // 有効な RSA キー。
byte e[ バッファサイズ 例: 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) {
    // 失敗ケース。
}
```

C.44.2.30 function wc_RsaKeyToPublicDer

```
int wc_RsaKeyToPublicDer(
    RsaKey * key,
    byte * output,
    word32 inLen
)
```

RSA 公開鍵を DER 形式に変換します。output に書き込み、書き込まれたバイト数を返します。

Parameters:

- **key** 変換する RSA キー構造体。
- **output** DER を保持する出力バッファ。(NULL の場合は長さのみを返します)

- **inLen** バッファの長さ。

See:

- wc_RsaPublicKeyDerSize
- [wc_RsaKeyToPublicDer_ex](#)
- [wc_InitRsaKey](#)

Return:

- 0 成功、書き込まれたバイト数。
- BAD_FUNC_ARG key または output が null の場合に返されます。
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます。
- <0 エラー

Example

```
RsaKey key;
```

```
wc_InitRsaKey(&key, NULL);
```

```
// キーを使用
```

```
const int BUFFER_SIZE = 1024; // バッファに適切なサイズ
```

```
byte output[BUFFER_SIZE];
```

```
if (wc_RsaKeyToPublicDer(&key, output, sizeof(output)) != 0) {
```

```
    // エラーを処理
```

```
}
```

C.44.2.31 function wc_RsaKeyToPublicDer_ex

```
int wc_RsaKeyToPublicDer_ex(
    RsaKey * key,
    byte * output,
    word32 inLen,
    int with_header
)
```

RSA 公開鍵を DER 形式に変換します。output に書き込み、書き込まれたバイト数を返します。with_header が 0 の場合、(seq + n + e) のみが ASN.1 DER 形式で返され、ヘッダーは除外されます。

Parameters:

- **key** 変換する RSA キー構造体。
- **output** DER を保持する出力バッファ。(NULL の場合は長さのみを返します)
- **inLen** バッファの長さ。

See:

- wc_RsaPublicKeyDerSize
- [wc_RsaKeyToPublicDer](#)
- [wc_InitRsaKey](#)

Return:

- 0 成功、書き込まれたバイト数。
- BAD_FUNC_ARG key または output が null の場合に返されます。
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます。
- <0 エラー

Example

```

RsaKey key;

wc_InitRsaKey(&key, NULL);
// キーを使用

const int BUFFER_SIZE = 1024; // バッファに適切なサイズ
byte output[BUFFER_SIZE];
if (wc_RsaKeyToPublicDer_ex(&key, output, sizeof(output), 0) != 0) {
    // エラーを処理
}

```

C.44.2.32 function wc_MakeRsaKey

```

int wc_MakeRsaKey(
    RsaKey * key,
    int size,
    long e,
    WC_RNG * rng
)

```

この関数は、長さ `size` (ビット単位) と指定された指数 (`e`) の RSA 秘密鍵を生成します。その後、このキーを提供された `RsaKey` 構造体に格納し、暗号化/復号に使用できるようにします。`e` に使用する安全な数値は 65537 です。`size` は `RSA_MIN_SIZE` 以上かつ `RSA_MAX_SIZE` 以下である必要があります。この関数を使用するには、コンパイル時にオプション `WOLFSSL_KEY_GEN` を有効にする必要があります。`./configure` を使用する場合は、`-enable-keygen` で実現できます。

Parameters:

- **key** 生成された秘密鍵を格納する `RsaKey` 構造体へのポインタ
- **size** 希望するキーの長さ (ビット単位)。 `RSA_MIN_SIZE` より大きく `RSA_MAX_SIZE` より小さい必要があります
- **e** キーを生成するために使用する指数パラメータ。安全な選択肢は 65537 です
- **rng** キーを作成する際の乱数生成に使用する `RNG` 構造体へのポインタ

See: なし

Return:

- 0 RSA 秘密鍵の生成に成功した場合に返されます
- `BAD_FUNC_ARG` いずれかの入力引数が `NULL` の場合、`size` パラメータが必要な境界外にある場合、または `e` が誤って選択された場合に返されます
- `RNG_FAILURE_E` 提供された `RNG` 構造体を使用してランダムブロックを生成する際にエラーがある場合に返されます
- `MP_INIT_E`
- `MP_READ_E` RSA キーの生成中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- `MP_CMP_E` RSA キーの生成中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- `MP_INVMOD_E` RSA キーの生成中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- `MP_EXPTMOD_E` RSA キーの生成中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- `MP_MOD_E` RSA キーの生成中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- `MP_MUL_E` RSA キーの生成中に使用される数学ライブラリにエラーがある場合に返される可能性があります

- MP_ADD_E RSA キーの生成中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_MULMOD_E RSA キーの生成中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_TO_E RSA キーの生成中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_MEM RSA キーの生成中に使用される数学ライブラリにエラーがある場合に返される可能性があります
- MP_ZERO_E RSA キーの生成中に使用される数学ライブラリにエラーがある場合に返される可能性があります

Example

```
RsaKey priv;
WC_RNG rng;
int ret = 0;
long e = 65537; // 指数に使用する標準値

wc_InitRsaKey(&priv, NULL); // ヒープヒントを使用しない。カスタムメモリなし
wc_InitRng(&rng);
// 2048 ビット長の秘密鍵を生成
ret = wc_MakeRsaKey(&priv, 2048, e, &rng);
if( ret != 0 ) {
    // 秘密鍵の生成エラー
}
```

C.44.2.33 function wc_RsaSetNonBlock

```
int wc_RsaSetNonBlock(
    RsaKey * key,
    RsaNb * nb
)
```

この関数は、ノンブロッキング RSA コンテキストを設定します。RsaNb コンテキストが設定されると、RSA 関数を多くの小さな操作に分割する、高速数学ベースのノンブロッキング exptmod が有効になります。WC_RSA_NONBLOCK が定義されている場合に有効になります。

Parameters:

- **key** RSA キー構造体
- **nb** この RSA キーが使用する RSA ノンブロッキング構造体。

See: [wc_RsaSetNonBlockTime](#)

Return:

- 0 成功
- BAD_FUNC_ARG key または nb が null の場合に返されます。

Example

```
int ret, count = 0;
RsaKey key;
RsaNb nb;

wc_InitRsaKey(&key, NULL);

// ノンブロッキング RSA モードを有効化 - コンテキストを提供
ret = wc_RsaSetNonBlock(key, &nb);
```

```

if (ret != 0)
    return ret;

do {
    ret = wc_RsaSSL_Sign(in, inLen, out, outSz, key, rng);
    count++; // ブロック回数を追跡
    if (ret == FP_WOULDBLOCK) {
        // ここで「その他の」作業を実行
    }
} while (ret == FP_WOULDBLOCK);
if (ret < 0) {
    return ret;
}

printf("RSA ノンブロック署名: サイズ %d、%d回\n", ret, count);

```

C.44.2.34 function wc_RsaSetNonBlockTime

```

int wc_RsaSetNonBlockTime(
    RsaKey * key,
    word32 maxBlockUs,
    word32 cpuMHz
)

```

この関数は、最大ブロッキング時間をマイクロ秒単位で設定します。CPU 速度（メガヘルツ単位）とともに事前計算されたテーブル（tfm.c exptModNbInst を参照）を使用して、次の操作が提供された最大ブロッキング時間内に完了できるかどうかを判断します。WC_RSA_NONBLOCK_TIME が定義されている場合に有効になります。

Parameters:

- **key** RSA キー構造体。
- **maxBlockUs** 最大ブロック時間（マイクロ秒）。
- **cpuMHz** CPU 速度（メガヘルツ単位）。

See: [wc_RsaSetNonBlock](#)

Return:

- 0 成功
- BAD_FUNC_ARG key が null の場合、または wc_RsaSetNonBlock が事前に呼び出されておらず key->nb が null の場合に返されます。

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.44.3 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 digestLen,
    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 inLen);

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 inLen);

int wc_RsaKeyToPublicDer_ex(RsaKey* key, byte* output, word32 inLen,
                             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);

```

C.45 dox_comments/header_files-ja/sakke.h

C.45.1 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.45.2 Functions Documentation

C.45.2.1 function wc_InitSakkeKey

```
int wc_InitSakkeKey(
    SakkeKey * key,
    void * heap,
    int devId
)
```

C.45.2.2 function wc_InitSakkeKey_ex

```
int wc_InitSakkeKey_ex(
    SakkeKey * key,
    int keySize,
    int curveId,
    void * heap,
    int devId
)
```

C.45.2.3 function wc_FreeSakkeKey

```
void wc_FreeSakkeKey(
    SakkeKey * key
)
```

C.45.2.4 function wc_MakeSakkeKey

```
int wc_MakeSakkeKey(
    SakkeKey * key,
    WC_RNG * rng
)
```

C.45.2.5 function wc_MakeSakkePublicKey

```
int wc_MakeSakkePublicKey(
    SakkeKey * key,
    ecc_point * pub
)
```

C.45.2.6 function wc_MakeSakkeRsk

```
int wc_MakeSakkeRsk(  
    SakkeKey * key,  
    const byte * id,  
    word16 idSz,  
    ecc_point * rsk  
)
```

C.45.2.7 function wc_ValidateSakkeRsk

```
int wc_ValidateSakkeRsk(  
    SakkeKey * key,  
    const byte * id,  
    word16 idSz,  
    ecc_point * rsk,  
    int * valid  
)
```

C.45.2.8 function wc_GenerateSakkeRskTable

```
int wc_GenerateSakkeRskTable(  
    const SakkeKey * key,  
    const ecc_point * rsk,  
    byte * table,  
    word32 * len  
)
```

C.45.2.9 function wc_ExportSakkeKey

```
int wc_ExportSakkeKey(  
    SakkeKey * key,  
    byte * data,  
    word32 * sz  
)
```

C.45.2.10 function wc_ImportSakkeKey

```
int wc_ImportSakkeKey(  
    SakkeKey * key,  
    const byte * data,  
    word32 sz  
)
```

C.45.2.11 function wc_ExportSakkePrivateKey

```
int wc_ExportSakkePrivateKey(  
    SakkeKey * key,  
    byte * data,  
    word32 * sz  
)
```

C.45.2.12 function wc_ImportSakkePrivateKey

```
int wc_ImportSakkePrivateKey(  
    SakkeKey * key,  
    const byte * data,  
    word32 sz  
)
```

C.45.2.13 function wc_EncodeSakkeRsk

```
int wc_EncodeSakkeRsk(  
    const SakkeKey * key,  
    ecc_point * rsk,  
    byte * out,  
    word32 * sz,  
    int raw  
)
```

C.45.2.14 function wc_DecomposeSakkeRsk

```
int wc_DecomposeSakkeRsk(  
    const SakkeKey * key,  
    const byte * data,  
    word32 sz,  
    ecc_point * rsk  
)
```

C.45.2.15 function wc_ImportSakkeRsk

```
int wc_ImportSakkeRsk(  
    SakkeKey * key,  
    const byte * data,  
    word32 sz  
)
```

C.45.2.16 function wc_ExportSakkePublicKey

```
int wc_ExportSakkePublicKey(  
    SakkeKey * key,  
    byte * data,  
    word32 * sz,  
    int raw  
)
```

C.45.2.17 function wc_ImportSakkePublicKey

```
int wc_ImportSakkePublicKey(  
    SakkeKey * key,  
    const byte * data,  
    word32 sz,  
    int trusted  
)
```

C.45.2.18 function wc_GetSakkeAuthSize

```
int wc_GetSakkeAuthSize(  
    SakkeKey * key,  
    word16 * authSz  
)
```

C.45.2.19 function wc_SetSakkeIdentity

```
int wc_SetSakkeIdentity(  
    SakkeKey * key,  
    const byte * id,  
    word16 idSz  
)
```

C.45.2.20 function wc_MakeSakkePointI

```
int wc_MakeSakkePointI(  
    SakkeKey * key,  
    const byte * id,  
    word16 idSz  
)
```

C.45.2.21 function wc_GetSakkePointI

```
int wc_GetSakkePointI(  
    SakkeKey * key,  
    byte * data,  
    word32 * sz  
)
```

C.45.2.22 function wc_SetSakkePointI

```
int wc_SetSakkePointI(  
    SakkeKey * key,  
    const byte * id,  
    word16 idSz,  
    const byte * data,  
    word32 sz  
)
```

C.45.2.23 function wc_GenerateSakkePointITable

```
int wc_GenerateSakkePointITable(  
    SakkeKey * key,  
    byte * table,  
    word32 * len  
)
```

C.45.2.24 function wc_SetSakkePointITable

```
int wc_SetSakkePointITable(  
    SakkeKey * key,  
    byte * table,
```

```
    word32 len  
)
```

C.45.2.25 function wc_ClearSakkePointITable

```
int wc_ClearSakkePointITable(  
    SakkeKey * key  
)
```

C.45.2.26 function wc_MakeSakkeEncapsulatedSSV

```
int wc_MakeSakkeEncapsulatedSSV(  
    SakkeKey * key,  
    enum wc_HashType hashType,  
    byte * ssv,  
    word16 ssvSz,  
    byte * auth,  
    word16 * authSz  
)
```

C.45.2.27 function wc_GenerateSakkeSSV

```
int wc_GenerateSakkeSSV(  
    SakkeKey * key,  
    WC_RNG * rng,  
    byte * ssv,  
    word16 * ssvSz  
)
```

C.45.2.28 function wc_SetSakkeRsk

```
int wc_SetSakkeRsk(  
    SakkeKey * key,  
    const ecc_point * rsk,  
    byte * table,  
    word32 len  
)
```

C.45.2.29 function wc_DeriveSakkeSSV

```
int wc_DeriveSakkeSSV(  
    SakkeKey * key,  
    enum wc_HashType hashType,  
    byte * ssv,  
    word16 ssvSz,  
    const byte * auth,  
    word16 authSz  
)
```

C.45.3 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.46 dox_comments/header_files-ja/sha256.h

C.46.1 Functions

	Name
int	wc_InitSha256 (wc_Sha256 * sha) この関数は SHA256 を初期化します。これは wc_Sha256Hash によって自動的に呼び出されます。
int	wc_Sha256Update (wc_Sha256 * sha, const byte * data, word32 len) 長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。
int	wc_Sha256Final (wc_Sha256 * sha256, byte * hash) データのハッシュ化を完了します。結果は hash に格納されます。sha256 構造体の状態をリセットします。
void	wc_Sha256Free (wc_Sha256 * sha256) Sha256 構造体をリセットします。注意: これは WOLFSSL_TI_HASH が定義されている場合にのみサポートされます。
int	wc_Sha256GetHash (wc_Sha256 * sha256, byte * hash) ハッシュデータを取得します。結果は hash に格納されます。sha256 構造体の状態はリセットされません。
int	wc_InitSha224 (wc_Sha224 * sha224) Sha224 構造体を初期化するために使用されます。
int	wc_Sha224Update (wc_Sha224 * sha224, const byte * data, word32 len) 長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。
int	wc_Sha224Final (wc_Sha224 * sha224, byte * hash) データのハッシュ化を完了します。結果は hash に格納されます。sha224 構造体の状態をリセットします。

C.46.2 Functions Documentation

C.46.2.1 function wc_InitSha256

```
int wc_InitSha256(  
    wc_Sha256 * sha  
)
```

この関数は SHA256 を初期化します。これは wc_Sha256Hash によって自動的に呼び出されます。

Parameters:

- **sha256** 暗号化に使用する sha256 構造体へのポインタ

See:

- **wc_Sha256Hash**
- **wc_Sha256Update**
- **wc_Sha256Final**

Return: 0 初期化に成功した場合に返されます

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.46.2.2 function wc_Sha256Update

```
int wc_Sha256Update(
    wc_Sha256 * sha,
    const byte * data,
    word32 len
)
```

長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。

Parameters:

- **sha256** 暗号化に使用する sha256 構造体へのポインタ
- **data** ハッシュ化されるデータ
- **len** ハッシュ化されるデータの長さ

See:

- [wc_Sha256Hash](#)
- [wc_Sha256Final](#)
- [wc_InitSha256](#)

Return: 0 ダイジェストへのデータ追加に成功した場合に返されます。

Example

```
Sha256 sha256[1];
byte data[] = { ハッシュ化されるデータ };
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.46.2.3 function wc_Sha256Final

```
int wc_Sha256Final(
    wc_Sha256 * sha256,
    byte * hash
)
```

データのハッシュ化を完了します。結果は hash に格納されます。sha256 構造体の状態をリセットします。

Parameters:

- **sha256** 暗号化に使用する sha256 構造体へのポインタ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Sha256Hash](#)
- [wc_Sha256GetHash](#)
- [wc_InitSha256](#)

Return: 0 完了に成功した場合に返されます。

Example

```
Sha256 sha256[1];
byte data[] = { ハッシュ化されるデータ };
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.46.2.4 function wc_Sha256Free

```
void wc_Sha256Free(
    wc_Sha256 * sha256
)
```

Sha256 構造体をリセットします。注意: これは WOLFSSL_TI_HASH が定義されている場合にのみサポートされます。

Parameters:

- **sha256** 解放する sha256 構造体へのポインタ。

See:

- [wc_InitSha256](#)
- [wc_Sha256Update](#)
- [wc_Sha256Final](#)

Return: none 戻り値なし。

Example

```
Sha256 sha256;
byte data[] = { ハッシュ化されるデータ };
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.46.2.5 function wc_Sha256GetHash

```
int wc_Sha256GetHash(  
    wc_Sha256 * sha256,  
    byte * hash  
)
```

ハッシュデータを取得します。結果は hash に格納されます。sha256 構造体の状態はリセットされません。

Parameters:

- **sha256** 暗号化に使用する sha256 構造体へのポインタ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Sha256Hash](#)
- [wc_Sha256Final](#)
- [wc_InitSha256](#)

Return: 0 完了に成功した場合に返されます。

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.46.2.6 function wc_InitSha224

```
int wc_InitSha224(  
    wc_Sha224 * sha224  
)
```

Sha224 構造体を初期化するために使用されます。

Parameters:

- **sha224** 初期化する Sha224 構造体へのポインタ。

See:

- [wc_Sha224Hash](#)
- [wc_Sha224Update](#)
- [wc_Sha224Final](#)

Return:

- 0 成功
- 1 sha224 が null のためにエラーが返されます。

Example

```

Sha224 sha224;
if(wc_InitSha224(&sha224) != 0)
{
    // エラーを処理
}

```

C.46.2.7 function wc_Sha224Update

```

int wc_Sha224Update(
    wc_Sha224 * sha224,
    const byte * data,
    word32 len
)

```

長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。

Parameters:

- **sha224** 暗号化に使用する Sha224 構造体へのポインタ。
- **data** ハッシュ化されるデータ。
- **len** ハッシュ化されるデータの長さ。

See:

- [wc_InitSha224](#)
- [wc_Sha224Final](#)
- [wc_Sha224Hash](#)

Return:

- 0 成功
- 1 関数が失敗した場合にエラーが返されます。
- BAD_FUNC_ARG sha224 または data が null の場合にエラーが返されます。

Example

```

Sha224 sha224;
byte data[] = { /* ハッシュ化されるデータ */};
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.46.2.8 function wc_Sha224Final

```

int wc_Sha224Final(
    wc_Sha224 * sha224,
    byte * hash
)

```

データのハッシュ化を完了します。結果は hash に格納されます。sha224 構造体の状態をリセットします。

Parameters:

- **sha224** 暗号化に使用する sha224 構造体へのポインタ

- **hash** ハッシュ値を保持するバイト配列。

See:

- `wc_InitSha224`
- `wc_Sha224Hash`
- `wc_Sha224Update`

Return:

- 0 成功
- <0 エラー

Example

```

Sha224 sha224;
byte data[] = { /* ハッシュ化されるデータ */;
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.46.3 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);

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);

```

C.47 dox_comments/header_files-ja/sha512.h**C.47.1 Functions**

	Name
int	wc_InitSha512 (wc_Sha512 * sha) この関数は SHA512 を初期化します。これは wc_Sha512Hash によって自動的に呼び出されます。

	Name
int	wc_Sha512Update (wc_Sha512 * sha, const byte * data, word32 len) 長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。
int	wc_Sha512Final (wc_Sha512 * sha512, byte * hash) データのハッシュ化を完了します。結果は hash に格納されます。
int	wc_InitSha384 (wc_Sha384 * sha) この関数は SHA384 を初期化します。これは wc_Sha384Hash によって自動的に呼び出されます。
int	wc_Sha384Update (wc_Sha384 * sha, const byte * data, word32 len) 長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。
int	wc_Sha384Final (wc_Sha384 * sha384, byte * hash) データのハッシュ化を完了します。結果は hash に格納されます。

C.47.2 Functions Documentation

C.47.2.1 function wc_InitSha512

```
int wc_InitSha512(
    wc_Sha512 * sha
)
```

この関数は SHA512 を初期化します。これは wc_Sha512Hash によって自動的に呼び出されます。

Parameters:

- **sha512** 暗号化に使用する sha512 構造体へのポインタ

See:

- **wc_Sha512Hash**
- **wc_Sha512Update**
- **wc_Sha512Final**

Return: 0 初期化に成功した場合に返されます

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.47.2.2 function wc_Sha512Update

```
int wc_Sha512Update(
    wc_Sha512 * sha,
```

```

    const byte * data,
    word32 len
)

```

長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。

Parameters:

- **sha512** 暗号化に使用する sha512 構造体へのポインタ
- **data** ハッシュ化されるデータ
- **len** ハッシュ化されるデータの長さ

See:

- [wc_Sha512Hash](#)
- [wc_Sha512Final](#)
- [wc_InitSha512](#)

Return: 0 ダイジェストへのデータ追加に成功した場合に返されます。

Example

```

Sha512 sha512[1];
byte data[] = { ハッシュ化されるデータ };
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.47.2.3 function wc_Sha512Final

```

int wc_Sha512Final(
    wc_Sha512 * sha512,
    byte * hash
)

```

データのハッシュ化を完了します。結果は hash に格納されます。

Parameters:

- **sha512** 暗号化に使用する sha512 構造体へのポインタ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Sha512Hash](#)
- [wc_Sha512Final](#)
- [wc_InitSha512](#)

Return: 0 ハッシュの完了に成功した場合に返されます。

Example

```

Sha512 sha512[1];
byte data[] = { ハッシュ化されるデータ };
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.47.2.4 function wc_InitSha384

```
int wc_InitSha384(
    wc_Sha384 * sha
)
```

この関数は SHA384 を初期化します。これは wc_Sha384Hash によって自動的に呼び出されます。

Parameters:

- **sha384** 暗号化に使用する sha384 構造体へのポインタ

See:

- [wc_Sha384Hash](#)
- [wc_Sha384Update](#)
- [wc_Sha384Final](#)

Return: 0 初期化に成功した場合に返されます

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.47.2.5 function wc_Sha384Update

```
int wc_Sha384Update(
    wc_Sha384 * sha,
    const byte * data,
    word32 len
)
```

長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。

Parameters:

- **sha384** 暗号化に使用する sha384 構造体へのポインタ
- **data** ハッシュ化されるデータ
- **len** ハッシュ化されるデータの長さ

See:

- [wc_Sha384Hash](#)
- [wc_Sha384Final](#)
- [wc_InitSha384](#)

Return: 0 ダイジェストへのデータ追加に成功した場合に返されます。

Example

```
Sha384 sha384[1];
byte data[] = { ハッシュ化されるデータ };
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.47.2.6 function wc_Sha384Final

```
int wc_Sha384Final(
    wc_Sha384 * sha384,
    byte * hash
)
```

データのハッシュ化を完了します。結果は hash に格納されます。

Parameters:

- **sha384** 暗号化に使用する sha384 構造体へのポインタ
- **hash** ハッシュ値を保持するバイト配列。

See:

- [wc_Sha384Hash](#)
- [wc_Sha384Final](#)
- [wc_InitSha384](#)

Return: 0 完了に成功した場合に返されます。

Example

```
Sha384 sha384[1];
byte data[] = { ハッシュ化されるデータ };
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.47.3 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);

```

C.48 dox_comments/header_files-ja/sha.h

C.48.1 Functions

	Name
int	wc_InitSha (wc_Sha * sha) この関数は SHA を初期化します。これは wc_ShaHash によって自動的に呼び出されます。
int	wc_ShaUpdate (wc_Sha * sha, const byte * data, word32 len) 長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。
int	wc_ShaFinal (wc_Sha * sha, byte * hash) データのハッシュ化を完了します。結果は hash に格納されます。sha 構造体の状態をリセットします。
void	wc_ShaFree (wc_Sha * sha) 初期化された Sha 構造体によって使用されるメモリをクリーンアップするために使用されます。
int	wc_ShaGetHash (wc_Sha * sha, byte * hash) ハッシュデータを取得します。結果は hash に格納されます。sha 構造体の状態はリセットされません。

C.48.2 Functions Documentation

C.48.2.1 function wc_InitSha

```

int wc_InitSha(
    wc_Sha * sha
)

```

この関数は SHA を初期化します。これは wc_ShaHash によって自動的に呼び出されます。

Parameters:

- **sha** 暗号化に使用する sha 構造体へのポインタ

See:

- **wc_ShaHash**
- **wc_ShaUpdate**
- **wc_ShaFinal**

Return: 0 初期化に成功した場合に返されます

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.48.2.2 function wc_ShaUpdate

```
int wc_ShaUpdate(
    wc_Sha * sha,
    const byte * data,
    word32 len
)
```

長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。

Parameters:

- **sha** 暗号化に使用する sha 構造体へのポインタ
- **data** ハッシュ化されるデータ
- **len** ハッシュ化されるデータの長さ

See:

- [wc_ShaHash](#)
- [wc_ShaFinal](#)
- [wc_InitSha](#)

Return: 0 ダイジェストへのデータ追加に成功した場合に返されます。

Example

```
Sha sha[1];
byte data[] = { // ハッシュ化されるデータ };
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.48.2.3 function wc_ShaFinal

```
int wc_ShaFinal(
    wc_Sha * sha,
    byte * hash
)
```

データのハッシュ化を完了します。結果は hash に格納されます。sha 構造体の状態をリセットします。

Parameters:

- **sha** 暗号化に使用する sha 構造体へのポインタ
- **hash** ハッシュ値を保持するバイト配列。

See:

- `wc_ShaHash`
- `wc_InitSha`
- `wc_ShaGetHash`

Return: 0 完了に成功した場合に返されます。

Example

```
Sha sha[1];
byte data[] = { ハッシュ化されるデータ };
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.48.2.4 function `wc_ShaFree`

```
void wc_ShaFree(
    wc_Sha * sha
)
```

初期化された Sha 構造体によって使用されるメモリをクリーンアップするために使用されます。

Parameters:

- **sha** 解放する Sha 構造体へのポインタ。

See:

- `wc_InitSha`
- `wc_ShaUpdate`
- `wc_ShaFinal`

Return: 戻り値なし。

Example

```
Sha sha;
wc_InitSha(&sha);
// sha を使用
wc_ShaFree(&sha);
```

C.48.2.5 function `wc_ShaGetHash`

```
int wc_ShaGetHash(
    wc_Sha * sha,
    byte * hash
)
```

ハッシュデータを取得します。結果は hash に格納されます。sha 構造体の状態はリセットされません。

Parameters:

- **sha** 暗号化に使用する sha 構造体へのポインタ
- **hash** ハッシュ値を保持するバイト配列。

See:

- `wc_ShaHash`
- `wc_ShaFinal`
- `wc_InitSha`

Return: 0 完了に成功した場合に返されます。

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.48.3 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);
```

C.49 dox_comments/header_files-ja/signature.h

C.49.1 Functions

	Name
int	wc_SignatureGetSize (enum wc_SignatureType sig_type, const void * key, word32 key_len) この関数は、結果の署名の最大サイズを返します。
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, const void * key, word32 key_len) この関数は、データをハッシュ化し、結果のハッシュとキーを使用して署名を検証することで、署名を検証します。
int	wc_SignatureGenerate (enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, byte * sig, word32 * sig_len, const void * key, word32 key_len, WC_RNG * rng) この関数は、キーを使用してデータから署名を生成します。最初にデータのハッシュを作成し、次にキーを使用してハッシュに署名します。

C.49.2 Functions Documentation

C.49.2.1 function wc_SignatureGetSize

```
int wc_SignatureGetSize(  
    enum wc_SignatureType sig_type,  
    const void * key,  
    word32 key_len  
)
```

この関数は、結果の署名の最大サイズを返します。

Parameters:

- **sig_type** WC_SIGNATURE_TYPE_ECC や WC_SIGNATURE_TYPE_RSA などの署名タイプの列挙値。
- **key** ecc_key や RsaKey などのキー構造体へのポインタ。
- **key_len** キー構造体のサイズ。

See:

- [wc_HashGetDigestSize](#)
- [wc_SignatureGenerate](#)
- [wc_SignatureVerify](#)

Return: sig_type がサポートされていない場合、SIG_TYPE_E を返します。sig_type が無効な場合、BAD_FUNC_ARG を返します。正の戻り値は、署名の最大サイズを示します。

Example

```
// 署名の長さを取得  
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) {  
    // 成功  
}
```

C.49.2.2 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,  
    const void * key,  
    word32 key_len  
)
```

この関数は、データをハッシュ化し、結果のハッシュとキーを使用して署名を検証することで、署名を検証します。

Parameters:

- **hash_type** “WC_HASH_TYPE_SHA256” などの “enum wc_HashType” からのハッシュタイプ。
- **sig_type** WC_SIGNATURE_TYPE_ECC や WC_SIGNATURE_TYPE_RSA などの署名タイプの列挙値。
- **data** ハッシュ化するデータを含むバッファへのポインタ。

- **data_len** データバッファの長さ。
- **sig** 署名を出力するバッファへのポインタ。
- **sig_len** 署名出力バッファの長さ。
- **key** ecc_key や RsaKey などのキー構造体へのポインタ。
- **key_len** キー構造体のサイズ。

See:

- [wc_SignatureGetSize](#)
- [wc_SignatureGenerate](#)

Return:

- 0 成功
- SIG_TYPE_E -231、署名タイプが有効化されていない/利用できない
- BAD_FUNC_ARG -173、不正な関数引数が提供された
- BUFFER_E -132、出力バッファが小さすぎるか、入力が大きすぎる。

Example

```
int ret;
ecc_key eccKey;

// 公開鍵をインポート
wc_ecc_init(&eccKey);
ret = wc_ecc_import_x963(eccPubKeyBuf, eccPubKeyLen, &eccKey);
// 公開鍵を使用して署名検証を実行
ret = wc_SignatureVerify(
    WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
    fileBuf, fileLen,
    sigBuf, sigLen,
    &eccKey, sizeof(eccKey));
printf(" 署名検証: %s\n", (ret == 0) ? "合格" : "不合格", ret);
wc_ecc_free(&eccKey);
```

C.49.2.3 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,
    const void * key,
    word32 key_len,
    WC_RNG * rng
)
```

この関数は、キーを使用してデータから署名を生成します。最初にデータのハッシュを作成し、次にキーを使用してハッシュに署名します。

Parameters:

- **hash_type** “WC_HASH_TYPE_SHA256” などの “enum wc_HashType” からのハッシュタイプ。
- **sig_type** WC_SIGNATURE_TYPE_ECC や WC_SIGNATURE_TYPE_RSA などの署名タイプの列挙値。
- **data** ハッシュ化するデータを含むバッファへのポインタ。
- **data_len** データバッファの長さ。

- **sig** 署名を出力するバッファへのポインタ。
- **sig_len** 署名出力バッファの長さ。
- **key** ecc_key や RsaKey などのキー構造体へのポインタ。
- **key_len** キー構造体のサイズ。
- **rng** 初期化された RNG 構造体へのポインタ。

See:

- [wc_SignatureGetSize](#)
- [wc_SignatureVerify](#)

Return:

- 0 成功
- SIG_TYPE_E -231、署名タイプが有効化されていない/利用できない
- BAD_FUNC_ARG -173、不正な関数引数が提供された
- BUFFER_E -132、出力バッファが小さすぎるか、入力が大きすぎる。

Example

```
int ret;
WC_RNG rng;
ecc_key eccKey;

wc_InitRng(&rng);
wc_ecc_init(&eccKey);

// キーを生成
ret = wc_ecc_make_key(&rng, 32, &eccKey);

// 署名の長さを取得してバッファを割り当て
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
sigBuf = malloc(sigLen);

// 公開鍵を使用して署名検証を実行
ret = wc_SignatureGenerate(
    WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
    fileBuf, fileLen,
    sigBuf, &sigLen,
    &eccKey, sizeof(eccKey),
    &rng);
printf(" 署名生成: %s\n", (ret == 0) ? "合格" : "不合格", ret);

free(sigBuf);
wc_ecc_free(&eccKey);
wc_FreeRng(&rng);
```

C.49.3 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,
    const 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,
    const void* key, word32 key_len,
    WC_RNG* rng);

```

C.50 dox_comments/header_files-ja/siphash.h

C.50.1 Functions

	Name
int	wc_InitSipHash (SipHash * siphash, const unsigned char * key, unsigned char outSz) この関数は、MAC サイズに対するキーを使用して SipHash を初期化します。
int	wc_SipHashUpdate (SipHash * siphash, const unsigned char * in, word32 inSz) 長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。
int	wc_SipHashFinal (SipHash * siphash, unsigned char * out, unsigned char outSz) データの MAC 処理を完了します。結果は out に格納されます。
int	wc_SipHash (const unsigned char * key, const unsigned char * in, word32 inSz, unsigned char * out, unsigned char outSz) この関数は、キーに基づいて MAC を計算するために、SipHash を使用してデータをワンショットで処理します。

C.50.2 Functions Documentation

C.50.2.1 function wc_InitSipHash

```

int wc_InitSipHash(
    SipHash * siphash,
    const unsigned char * key,
    unsigned char outSz
)

```

この関数は、MAC サイズに対するキーを使用して SipHash を初期化します。

Parameters:

- **siphash** MAC に使用する SipHash 構造体へのポインタ
- **key** 16 バイト配列へのポインタ
- **outSz** MAC として出力するバイト数

See:

- **wc_SipHash**
- **wc_SipHashUpdate**
- **wc_SipHashFinal**

Return:

- 0 初期化に成功した場合に返されます
- BAD_FUNC_ARG siphash または key が NULL の場合に返されます
- BAD_FUNC_ARG outSz が 8 でも 16 でもない場合に返されます

Example

```
SipHash siphash[1];
unsigned char key[16] = { ... };
byte macSz = 8; // 8 または 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.50.2.2 function wc_SipHashUpdate

```
int wc_SipHashUpdate(
    SipHash * siphash,
    const unsigned char * in,
    word32 inSz
)
```

長さ len の提供されたバイト配列を継続的にハッシュするために呼び出すことができます。

Parameters:

- **siphash** MAC に使用する SipHash 構造体へのポインタ
- **in** MAC するデータ
- **inSz** MAC するデータのサイズ

See:

- [wc_SipHash](#)
- [wc_InitSipHash](#)
- [wc_SipHashFinal](#)

Return:

- 0 MAC へのデータ追加に成功した場合に返されます
- BAD_FUNC_ARG siphash が NULL の場合に返されます
- BAD_FUNC_ARG in が NULL で inSz がゼロでない場合に返されます

Example

```
SipHash siphash[1];
byte data[] = { MAC するデータ };
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.50.2.3 function wc_SipHashFinal

```

int wc_SipHashFinal(
    SipHash * siphash,
    unsigned char * out,
    unsigned char outSz
)

```

データの MAC 処理を完了します。結果は out に格納されます。

Parameters:

- **siphash** MAC に使用する SipHash 構造体へのポインタ
- **out** MAC 値を保持するバイト配列
- **outSz** MAC として出力するバイト数

See:

- [wc_SipHash](#)
- [wc_InitSipHash](#)
- [wc_SipHashUpdate](#)

Return:

- 0 完了に成功した場合に返されます。
- BAD_FUNC_ARG siphash または out が NULL の場合に返されます
- BAD_FUNC_ARG outSz が初期化された値と同じでない場合に返されます

Example

```

SipHash siphash[1];
byte mac[8] = { ... }; // 8 または 16 バイト
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.50.2.4 function wc_SipHash

```

int wc_SipHash(
    const unsigned char * key,
    const unsigned char * in,
    word32 inSz,
    unsigned char * out,
    unsigned char outSz
)

```

この関数は、キーに基づいて MAC を計算するために、SipHash を使用してデータをワンショットで処理します。

Parameters:

- **key** 16 バイト配列へのポインタ
- **in** MAC するデータ
- **inSz** MAC するデータのサイズ
- **out** MAC 値を保持するバイト配列
- **outSz** MAC として出力するバイト数

See:

- [wc_InitSipHash](#)
- [wc_SipHashUpdate](#)
- [wc_SipHashFinal](#)

Return:

- 0 MAC に成功した場合に返されます
- BAD_FUNC_ARG key または out が NULL の場合に返されます
- BAD_FUNC_ARG in が NULL で inSz がゼロでない場合に返されます
- BAD_FUNC_ARG outSz が 8 でも 16 でもない場合に返されます

Example

```
unsigned char key[16] = { ... };
byte data[] = { MAC するデータ };
word32 len = sizeof(data);
byte mac[8] = { ... }; // 8 または 16 バイト
byte macSz = sizeof(mac);

if ((ret = wc_SipHash(key, data, len, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHash failed");
}
```

C.50.3 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.51 dox_comments/header_files-ja/srp.h

C.51.1 Functions

	Name
int	wc_SrpInit (Srp * srp, SrpType type, SrpSide side) 使用のために Srp 構造体を初期化します。
void	wc_SrpTerm (Srp * srp) 使用後に Srp 構造体のリソースを解放します。
int	wc_SrpSetUsername (Srp * srp, const byte * username, word32 size) ユーザー名を設定します。この関数は wc_SrpInit の後に呼び出す必要があります。
int	wc_SrpSetParams (Srp * srp, const byte * N, word32 nSz, const byte * g, word32 gSz, const byte * salt, word32 saltSz) ユーザー名に基づいて srp パラメータを設定します。wc_SrpSetUsername の後に呼び出す必要があります。
int	wc_SrpSetPassword (Srp * srp, const byte * password, word32 size) パスワードを設定します。パスワードの設定は、srp 構造体にクリアパスワードデータを永続化しません。クライアントは $x = H(\text{salt} + H(\text{user}; \text{pswd}))$ を計算し、auth フィールドに格納します。この関数は wc_SrpSetParams の後に呼び出す必要があり、クライアント側のみです。
int	wc_SrpSetVerifier (Srp * srp, const byte * verifier, word32 size) 検証子を設定します。この関数は wc_SrpSetParams の後に呼び出す必要があり、サーバー側のみです。
int	wc_SrpGetVerifier (Srp * srp, byte * verifier, word32 * size) 検証子を取得します。クライアントは $v = g^x \% N$ で検証子を計算します。この関数は wc_SrpSetPassword の後に呼び出すことができ、クライアント側のみです。
int	wc_SrpSetPrivate (Srp * srp, const byte * priv, word32 size) 秘密エフェメラル値を設定します。秘密エフェメラル値は次のように知られています: クライアント側では a。a = random() サーバー側では b。b = random() この関数は単体テストケースや、開発者が外部の乱数ソースを使用してエフェメラル値を設定したい場合に便利です。この関数は wc_SrpGetPublic の前に呼び出すことができます。
int	wc_SrpGetPublic (Srp * srp, byte * pub, word32 * size) 公開エフェメラル値を取得します。公開エフェメラル値は次のように知られています: クライアント側では A。A = $g^a \% N$ サーバー側では B。B = $(k * v + (g^b \% N)) \% N$ この関数は wc_SrpSetPassword または wc_SrpSetVerifier の後に呼び出す必要があります。関数 wc_SrpSetPrivate は wc_SrpGetPublic の前に呼び出すことができます。

	Name
int	wc_SrpComputeKey (Srp * srp, byte * clientPubKey, word32 clientPubKeySz, byte * serverPubKey, word32 serverPubKeySz) セッション鍵を計算します。鍵は成功後に srp->key でアクセスできます。
int	wc_SrpGetProof (Srp * srp, byte * proof, word32 * size) 証明を取得します。この関数は wc_SrpComputeKey の後に呼び出す必要があります。
int	wc_SrpVerifyPeersProof (Srp * srp, byte * proof, word32 size) ピアの証明を検証します。この関数は wc_SrpGetSessionKey の前に呼び出す必要があります。

C.51.2 Functions Documentation

C.51.2.1 function wc_SrpInit

```
int wc_SrpInit(
    Srp * srp,
    SrpType type,
    SrpSide side
)
```

使用のために Srp 構造体を初期化します。

Parameters:

- **srp** 初期化する Srp 構造体。
- **type** 使用するハッシュタイプ。
- **side** 通信の側。

See:

- [wc_SrpTerm](#)
- [wc_SrpSetUsername](#)

Return:

- 0 成功時。
- BAD_FUNC_ARG srp が null の場合や SrpSide が SRP_CLIENT_SIDE または SRP_SERVER_SIDE でない場合など、引数に問題がある場合に返されます。
- NOT_COMPILED_IN 引数として渡された型が wolfCrypt ビルドで設定されていない場合に返されます。
- <0 エラー時。

Example

```
Srp srp;
if (wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE) != 0)
{
    // 初期化エラー
}
else
{
    wc_SrpTerm(&srp);
}
```

C.51.2.2 function wc_SrpTerm

```
void wc_SrpTerm(  
    Srp * srp  
)
```

使用後に Srp 構造体のリソースを解放します。

Parameters:

- **srp** 終了する Srp 構造体へのポインタ。

See: [wc_SrpInit](#)

Return: none 戻り値なし。

Example

```
Srp srp;  
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);  
// srp を使用  
wc_SrpTerm(&srp)
```

C.51.2.3 function wc_SrpSetUsername

```
int wc_SrpSetUsername(  
    Srp * srp,  
    const byte * username,  
    word32 size  
)
```

ユーザー名を設定します。この関数は wc_SrpInit の後に呼び出す必要があります。

Parameters:

- **srp** Srp 構造体。
- **username** ユーザー名を含むバッファ。
- **size** ユーザー名のサイズ (バイト単位)

See:

- [wc_SrpInit](#)
- [wc_SrpSetParams](#)
- [wc_SrpTerm](#)

Return:

- 0 ユーザー名が正常に設定されました。
- BAD_FUNC_ARG: srp または username が null の場合に返されます。
- MEMORY_E: srp->user のメモリ割り当てに問題がある場合に返されます
- < 0: エラー。

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)  
{  
    // ユーザー名の設定エラーが発生しました。}
```

```

}
wc_SrpTerm(&srp);

```

C.51.2.4 function wc_SrpSetParams

```

int wc_SrpSetParams(
    Srp * srp,
    const byte * N,
    word32 nSz,
    const byte * g,
    word32 gSz,
    const byte * salt,
    word32 saltSz
)

```

ユーザー名に基づいて srp パラメータを設定します。wc_SrpSetUsername の後に呼び出す必要があります。

Parameters:

- **srp** Srp 構造体。
- **N** モジュラス。N = 2q+1、[q、N] は素数。
- **nSz** N のサイズ (バイト単位)。
- **g** N を法とする生成元。
- **gSz** g のサイズ (バイト単位)
- **salt** 小さなランダムソルト。各ユーザー名に固有。
- **saltSz** ソルトのサイズ (バイト単位)

See:

- [wc_SrpInit](#)
- [wc_SrpSetUsername](#)
- [wc_SrpTerm](#)

Return:

- 0 成功
- BAD_FUNC_ARG srp、N、g、または salt が null の場合、または nSz < gSz の場合に返されます。
- SRP_CALL_ORDER_E wc_SrpSetUsername の前に wc_SrpSetParams が呼び出された場合に返されます。
- <0 エラー

Example

```

Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // バイト配列 N の内容
byte g[] = { }; // バイト配列 g の内容
byte salt[] = { }; // バイト配列 salt の内容

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)
{
    // パラメータ設定エラー
}

```

```

}
wc_SrpTerm(&srp);

```

C.51.2.5 function wc_SrpSetPassword

```

int wc_SrpSetPassword(
    Srp * srp,
    const byte * password,
    word32 size
)

```

パスワードを設定します。パスワードの設定は、srp 構造体にクリアパスワードデータを永続化しません。クライアントは $x = H(\text{salt} + H(\text{user}:\text{pswd}))$ を計算し、auth フィールドに格納します。この関数は wc_SrpSetParams の後に呼び出す必要があり、クライアント側のみです。

Parameters:

- **srp** Srp 構造体。
- **password** パスワードを含むバッファ。
- **size** パスワードのサイズ (バイト単位)。

See:

- [wc_SrpInit](#)
- [wc_SrpSetUsername](#)
- [wc_SrpSetParams](#)

Return:

- 0 成功
- BAD_FUNC_ARG srp または password が null の場合、または srp->side が SRP_CLIENT_SIDE に設定されていない場合に返されます。
- SRP_CALL_ORDER_E wc_SrpSetPassword が順序外で呼び出された場合に返されます。
- <0 エラー

Example

```

Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;

byte N[] = { }; // バイト配列 N の内容
byte g[] = { }; // バイト配列 g の内容
byte salt[] = { }; // バイト配列 salt の内容

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)
{
    // パスワード設定エラー
}

wc_SrpTerm(&srp);

```


C.51.2.6 function wc_SrpSetVerifier

```
int wc_SrpSetVerifier(
    Srp * srp,
    const byte * verifier,
    word32 size
)
```

検証子を設定します。この関数は wc_SrpSetParams の後に呼び出す必要があり、サーバー側のみです。

Parameters:

- **srp** Srp 構造体。
- **verifier** 検証子を含む構造体。
- **size** 検証子のサイズ (バイト単位)。

See:

- [wc_SrpInit](#)
- [wc_SrpSetParams](#)
- [wc_SrpGetVerifier](#)

Return:

- 0 成功
- BAD_FUNC_ARG srp または verifier が null の場合、または srp->side が SRP_SERVER_SIDE でない場合に返されます。
- <0 エラー

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // バイト配列 N の内容
byte g[] = { }; // バイト配列 g の内容
byte salt[] = { }; // バイト配列 salt の内容
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[] = { }; // 何らかの検証子の内容

if(wc_SrpSetVerifier(&srp, verifier, sizeof(verifier)) != 0)
{
    // 検証子設定エラー
}

wc_SrpTerm(&srp);
```

C.51.2.7 function wc_SrpGetVerifier

```
int wc_SrpGetVerifier(
    Srp * srp,
    byte * verifier,
    word32 * size
)
```

検証子を取得します。クライアントは $v = g^x \% N$ で検証子を計算します。この関数は wc_SrpSetPassword の後に呼び出すことができ、クライアント側のみです。

Parameters:

- **srp** Srp 構造体。
- **verifier** 検証子を書き込むバッファ。
- **size** バッファサイズ (バイト単位)。検証子のサイズで更新されます。

See:

- [wc_SrpSetVerifier](#)
- [wc_SrpSetPassword](#)

Return:

- 0 成功
- BAD_FUNC_ARG srp、verifier、または size が null の場合、または srp->side が SRP_CLIENT_SIDE でない場合に返されます。
- SRP_CALL_ORDER_E wc_SrpGetVerifier が順序外で呼び出された場合に返されます。
- <0 エラー

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;

byte N[] = { }; // バイト配列 N の内容
byte g[] = { }; // バイト配列 g の内容
byte salt[] = { }; // バイト配列 salt の内容
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)
{
    // 検証子取得エラー
}
wc_SrpTerm(&srp);
```

C.51.2.8 function wc_SrpSetPrivate

```
int wc_SrpSetPrivate(
    Srp * srp,
    const byte * priv,
    word32 size
)
```

秘密エフェメラル値を設定します。秘密エフェメラル値は次のように知られています: クライアント側では a。a = random() サーバー側では b。b = random() この関数は単体テストケースや、開発者が外部の乱数ソースを使用してエフェメラル値を設定したい場合に便利です。この関数は wc_SrpGetPublic の前に呼び出すことができます。

Parameters:

- **srp** Srp 構造体。
- **priv** エフェメラル値。
- **size** private のサイズ (バイト単位)。

See: [wc_SrpGetPublic](#)

Return:

- 0 成功
- BAD_FUNC_ARG srp、private、または size が null の場合に返されます。
- SRP_CALL_ORDER_E wc_SrpSetPrivate が順序外で呼び出された場合に返されます。
- <0 エラー

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // バイト配列 N の内容
byte g[] = { }; // バイト配列 g の内容
byte salt[] = { }; // バイト配列 salt の内容
byte verifier = { }; // 何らかの検証子の内容
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[] = { }; // 何らかのエフェメラル値
if( wc_SrpSetPrivate(&srp, b, sizeof(b)) != 0)
{
    // 秘密エフェメラル設定エラー
}

wc_SrpTerm(&srp);
```

C.51.2.9 function wc_SrpGetPublic

```
int wc_SrpGetPublic(
    Srp * srp,
    byte * pub,
    word32 * size
)
```

公開エフェメラル値を取得します。公開エフェメラル値は次のように知られています: クライアント側では $A = g^a \% N$ サーバー側では $B = (k * v + (g^b \% N)) \% N$ この関数は wc_SrpSetPassword または wc_SrpSetVerifier の後に呼び出す必要があります。関数 wc_SrpSetPrivate は wc_SrpGetPublic の前に呼び出すことができます。

Parameters:

- **srp** Srp 構造体。
- **pub** 公開エフェメラル値を書き込むバッファ。
- **size** バッファサイズ (バイト単位)。エフェメラル値のサイズで更新されます。

See:

- [wc_SrpSetPrivate](#)
- [wc_SrpSetPassword](#)

- `wc_SrpSetVerifier`

Return:

- 0 成功
- BAD_FUNC_ARG srp、pub、または size が null の場合に返されます。
- SRP_CALL_ORDER_E wc_SrpGetPublic が順序外で呼び出された場合に返されます。
- BUFFER_E size < srp.N の場合に返されます。
- <0 エラー

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;

byte N[] = { }; // バイト配列 N の内容
byte g[] = { }; // バイト配列 g の内容
byte salt[] = { }; // バイト配列 salt の内容
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 = 0;

if( wc_SrpGetPublic(&srp, public, &publicSz) != 0)
{
    // 公開エフェメラル取得エラー
}

wc_SrpTerm(&srp);
```

C.51.2.10 function wc_SrpComputeKey

```
int wc_SrpComputeKey(
    Srp * srp,
    byte * clientPubKey,
    word32 clientPubKeySz,
    byte * serverPubKey,
    word32 serverPubKeySz
)
```

セッション鍵を計算します。鍵は成功後に srp->key でアクセスできます。

Parameters:

- **srp** Srp 構造体。
- **clientPubKey** クライアントの公開エフェメラル値。
- **clientPubKeySz** クライアントの公開エフェメラル値のサイズ。
- **serverPubKey** サーバーの公開エフェメラル値。
- **serverPubKeySz** サーバーの公開エフェメラル値のサイズ。

See: `wc_SrpGetPublic`

Return:

- 0 成功
- BAD_FUNC_ARG srp、clientPubKey、または serverPubKey が null の場合、または clientPubKeySz または serverPubKeySz が 0 の場合に返されます。
- SRP_CALL_ORDER_E wc_SrpComputeKey が順序外で呼び出された場合に返されます。
- <0 エラー

Example

```
Srp server;

byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;
byte N[] = { }; // バイト配列 N の内容
byte g[] = { }; // バイト配列 g の内容
byte salt[] = { }; // バイト配列 salt の内容
byte verifier[] = { }; // 何らかの検証子の内容
byte serverPubKey[] = { }; // サーバー公開鍵の内容
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.51.2.11 function wc_SrpGetProof

```
int wc_SrpGetProof(
    Srp * srp,
    byte * proof,
    word32 * size
)
```

証明を取得します。この関数は wc_SrpComputeKey の後に呼び出す必要があります。

Parameters:

- **srp** Srp 構造体。
- **proof** ピアの証明。
- **size** 証明のサイズ (バイト単位)。

See: [wc_SrpComputeKey](#)

Return:

- 0 成功
- BAD_FUNC_ARG srp、proof、または size が null の場合に返されます。
- BUFFER_E size が srp->type のハッシュサイズより小さい場合に返されます。
- <0 エラー

Example

```

Srp cli;
byte clientProof[SRP_MAX_DIGEST_SIZE];
word32 clientProofSz = SRP_MAX_DIGEST_SIZE;

// 前の例のステップに従って Srp を初期化

if (wc_SrpGetProof(&cli, clientProof, &clientProofSz) != 0)
{
    // 証明取得エラー
}

```

C.51.2.12 function wc_SrpVerifyPeersProof

```

int wc_SrpVerifyPeersProof(
    Srp * srp,
    byte * proof,
    word32 size
)

```

ピアの証明を検証します。この関数は wc_SrpGetSessionKey の前に呼び出す必要があります。

Parameters:

- **srp** Srp 構造体。
- **proof** ピアの証明。
- **size** 証明のサイズ (バイト単位)。

See:

- wc_SrpGetSessionKey
- wc_SrpGetProof
- wc_SrpTerm

Return:

- 0 成功
- <0 エラー

Example

```

Srp cli;
Srp srv;
byte clientProof[SRP_MAX_DIGEST_SIZE];
word32 clientProofSz = SRP_MAX_DIGEST_SIZE;

// 前の例のステップに従って Srp を初期化
// 最初に証明を取得
wc_SrpGetProof(&cli, clientProof, &clientProofSz)

if (wc_SrpVerifyPeersProof(&srv, clientProof, clientProofSz) != 0)
{
    // 証明検証エラー
}

```

C.51.3 Source code

```

int wc_SrpInit(Srp* srp, SrpType type, SrpSide side);

```

```

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.52 dox_comments/header_files-ja/ssl.h

C.52.1 Functions

	Name
WOLFSSL_METHOD *	wolfDTLSv1_2_client_method_ex (void * heap) この関数は DTLS v1.2 クライアントメソッドを初期化します。
WOLFSSL_METHOD *	wolfSSLv23_method (void) この関数は wolfSSLv23_client_method に似た WOLFSSL_METHOD を返しますが、どちら側 (サーバ/クライアント) であるかがまだ決定されていない点が異なります。
WOLFSSL_METHOD *	**wolfSSLv3_server_method 関数は、アプリケーションがサーバであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfSSLv3_client_method 関数は、アプリケーションがクライアントであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

	Name
WOLFSSL_METHOD *	**wolfTLSv1_server_method 関数は、アプリケーションがサーバであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_client_method 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_1_server_method 関数は、アプリケーションがサーバであり、TLS 1.1 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_1_client_method 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_2_server_method 関数は、アプリケーションがサーバであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_2_client_method 関数は、アプリケーションがクライアントであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfDTLSv1_client_method 関数は、アプリケーションがクライアントであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポート付きでコンパイルされている場合 (-enable_dtls、または wolfSSL_DTLS を定義することによって) にのみ利用可能です。

	Name
WOLFSSL_METHOD *	**wolfDTLSv1_server_method関数は、アプリケーションがサーバーであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls、または wolfSSL_DTLS を定義することによって)。
WOLFSSL_METHOD *	**wolfDTLSv1_3_server_method関数は、アプリケーションがサーバーであり、DTLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLSv1.3 サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls13、または wolfSSL_DTLS13 を定義することによって)。
WOLFSSL_METHOD *	**wolfDTLSv1_3_client_method関数は、アプリケーションがクライアントであり、DTLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLSv1.3 サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls13、または wolfSSL_DTLS13 を定義することによって)。
WOLFSSL_METHOD *	**wolfDTLS_server_method関数は、アプリケーションがサーバーであり、利用可能な最新バージョンの DTLS および許可される最小バージョンまでのすべてのバージョンをサポートすることを示すために使用されます。デフォルトの許可される最小バージョンは、WOLFSSL_MIN_DTLS_DOWNGRADE の定義に基づいており、wolfSSL_SetMinVersion() を使用して実行時に変更できます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls、または wolfSSL_DTLS を定義することによって)。

	Name
WOLFSSL_METHOD *	**wolfDTLS_client_method関数は、アプリケーションがクライアントであり、利用可能な最新バージョンの DTLS および許可される最小バージョンまでのすべてのバージョンをサポートすることを示すために使用されます。デフォルトの許可される最小バージョンは、WOLFSSL_MIN_DTLS_DOWNGRADE の定義に基づいており、wolfSSL_SetMinVersion() を使用して実行時に変更できます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls、または wolfSSL_DTLS を定義することによって)。
WOLFSSL_METHOD *	wolfDTLSv1_2_server_method(void) この関数は、サーバー側用の WOLFSSL_METHOD を作成し、初期化します。
int	wolfSSL_use_old_poly(WOLFSSL * ssl, int value)chacha-poly AEAD 構築の最初のリリースと新しいバージョンの間にはいくつかの違いがあるため、古いバージョンを使用するサーバー/クライアントと通信するためのオプションを追加しました。デフォルトでは、wolfSSL は新しいバージョンを使用します。
int	wolfSSL_dtls_import(WOLFSSL * ssl, const unsigned char * buf, unsigned int sz)wolfSSL_dtls_import() 関数は、シリアル化されたセッション状態を解析するために使用されます。これにより、ハンドシェイクが完了した後に接続を再開できます。
int	wolfSSL_tls_import(WOLFSSL * ssl, const unsigned char * buf, unsigned int sz)シリアル化された TLS セッションをインポートするために使用されます。この関数は、接続の状態をインポートするためのものです。警告：buf には状態に関する機密情報が含まれており、保存する場合は暗号化してから保存するのが最善です。マクロ WOLFSSL_SESSION_EXPORT_DEBUG を定義することで、追加のデバッグ情報を表示できます。
int	wolfSSL_CTX_dtls_set_export(WOLFSSL_CTX * ctx, wc_dtls_export func)wolfSSL_CTX_dtls_set_export() 関数は、セッションをエクスポートするためのコールバック関数を設定するために使用されます。以前に保存されたエクスポート関数をクリアするために、パラメータ func として NULL を渡すことができます。サーバー側で使用され、ハンドシェイクが完了した直後に呼び出されます。

	Name
int	wolfSSL_dtls_set_export (WOLFSSL * ssl, wc_dtls_export func)wolfSSL_dtls_set_export() 関数は、セッションをエクスポートするためのコールバック関数を設定するために使用されます。以前に保存されたエクスポート関数をクリアするために、パラメータ func として NULL を渡すことができます。サーバー側で使用され、ハンドシェイクが完了した直後に呼び出されます。
int	wolfSSL_dtls_export (WOLFSSL * ssl, unsigned char * buf, unsigned int * sz)wolfSSL_dtls_export() 関数は、WOLFSSL セッションを提供されたバッファにシリアル化するために使用されます。セッションを送信するための関数コールバックを使用するよりもメモリオーバーヘッドが少なく、セッションをシリアル化するタイミングを選択できます。関数に渡されたときに buffer が NULL の場合、sz は WOLFSSL セッションをシリアル化するために必要なバッファのサイズに設定されます。
int	wolfSSL_tls_export (WOLFSSL * ssl, unsigned char * buf, unsigned int * sz) シリアル化された TLS セッションをエクスポートするために使用されます。この関数は接続のシリアル化された状態をエクスポートするためのものです。ほとんどの場合、wolfSSL_tls_export の代わりに wolfSSL_get1_session を使用する必要があります。追加のデバッグ情報は、マクロ WOLFSSL_SESSION_EXPORT_DEBUG が定義されている場合に表示できます。警告: buf には状態に関する機密情報が含まれているため、保存する場合は暗号化してから保存することが最善です。

	Name
int	wolfSSL_CTX_load_static_memory (WOLFSSL_CTX ** ctx, wolfSSL_method_func method, unsigned char * buf, unsigned int sz, int flag, int max) この関数は、CTX 用の静的メモリを確保 するために使用されます。確保されたメモリは、 CTX のライフタイム中および CTX から作成された SSL オブジェクトに使用されます。NULL の ctx ポインタと wolfSSL_method_func 関数を渡すこ とで、CTX 自体の作成も静的メモリを使用します。 wolfSSL_method_func は、WOLFSSL_METHOD* (wolfSSL_method_func)(void heap); という関数シ グネチャを持ちます。max に 0 を渡すと、設定さ れていないかのように動作し、最大同時使用制限 は適用されません。渡された flag 値は、メモリ の使用方法と動作中の挙動を決定します。利用可能 なフラグは次のとおりです: 0 - デフォルトの一般 メモリ、WOLFMEM_IO_POOL - メッセージの送 受信時の入出力バッファに使用され、一般メモリ をオーバーライドするため、渡されたバッファ内 のすべてのメモリが IO に使用されます、 WOLFMEM_IO_FIXED - WOLFMEM_IO_POOL と 同じですが、各 SSL がライフタイム中に 2 つのバ ッファを保持します、WOLFMEM_TRACK_STATS - 各 SSL が実行中にメモリ統計を追跡します。
int	wolfSSL_CTX_is_static_memory (WOLFSSL_CTX * ctx, WOLFSSL_MEM_STATS * mem_stats) この 関数は接続の動作を変更せず、静的メモリの使用 状況に関する情報を収集するためにのみ使用され ます。
int	wolfSSL_is_static_memory (WOLFSSL * ssl, WOLFSSL_MEM_CONN_STATS * mem_stats)wolfSSL_is_static_memory は、SSL の静的メモリ使用状況に関する情報を収集するた めに使用されます。戻り値は静的メモリが使用さ れているかどうかを示し、 WOLFSSL_MEM_CONN_STATS は、静的メモリを ロードする際に親 CTX に WOLFMEM_TRACK_STATS フラグが渡された場合 にのみ入力されます。
int	wolfSSL_CTX_use_certificate_file (WOLFSSL_CTX * ctx, const char * file, int format) この関数は、 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは file 引数によって提供されます。format 引数はファイ ルのフォーマットタイプを指定します。 SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM のいずれかです。適切な使用方法については、 example を参照してください。

	Name
int	wolfSSL_CTX_use_PrivateKey_file (WOLFSSL_CTX * ctx, const char * file, int format) この関数は、秘密鍵ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは file 引数によって提供されます。format 引数はファイルのフォーマットタイプを指定します - SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用方法については、example を参照してください。
int	wolfSSL_CTX_load_verify_locations (WOLFSSL_CTX * ctx, const char * file, const char * path) この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである場合があります。同じファイルに複数の CA 証明書が含まれている場合、wolfSSL はファイル内に提示された順序でそれらをロードします。path 引数は、信頼されたルート CA の証明書を含むディレクトリの名前へのポインタです。file の値が NULL でない場合、必要であれば path を NULL として指定できます。path が指定され、ライブラリのビルド時に NO_WOLFSSL_DIR が定義されていない場合、wolfSSL は指定されたディレクトリにあるすべての CA 証明書をロードします。この関数はディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダー "-----BEGIN CERTIFICATE-----" を持つ PEM 形式の CERT_TYPE ファイルを想定しています。

	Name
int	wolfSSL_CTX_load_verify_locations_ex (WOLFSSL_CTX * ctx, const char * file, const char * path, word32 flags) この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである場合があります。同じファイルに複数の CA 証明書が含まれている場合、wolfSSL はファイル内に提示された順序でそれらをロードします。path 引数は、信頼されたルート CA の証明書を含むディレクトリの名前へのポインタです。file の値が NULL でない場合、必要でなければ path を NULL として指定できます。path が指定され、ライブラリのビルド時に NO_WOLFSSL_DIR が定義されていない場合、wolfSSL は指定されたディレクトリにあるすべての CA 証明書をロードします。この関数は、指定されたフラグに基づいてディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダー “—BEGIN CERTIFICATE—” を持つ PEM 形式の CERT_TYPE ファイルを想定しています。
const char **	wolfSSL_get_system_CA_dirs (word32 * num) この関数は、wolfSSL_CTX_load_system_CA_certs が呼び出されたときに wolfSSL がシステム CA 証明書を検索するディレクトリを表す文字列の配列へのポインタを返します。証明書をアクセス可能なシステムディレクトリに保存しないシステム (Apple プラットフォームなど) では、この関数は常に NULL を返します。
int	wolfSSL_CTX_load_system_CA_certs (WOLFSSL_CTX * ctx) ほとんどのプラットフォーム (Linux および Windows を含む) において、この関数は OS 依存の CA 証明書ストアから WOLFSSL_CTX へ CA 証明書を読み込もうと試みます。読み込まれた証明書は信頼されます。
int	wolfSSL_CTX_trust_peer_cert (WOLFSSL_CTX * ctx, const char * file, int type) この関数は、TLS/SSL ハンドシェイクを実行する際にピアを検証するために使用する証明書を読み込みます。ハンドシェイク中に送信されるピア証明書は、利用可能な場合は SKID を使用し、署名を使用して比較されます。これら 2 つが一致しない場合は、読み込まれた CA が使用されます。この機能はマクロ WOLFSSL_TRUST_PEER_CERT を定義することで有効になります。適切な使用方法については、examples を参照してください。

	Name
int	wolfSSL_CTX_use_certificate_chain_file (WOLFSSL_CTX * ctx, const char * file) この関数は、証明書チェーンを SSL コンテキスト (WOLFSSL_CTX) に読み込みます。証明書チェーンを含むファイルは file 引数によって提供され、PEM 形式の証明書を含む必要があります。この関数は、MAX_CHAIN_DEPTH(デフォルト = 9、internal.h で定義) 個までの証明書と、サブジェクト証明書を処理します。
int	wolfSSL_CTX_use_RSAPrivateKey_file (WOLFSSL_CTX * ctx, const char * file, int format) この関数は、SSL 接続で使用されるプライベート RSA 鍵を SSL コンテキスト (WOLFSSL_CTX) に読み込みます。この関数は、wolfSSL が OpenSSL 互換レイヤーを有効にしてコンパイルされた場合 (-enable_opensslExtra、#define OPENSSL_EXTRA) にのみ利用可能であり、より一般的に使用される wolfSSL_CTX_use_PrivateKey_file() 関数と同一です。file 引数には、format 引数で指定された形式の RSA プライベート鍵ファイルへのポインタが含まれます。
long	wolfSSL_get_verify_depth (WOLFSSL * ssl) この関数は、有効なセッション (つまり、非 NULL のセッションオブジェクト (ssl) が存在する場合) に対して許可される最大チェーン深度を返します。デフォルトは 9 です。
long	wolfSSL_CTX_get_verify_depth (WOLFSSL_CTX * ctx) この関数は、CTX 構造体を使用して証明書チェーンの深度を取得します。
int	wolfSSL_use_certificate_file (WOLFSSL * ssl, const char * file, int format) この関数は、証明書ファイルを SSL セッション (WOLFSSL 構造体) に読み込みます。証明書ファイルは file 引数によって提供されます。format 引数は、ファイルの形式タイプ (SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM) を指定します。
int	wolfSSL_use_PrivateKey_file (WOLFSSL * ssl, const char * file, int format) この関数は、プライベート鍵ファイルを SSL セッション (WOLFSSL 構造体) に読み込みます。鍵ファイルは file 引数によって提供されます。format 引数は、ファイルの形式タイプ (SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM) を指定します。

	Name
int	wolfSSL_use_certificate_chain_file (WOLFSSL * ssl, const char * file) この関数は、証明書のチェーンを SSL セッション (WOLFSSL 構造体) にロードします。証明書チェーンを含むファイルは file 引数で指定され、PEM 形式の証明書を含んでいる必要があります。この関数は、MAX_CHAIN_DEPTH(デフォルト = 9、internal.h で定義) 個までの証明書と、サブジェクト証明書を処理します。
int	wolfSSL_use_RSAPrivateKey_file (WOLFSSL * ssl, const char * file, int format) この関数は、SSL 接続で使用されるプライベート RSA 鍵を SSL セッション (WOLFSSL 構造体) にロードします。この関数は、wolfSSL が OpenSSL 互換レイヤーを有効にしてコンパイルされている場合 (-enable_opensslExtra、#define OPENSSSL_EXTRA) にのみ利用可能であり、より一般的に使用される wolfSSL_use_PrivateKey_file() 関数と同一です。file 引数は、format で指定された形式の RSA プライベート鍵ファイルへのポインタを含みます。
int	wolfSSL_CTX_der_load_verify_locations (WOLFSSL_CTX * ctx, const char * file, int format) この関数は wolfSSL_CTX_load_verify_locations に似ていますが、DER 形式の CA ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードできます。PEM 形式の CA ファイルをロードするためにも使用できます。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数で指定されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである可能性があります。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイル内で提示された順序でそれらをロードします。format 引数は、証明書の形式が SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1(DER) のいずれかであることを指定します。 wolfSSL_CTX_load_verify_locations とは異なり、この関数は指定されたディレクトリパスからの CA 証明書のロードを許可しません。この関数は、wolfSSL ライブラリが WOLFSSL_DER_LOAD を定義してコンパイルされている場合にのみ利用可能です。
WOLFSSL_CTX *	wolfSSL_CTX_new (WOLFSSL_METHOD *) この関数は、入力として希望する SSL/TLS プロトコルメソッドを取り、新しい SSL コンテキストを作成します。
WOLFSSL *	wolfSSL_new (WOLFSSL_CTX *) この関数は、既に作成された SSL コンテキストを入力として取り、新しい SSL セッションを作成します。

	Name
int	wolfSSL_set_fd (WOLFSSL * ssl, int fd) この関数は、ファイルディスクリプタ (fd) を SSL 接続の入出力機能として割り当てます。通常、これはソケットファイルディスクリプタになります。
int	wolfSSL_set_dtls_fd_connected (WOLFSSL * ssl, int fd) この関数は、ファイルディスクリプタ (fd) を SSL 接続の入出力機能として割り当てます。通常、これはソケットファイルディスクリプタになります。これは、ソケットが接続されていることをマークするため、DTLS 固有の API です。この fd に対する recvfrom および sendto 呼び出しは、addr および addr_len パラメータが NULL に設定されます。
int	wolfDTLS_SetChGoodCb (WOLFSSL * ssl, ClientHelloGoodCb cb, void * user_ctx) 正しく処理および検証された DTLS クライアント hello に対するコールバックを設定できます。クッキー交換メカニズム (DTLS 1.2 の HelloVerifyRequest またはクッキー拡張を伴う DTLS 1.3 の HelloRetryRequest) を使用する場合、このコールバックはクッキー交換が成功した後に呼び出されます。これは、1 つの WOLFSSL オブジェクトを新しい接続のリスナーとして使用し、ClientHello が検証された後 (クッキー交換を通じて、または ClientHello が正しい形式であるかをチェックするだけで) に WOLFSSL オブジェクトを分離できるようにするのに役立ちます。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) 渡された優先度レベルでの暗号の名前を取得します。
int	wolfSSL_get_ciphers (char * buf, int len) この関数は、wolfSSL で有効になっている暗号を取得します。
const char *	wolfSSL_get_cipher_name (WOLFSSL * ssl) この関数は、引数を wolfSSL_get_cipher_name_internal に渡すことによって、DHE-RSA 形式で暗号名を取得します。
int	wolfSSL_get_fd (const WOLFSSL * ssl) この関数は、SSL 接続の入力機能として使用される読み取りファイル記述子 (fd) を返します。通常、これはソケットファイル記述子になります。
int	wolfSSL_get_wfd (const WOLFSSL * ssl) この関数は、SSL 接続の出力機能として使用される書き込みファイルディスクリプタ (fd) を返します。通常はソケットファイルディスクリプタになります。

	Name
void	wolfSSL_set_using_nonblock (WOLFSSL * ssl, int nonblock) この関数は、WOLFSSL オブジェクトに対して、下層の I/O がノンブロッキングであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキングソケットと共に使用する場合は、wolfSSL_set_using_nonblock() を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、recvfrom 呼び出しがブロックすることを意味すると認識できます。
int	wolfSSL_get_using_nonblock (WOLFSSL *) この関数により、アプリケーションは wolfSSL がノンブロッキング I/O を使用しているかどうかを判定できます。wolfSSL がノンブロッキング I/O を使用している場合、この関数は 1 を返し、それ以外の場合は 0 を返します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキングソケットと共に使用する場合は、wolfSSL_set_using_nonblock() を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、recvfrom 呼び出しがブロックすることを意味すると認識できます。
int	wolfSSL_write (WOLFSSL * ssl, const void * data, int sz) この関数は、バッファ data から sz バイトを SSL 接続 ssl に書き込みます。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_write() は SSL/TLS セッションをネゴシエートします。(D)TLSv1.3 を使用していて early data 機能がコンパイルされている場合、この関数はデータ送信が可能になるまでハンドシェイクを進めます。次の wolfSSL_Connect()、wolfSSL_Accept()、wolfSSL_read() の呼び出しでハンドシェイクが完了します。wolfSSL_write() は、ブロッキングとノンブロッキングの両方の I/O で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_write() は、下層の I/O が wolfSSL_write() を継続するために必要な要求を満たせなくなった時点で返されます。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、下層の I/O の準備ができれば、wolfSSL_write() の呼び出しを繰り返す必要があります。下層の I/O がブロッキングの場合、wolfSSL_write() は、サイズ sz のバッファデータが完全に書き込まれるか、エラーが発生するまで返されません。

	Name
int	wolfSSL_read (WOLFSSL * ssl, void * data, int sz) この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファ data に読み込みます。読み取られたバイトは内部受信バッファから削除されます。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_read() は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズが 16kB の SSL レコードを使用します (最大レコードサイズは、wolfssl/internal.h 内の MAX_RECORD_SIZE 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号できるようにする前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_read() の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。内部 wolfSSL 受信バッファには、まだ復号されていない追加データが待機している可能性があります。これは次の wolfSSL_read() 呼び出しで取得および復号されます。sz が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_read() は内部読み取りバッファ内で利用可能なバイトを返します。内部読み取りバッファにまだバイトがバッファリングされていない場合、wolfSSL_read() の呼び出しは次のレコードの処理をトリガします。

	Name
int	wolfSSL_peek(WOLFSSL * ssl, void * data, int sz) この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファ data にコピーします。この関数は、内部 SSL セッション受信バッファ内のデータが削除または変更されない点を除いて、wolfSSL_read() と同じです。wolfSSL_read() と同様に、必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_peek() は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズが 16kB の SSL レコードを使用します (最大レコードサイズは、wolfssl/internal.h 内の MAX_RECORD_SIZE 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号できるようにする前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_peek() の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。内部 wolfSSL 受信バッファには、まだ復号されていない追加データが待機している可能性があります、これは次の wolfSSL_peek() または wolfSSL_read() 呼び出しで取得および復号されます。sz が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_peek() は内部読み取りバッファ内で利用可能なバイトを返します。内部読み取りバッファにまだバイトがバッファリングされていない場合、wolfSSL_peek() の呼び出しは次のレコードの処理をトリガします。
int	wolfSSL_accept(WOLFSSL * ssl) この関数はサーバ側で呼び出され、SSL クライアントが SSL/TLS ハンドシェイクを開始するのを待機します。この関数が呼び出されるとき、下層の通信チャネルはすでに設定されています。wolfSSL_accept() は、ブロッキングとノンブロッキングの両方の I/O で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_accept() は、下層の I/O が wolfSSL_accept がハンドシェイクを継続するために必要な要求を満たせなくなった時点で返されます。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、データの読み取りが可能になったら wolfSSL_accept() の呼び出しを繰り返す必要があり、wolfSSL は中断した箇所から再開します。ノンブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。下層の I/O がブロッキングの場合、wolfSSL_accept() は、ハンドシェイクが完了するか、エラーが発生するまで返されません。

	Name
int	wolfDTLS_accept_stateless (WOLFSSL * ssl) この関数はサーバ側で呼び出され、SSL クライアントが DTLS ハンドシェイクを開始するのをステートレスに待機します。
void	wolfSSL_CTX_free (WOLFSSL_CTX * ctx) この関数は、割り当てられた WOLFSSL_CTX オブジェクトを解放します。この関数は CTX 参照カウントをデクリメントし、参照カウントが 0 になった場合にのみコンテキストを解放します。
void	wolfSSL_free (WOLFSSL * ssl) この関数は、割り当てられた wolfSSL オブジェクトを解放します。
int	wolfSSL_shutdown (WOLFSSL * ssl) この関数は、SSL セッション ssl を使用してアクティブな SSL/TLS 接続をシャットダウンします。この関数は、ピアに「close notify」アラートを送信しようと試みます。呼び出し側のアプリケーションは、ピアからの応答として「close notify」アラートが送信されるのを待つか、wolfSSL_shutdown() を直接呼び出した後に基盤となる接続をシャットダウンするか (リソースを節約するため) を選択できます。TLS 仕様では、どちらのオプションも許可されています。基盤となる接続を将来再び使用する場合は、ピア間の同期を維持するために、完全な双方向シャットダウン手順を実行する必要があります。wolfSSL_shutdown() は、ブロッキング I/O とノンブロッキング I/O の両方で動作します。基盤となる I/O がノンブロッキングの場合、基盤となる I/O が wolfSSL_shutdown() の継続に必要な要求を満たせなかった場合、wolfSSL_shutdown() はエラーを返します。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、基盤となる I/O の準備ができたときに、wolfSSL_shutdown() の呼び出しを繰り返す必要があります。

	Name
int	wolfSSL_send (WOLFSSL * ssl, const void * data, int sz, int flags) この関数は、指定されたフラグを使用して、バッファ data から sz バイトを SSL 接続 ssl に書き込みます。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_send() は SSL/TLS セッションをネゴシエートします。wolfSSL_send() は、ブロッキング I/O とノンブロッキング I/O の両方で動作します。基盤となる I/O がノンブロッキングの場合、基盤となる I/O が wolfSSL_send の継続に必要な要求を満たせなかった場合、wolfSSL_send() は戻ります。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、基盤となる I/O の準備ができたときに、wolfSSL_send() の呼び出しを繰り返す必要があります。基盤となる I/O がブロッキングの場合、wolfSSL_send() は、サイズ sz のバッファ data が完全に書き込まれるか、エラーが発生するまで戻りません。
int	wolfSSL_recv (WOLFSSL * ssl, void * data, int sz, int flags) この関数は、指定されたフラグを使用して、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファ data に読み込みます。読み取られたバイトは、内部受信バッファから削除されます。この関数は、基盤となる読み取り操作の recv フラグをアプリケーションが設定できる点を除いて、wolfSSL_read() と同じです。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_recv() は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズ 16kB の SSL レコードを使用します (最大レコードサイズは、/wolfssl/internal.h の MAX_RECORD_SIZE 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号する前に、内部で SSL レコード全体を読み取る必要があります。このため、wolfSSL_recv() の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。まだ復号されていない追加データが内部 wolfSSL 受信バッファで待機している可能性があり、次の wolfSSL_recv() 呼び出しで取得および復号されます。sz が内部読み取りバッファのバイト数より大きい場合、SSL_recv() は内部読み取りバッファで利用可能なバイトを返します。内部読み取りバッファにまだバッファリングされているバイトがない場合、wolfSSL_recv() の呼び出しは次のレコードの処理をトリガーします。

	Name
int	wolfSSL_get_error (WOLFSSL * ssl, int ret) この関数は、前回の API 関数呼び出し (wolfSSL_connect、wolfSSL_accept、wolfSSL_read、wolfSSL_write 等) がエラーリターンコード (SSL_FAILURE) になった理由を説明する一意のエラーコードを返します。前回の関数の戻り値は、ret を通じて wolfSSL_get_error に渡されます。wolfSSL_get_error が呼び出されて一意のエラーコードを返した後、wolfSSL_ERR_error_string() を呼び出して、人間が読めるエラー文字列を取得できます。詳細については、wolfSSL_ERR_error_string() を参照してください。
int	wolfSSL_get_alert_history (WOLFSSL * ssl, WOLFSSL_ALERT_HISTORY * h) この関数はアラート履歴を取得します。
int	wolfSSL_set_session (WOLFSSL * ssl, WOLFSSL_SESSION * session) この関数は、SSL オブジェクト ssl が SSL/TLS 接続を確立するために使用されるときに使用されるセッションを設定します。セッション再開の場合、セッションオブジェクトで wolfSSL_shutdown() を呼び出す前に、アプリケーションは wolfSSL_get1_session() を呼び出してオブジェクトからセッション ID を保存する必要があります。これはセッションへのポインタを返します。後で、アプリケーションは新しい WOLFSSL オブジェクトを作成し、wolfSSL_set_session() を使用して保存されたセッションを割り当てる必要があります。この時点で、アプリケーションは wolfSSL_connect() を呼び出すことができ、wolfSSL はセッションの再開を試みます。wolfSSL サーバーコードは、デフォルトでセッション再開を許可します。wolfSSL_get1_session() によって返されたオブジェクトは、アプリケーションが使用を終えた後、wolfSSL_SESSION_free() を呼び出して解放する必要があります。

	Name
WOLFSSL_SESSION *	wolfSSL_get_session (WOLFSSL * ssl) NO_SESSION_CACHE_REF が定義されている場合、この関数は ssl で使用されている現在のセッション (WOLFSSL_SESSION) へのポインタを返します。この関数は、WOLFSSL_SESSION オブジェクトへの非永続的なポインタを返します。返されたポインタは、wolfSSL_free が呼び出されたときに解放されます。この呼び出しは、現在のセッションを検査または変更するためにのみ使用する必要があります。セッション再開には、wolfSSL_get1_session() を使用することをお勧めします。後方互換性のため、NO_SESSION_CACHE_REF が定義されていない場合、この関数はローカルキャッシュに格納されている永続的なセッションオブジェクトポインタを返します。キャッシュサイズは有限であり、アプリケーションが wolfSSL_set_session() を呼び出すまでに、別の ssl 接続によってセッションオブジェクトが上書きされるリスクがあります。アプリケーションで NO_SESSION_CACHE_REF を定義し、セッション再開に wolfSSL_get1_session() を使用することをお勧めします。
void	wolfSSL_flush_sessions (WOLFSSL_CTX * ctx, long tm) この関数は、期限切れのセッションをセッションキャッシュからフラッシュします。時刻 tm は、時刻比較に使用されます。wolfSSL は現在セッションに静的テーブルを使用しているため、フラッシュは不要です。そのため、この関数は現在スタブにすぎません。この関数は、wolfSSL が OpenSSL 互換レイヤーでコンパイルされている場合、OpenSSL 互換性 (SSL_flush_sessions) を提供します。
int	wolfSSL_SetServerID (WOLFSSL * ssl, const unsigned char * id, int len, int newSession) この関数は、クライアントセッションをサーバー ID に関連付けます。newSession フラグがオンの場合、既存のセッションは再利用されません。
int	wolfSSL_GetSessionIndex (WOLFSSL * ssl) この関数は、WOLFSSL 構造体のセッションインデックスを取得します。
int	wolfSSL_GetSessionAtIndex (int index, WOLFSSL_SESSION * session) この関数はセッションキャッシュの指定されたインデックスにあるセッションを取得し、メモリにコピーします。WOLFSSL_SESSION 構造体はセッション情報を保持します。
WOLFSSL_X509_CHAIN *	wolfSSL_SESSION_get_peer_chain (WOLFSSL_SESSION * session) WOLFSSL_SESSION 構造体からピア証明書チェーンを返します。

	Name
void	wolfSSL_CTX_set_verify (WOLFSSL_CTX * ctx, int mode, VerifyCallback verify_callback) この関数はリモートピアの検証方法を設定し、また検証コールバックを SSL コンテキストに登録できるようにします。検証コールバックは検証失敗が発生した場合にのみ呼び出されます。検証コールバックが不要な場合は、verify_callback に NULL ポインタを使用できます。ピア証明書の検証モードは論理 OR 演算されたフラグのリストです。可能なフラグ値は以下の通りです。SSL_VERIFY_NONE クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常通り続行されます。サーバーモード：サーバーはクライアントに証明書要求を送信しません。したがって、クライアント検証は有効になりません。SSL_VERIFY_PEER クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトで有効になっているため、このオプションを使用しても効果はありません。サーバーモード：サーバーはクライアントに証明書要求を送信し、受信したクライアント証明書を検証します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：クライアントが要求されたときに証明書の送信に失敗した場合（SSL サーバーで SSL_VERIFY_PEER を使用している場合）、サーバー側で検証が失敗します。SSL_VERIFY_FAIL_EXCEPT_PSK クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じですが、PSK 接続の場合を除きます。PSK 接続が行われている場合、接続はピア証明書なしで続行されます。

	Name
void	wolfSSL_set_verify(WOLFSSL * ssl, int mode, VerifyCallback verify_callback) この関数はリモートピアの検証方法を設定し、また検証コールバックを SSL セッションに登録できるようにします。検証コールバックは検証失敗が発生した場合にのみ呼び出されます。検証コールバックが不要な場合は、verify_callback に NULL ポインタを使用できます。ピア証明書の検証モードは論理 OR 演算されたフラグのリストです。可能なフラグ値は以下の通りです。SSL_VERIFY_NONE クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常通り続行されます。サーバーモード：サーバーはクライアントに証明書要求を送信しません。したがって、クライアント検証は有効になりません。SSL_VERIFY_PEER クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトで有効になっているため、このオプションを使用しても効果はありません。サーバーモード：サーバーはクライアントに証明書要求を送信し、受信したクライアント証明書を検証します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：クライアントが要求されたときに証明書の送信に失敗した場合（SSL サーバーで SSL_VERIFY_PEER を使用している場合）、サーバー側で検証が失敗します。SSL_VERIFY_FAIL_EXCEPT_PSK クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じですが、PSK 接続の場合を除きます。PSK 接続が行われている場合、接続はピア証明書なしで続行されます。
void	wolfSSL_SetCertCbCtx(WOLFSSL * ssl, void * ctx) この関数は検証コールバック用のユーザー CTX オブジェクト情報を格納します。
void	wolfSSL_CTX_SetCertCbCtx(WOLFSSL_CTX * ctx, void * userCtx) この関数は検証コールバック用のユーザー CTX オブジェクト情報を格納します。
int	wolfSSL_pending(WOLFSSL * ssl) この関数は SSL オブジェクト内でバッファリングされ、wolfSSL_read() によって読み取り可能な利用可能なバイト数を返します。
void	wolfSSL_load_error_strings(void) この関数は OpenSSL 互換性 (SSL_load_error_string) のためのものであり、何も動作を行いません。

	Name
int	wolfSSL_library_init (void) この関数は wolfSSL_CTX_new() 内で内部的に呼び出されます。この関数は wolfSSL_Init() のラッパーであり、wolfSSL が OpenSSL 互換性レイヤーでコンパイルされた場合の OpenSSL 互換性 (SSL_library_init) のために存在します。wolfSSL_Init() は、より一般的に使用される wolfSSL 初期化関数です。
int	wolfSSL_SetDevId (WOLFSSL * ssl, int devId) この関数は WOLFSSL セッションレベルで Device Id を設定します。
int	wolfSSL_CTX_SetDevId (WOLFSSL_CTX * ctx, int devId) この関数は WOLFSSL_CTX コンテキストレベルで Device Id を設定します。
int	wolfSSL_CTX_GetDevId (WOLFSSL_CTX * ctx, WOLFSSL * ssl) この関数は Device Id を取得します。
long	wolfSSL_CTX_set_session_cache_mode (WOLFSSL_CTX * ctx, long mode) この関数は SSL セッションキャッシングを有効または無効にします。動作は mode に使用される値に依存します。mode に使用できる値は以下の通りです。 SSL_SESS_CACHE_OFF - セッションキャッシングを無効にします。セッションキャッシングはデフォルトで有効になっています。 SSL_SESS_CACHE_NO_AUTO_CLEAR - セッションキャッシュの自動フラッシュを無効にします。自動フラッシュはデフォルトで有効になっています。
int	wolfSSL_set_session_secret_cb (WOLFSSL * ssl, SessionSecretCb cb, void * ctx) この関数は、セッションシークレットコールバック関数を設定します。SessionSecretCb 型は次のシグネチャを持ちます: int (SessionSecretCb)(WOLFSSL ssl, void* secret, int* secretSz, void* ctx)。WOLFSSL 構造体の sessionSecretCb メンバが、パラメータ cb に設定されます。
int	wolfSSL_save_session_cache (const char * fname) この関数は、セッションキャッシュをファイルに永続化します。追加のメモリ使用量のため、memsave は使用しません。
int	wolfSSL_restore_session_cache (const char * fname) この関数は、永続的なセッションキャッシュをファイルから復元します。追加のメモリ使用量のため、memstore は使用しません。
int	wolfSSL_memsave_session_cache (void * mem, int sz) この関数は、セッションキャッシュをメモリに永続化します。
int	wolfSSL_memrestore_session_cache (const void * mem, int sz) この関数は、永続的なセッションキャッシュをメモリから復元します。

	Name
int	wolfSSL_get_session_cache_memsize (void) この関数は、セッションキャッシュの保存バッファがどのくらい大きくあるべきかを返します。
int	wolfSSL_CTX_save_cert_cache (WOLFSSL_CTX * ctx, const char * fname) この関数は、証明書キャッシュをメモリからファイルに書き込みます。
int	wolfSSL_CTX_restore_cert_cache (WOLFSSL_CTX * ctx, const char * fname) この関数は、証明書キャッシュをファイルから永続化します。
int	wolfSSL_CTX_memsave_cert_cache (WOLFSSL_CTX * ctx, void * mem, int sz, int * used) この関数は、証明書キャッシュをメモリに永続化します。
int	wolfSSL_CTX_memrestore_cert_cache (WOLFSSL_CTX * ctx, const void * mem, int sz) この関数は、証明書キャッシュをメモリから復元します。
int	wolfSSL_CTX_get_cert_cache_memsize (WOLFSSL_CTX * ctx) 証明書キャッシュの保存バッファに必要なサイズを返します。
int	wolfSSL_CTX_set_cipher_list (WOLFSSL_CTX * ctx, const char * list) この関数は、指定された WOLFSSL_CTX に対する暗号スイートリストを設定します。設定したリストは、本コンテキストから新規に作成されるすべての WOLFSSL セッションのデフォルトリストとなり、文字列内の順序がそのまま優先順位（高い順から低い順）として用いられます。本関数を呼び出すたびに前回のリストは置き換えられます。リストは null 終端のテキスト文字列で、暗号スイート名および／または OpenSSL 互換のキーワードをコロンで区切って指定します（例: "TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-SHA256"）。
int	** wolfSSL_set_cipher_list の説明を参照してください。本関数を呼び出すたびに、当該セッションのリストは置き換えられます。
void	wolfSSL_dtls_set_using_nonblock (WOLFSSL * ssl, int nonblock) この関数は、WOLFSSL DTLS オブジェクトに対して、下層の UDP I/O がノンブロッキングであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキング UDP ソケットと共に使用する場合は、 wolfSSL_dtls_set_using_nonblock() を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、recvfrom 呼び出しがブロックすることを意味すると認識できます。

	Name
int	wolfSSL_dtls_get_using_nonblock (WOLFSSL *ssl) この関数により、アプリケーションは wolfSSL が UDP でノンブロッキング I/O を使用しているかどうかを判定できます。wolfSSL がノンブロッキング I/O を使用している場合、この関数は 1 を返し、それ以外の場合は 0 を返します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキング UDP ソケットと共に使用する場合は、wolfSSL_dtls_set_using_nonblock() を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、recvfrom 呼び出しがブロックすることを意味すると認識できます。この関数は DTLS セッションにのみ意味があります。
int	wolfSSL_dtls_get_current_timeout (WOLFSSL *ssl) この関数は、WOLFSSL オブジェクトの現在のタイムアウト値を秒単位で返します。ノンブロッキングソケットを使用する場合、ユーザコード内の何かが、利用可能な受信データをいつチェックするか、およびどのくらい待機しているかを決定する必要があります。この関数が返す値は、アプリケーションがどのくらい待機すべきかを示します。
int	wolfSSL_dtls13_use_quick_timeout (WOLFSSL *ssl) この関数は、アプリケーションがより短いタイムアウトを設定すべき場合に true を返します。ノンブロッキングソケットを使用する場合、ユーザコード内の何かが、利用可能なデータをいつチェックするか、およびどのくらい待機する必要があるかを決定する必要があります。この関数が true を返す場合、ライブラリはすでに通信の中断を検出していますが、他のピアからのメッセージがまだ転送中である可能性に備えて、もう少し待機したいことを意味します。このタイマーの値を微調整するのはアプリケーション次第であり、適切な値は dtls_get_current_timeout() / 4 かもしれません。
void	wolfSSL_dtls13_set_send_more_acks (WOLFSSL *ssl, int value) この関数は、中断を検出したときにライブラリが ACK を他のピアに即座に送信すべきかどうかを設定します。ACK を即座に送信すると最小のレイテンシが保証されますが、必要以上に帯域幅を消費する可能性があります。アプリケーションが自分でタイマーを管理し、このオプションが 0 に設定されている場合、アプリケーションコードは wolfSSL_dtls13_use_quick_timeout() を使用して、これらの遅延 ACK を送信するためにより短いタイムアウトを設定すべきかどうかを判定できます。

	Name
int	wolfSSL_dtls_set_timeout_init (WOLFSSL * ssl, int timeout) この関数は dtls タイムアウトを設定します。
int	wolfSSL_dtls_set_timeout_max (WOLFSSL * ssl, int timeout) この関数は最大 dtls タイムアウトを設定します。
int	wolfSSL_dtls_got_timeout (WOLFSSL * ssl) DTLS でノンブロッキングソケットを使用する場合、制御コードが送信がタイムアウトしたと判断したときに、この関数を WOLFSSL オブジェクトに対して呼び出す必要があります。この関数は、タイムアウト値の調整を含め、最後の送信を再試行するために必要なアクションを実行します。時間が経過しすぎた場合、失敗が返されます。
int	wolfSSL_dtls_retransmit (WOLFSSL * ssl) DTLS でノンブロッキングソケットを使用する場合、この関数は予想されるタイムアウト値と再送信カウントを無視して、最後のハンドシェイクフライトを再送信します。これは、DTLS を使用していて、タイムアウトと再試行回数さえも管理する必要があるアプリケーションに役立ちます。
int	wolfSSL_dtls (WOLFSSL * ssl) この関数は、SSL セッションが DTLS を使用するように設定されているかどうかを判定するために使用されます。
int	wolfSSL_dtls_set_peer (WOLFSSL * ssl, void * peer, unsigned int peerSz) この関数は、DTLS ピア peer(sockaddr_in) をサイズ peerSz で設定します。
int	wolfSSL_dtls_set_pending_peer (WOLFSSL * ssl, void * peer, unsigned int peerSz) この関数は、保留中の DTLS ピア peer(sockaddr_in) をサイズ peerSz で設定します。これにより、次のレコードの保護を正常に解除したときに通常のピアにアップグレードされる保留中のピアが設定されます。これは、ピアのアドレスが変更される可能性があるシナリオで、オフパス攻撃者がピアアドレスを変更するのを防ぐのに役立ちます。これは、新しいピアアドレスへのシームレスで安全な移行を可能にするために、Connection ID と共に使用する必要があります。
int	wolfSSL_dtls_get_peer (WOLFSSL * ssl, void * peer, unsigned int * peerSz) この関数は、現在の DTLS ピアの sockaddr_in(サイズ peerSz) を取得します。この関数は peerSz を、SSL セッションに格納されている実際の DTLS ピアサイズと比較します。ピアが peer に収まる場合、ピアの sockaddr_in が peer にコピーされ、peerSz が peer のサイズに設定されます。

	Name
int	wolfSSL_dtls_get0_peer (WOLFSSL * ssl, const void ** peer, unsigned int * peerSz) この関数は現在の DTLS ピアの sockaddr_in (サイズ peerSz) を取得します。これは wolfSSL_dtls_get_peer() のゼロコピー代替です。
char *	wolfSSL_ERR_error_string (unsigned long errNumber, char * data) この関数は wolfSSL_get_error() によって返されたエラーコードを、より人間が読みやすいエラー文字列に変換します。errNumber は wolfSSL_get_error() によって返されたエラーコードであり、data はエラー文字列が配置される格納バッファです。data の最大長はデフォルトで 80 文字であり、wolfssl/wolfcrypt/error.h の MAX_ERROR_SZ で定義されています。
void	wolfSSL_ERR_error_string_n (unsigned long e, char * buf, unsigned long len) この関数は wolfSSL_ERR_error_string() のバージョンであり、len は buf に書き込まれる最大文字数を指定します。wolfSSL_ERR_error_string() と同様に、この関数は wolfSSL_get_error() から返されたエラーコードをより人間が読みやすいエラー文字列に変換します。人間が読みやすい文字列は buf に配置されます。
int	wolfSSL_get_shutdown (const WOLFSSL * ssl) この関数は Options 構造体の closeNotify、connReset、または sentNotify メンバのシャットダウン条件をチェックします。Options 構造体は WOLFSSL 構造体内にあります。
int	wolfSSL_session_reused (WOLFSSL * ssl) この関数は options 構造体の resuming メンバを返します。このフラグはセッションを再利用するかどうかを示します。再利用しない場合、新しいセッションを確立する必要があります。
int	wolfSSL_is_init_finished (const WOLFSSL * ssl) この関数は接続が確立されているかどうかをチェックします。
const char *	wolfSSL_get_version (WOLFSSL * ssl) 使用されている SSL バージョンを文字列として返します。
int	wolfSSL_get_current_cipher_suite (WOLFSSL * ssl) ssl セッションが使用している現在の暗号スイートを返します。
WOLFSSL_CIPHER *	wolfSSL_get_current_cipher (WOLFSSL * ssl) この関数は ssl セッション内の現在の暗号へのポインタを返します。
const char *	wolfSSL_CIPHER_get_name (const WOLFSSL_CIPHER * cipher) この関数は SSL オブジェクト内の暗号スイートを利用可能なスイートと照合し、文字列表現を返します。
const char *	wolfSSL_get_cipher (WOLFSSL *) この関数は SSL オブジェクト内の暗号スイートを利用可能なスイートと照合します。

	Name
WOLFSSL_SESSION *	wolfSSL_get1_session(WOLFSSL * ssl) この関数は WOLFSSL 構造体から WOLFSSL_SESSION を参照型として返します。これには wolfSSL_SESSION_free を呼び出してセッション参照を解放する必要があります。指される WOLFSSL_SESSION には、セッション再開を実行し、新しいハンドシェイクなしで接続を再確立するために必要なすべての情報が含まれています。セッション再開のために、セッションオブジェクトで wolfSSL_shutdown() を呼び出す前に、アプリケーションは wolfSSL_get1_session() の呼び出しでオブジェクトからセッション ID を保存する必要があります、これはセッションへのポインタを返します。後で、アプリケーションは新しい WOLFSSL オブジェクトを作成し、wolfSSL_set_session() で保存されたセッションを割り当てる必要があります。この時点で、アプリケーションは wolfSSL_connect() を呼び出すことができ、wolfSSL はセッションの再開を試みます。wolfSSL サーバーコードはデフォルトでセッション再開を許可します。wolfSSL_get1_session() によって返されたオブジェクトは、アプリケーションが使用を終えた後、wolfSSL_SESSION_free() を呼び出すことによって解放する必要があります。 **wolfSSLv23_client_method関数は、アプリケーションがクライアントであり、SSL 3.0 から TLS 1.3 までの間でサーバがサポートする最も高いプロトコルバージョンをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい WOLFSSL_METHOD 構造体のためのメモリを割り当て、初期化します。wolfSSL のクライアントとサーバの両方は、堅牢なバージョンダウングレード機能を持っています。どちらか一方で特定のプロトコルバージョンメソッドが使用された場合、そのバージョンのみがネゴシエートされるか、エラーが返されます。例えば、TLSv1 を使用するクライアントが SSLv3 のみのサーバに接続しようとするとう失敗し、同様に TLSv1.1 への接続も失敗します。この問題を解決するために、wolfSSLv23_client_method() 関数を使用するクライアントは、サーバがサポートする最も高いプロトコルバージョンを使用し、必要に応じて SSLv3 までダウングレードします。この場合、クライアントは SSLv3 から TLSv1.3 までを実行しているサーバに接続できます。
WOLFSSL_METHOD *	**wolfSSLv23_client_method関数は、アプリケーションがクライアントであり、SSL 3.0 から TLS 1.3 までの間でサーバがサポートする最も高いプロトコルバージョンをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい WOLFSSL_METHOD 構造体のためのメモリを割り当て、初期化します。wolfSSL のクライアントとサーバの両方は、堅牢なバージョンダウングレード機能を持っています。どちらか一方で特定のプロトコルバージョンメソッドが使用された場合、そのバージョンのみがネゴシエートされるか、エラーが返されます。例えば、TLSv1 を使用するクライアントが SSLv3 のみのサーバに接続しようとするとう失敗し、同様に TLSv1.1 への接続も失敗します。この問題を解決するために、wolfSSLv23_client_method() 関数を使用するクライアントは、サーバがサポートする最も高いプロトコルバージョンを使用し、必要に応じて SSLv3 までダウングレードします。この場合、クライアントは SSLv3 から TLSv1.3 までを実行しているサーバに接続できます。
int	wolfSSL_BIO_get_mem_data(WOLFSSL_BIO * bio, void * p) これは、内部メモリバッファの先頭にバイトポインタを設定するために使用されます。

	Name
long	wolfSSL_BIO_set_fd (WOLFSSL_BIO * b, int fd, int flag) bio が使用するファイルディスクリプタを設定します。
int	wolfSSL_BIO_set_close (WOLFSSL_BIO * b, long flag) クローズフラグを設定します。これは、BIO が解放される際に I/O ストリームをクローズすべきかを示すために使用されます。
WOLFSSL_BIO_METHOD *	wolfSSL_BIO_s_socket (void) これは、BIO_SOCKET タイプの WOLFSSL_BIO_METHOD を取得するために使用されます。
int	wolfSSL_BIO_set_write_buf_size (WOLFSSL_BIO * b, long size) これは、WOLFSSL_BIO の書き込みバッファのサイズを設定するために使用されます。書き込みバッファが以前に設定されていた場合、この関数はサイズをリセットする際にそれを解放します。これは、読み取りと書き込みのインデックスを 0 にリセットする点で wolfSSL_BIO_reset に似ています。
int	wolfSSL_BIO_make_bio_pair (WOLFSSL_BIO * b1, WOLFSSL_BIO * b2) これは、2 つの bio をペアにするために使用されます。ペアになった bio は双方向パイプのように動作し、一方への書き込みはもう一方から読み取ることができ、その逆も同様です。両方の bio が同じスレッドにあることが期待されます。この関数はスレッドセーフではありません。2 つの bio のうちの 1 つを解放すると、両方のペアが解除されます。いずれかの bio に書き込みバッファサイズが以前に設定されていなかった場合、ペアになる前にデフォルトサイズの 17000(WOLFSSL_BIO_SIZE) に設定されます。
int	wolfSSL_BIO_ctrl_reset_read_request (WOLFSSL_BIO * b) これは、読み取り要求フラグを 0 に戻すために使用されます。
int	wolfSSL_BIO_nread0 (WOLFSSL_BIO * bio, char ** buf) これは、読み取り用のバッファポインタを取得するために使用されます。 wolfSSL_BIO_nread とは異なり、内部読み取りインデックスは関数呼び出しから返される数だけ進められません。返される値を超えて読み取ると、配列の境界外を読み取る結果になる可能性があります。
int	wolfSSL_BIO_nread (WOLFSSL_BIO * bio, char ** buf, int num) これは、読み取り用のバッファポインタを取得するために使用されます。内部読み取りインデックスは、関数呼び出しから返される数だけ進められ、buf は読み取るバッファの先頭を指します。読み取りバッファ内のバイト数が num で要求された値より少ない場合、より小さい値が返されます。返される値を超えて読み取ると、配列の境界外を読み取る結果になる可能性があります。

	Name
int	wolfSSL_BIO_nwrite (WOLFSSL_BIO * bio, char ** buf, int num) 関数が返す数だけのバイトを書き込むためのバッファへのポインタを取得します。返される値よりも多くのバイトを返されたポインタに書き込むと、境界外への書き込みになる可能性があります。
int	wolfSSL_BIO_reset (WOLFSSL_BIO * bio) bio を初期状態にリセットします。例えば、BIO_BIO タイプの場合、これは読み取りと書き込みのインデックスをリセットします。
int	wolfSSL_BIO_seek (WOLFSSL_BIO * bio, int ofs) この関数は、ファイルポインタを指定されたオフセットに調整します。これはファイルの先頭からのオフセットです。
int	wolfSSL_BIO_write_filename (WOLFSSL_BIO * bio, char * name) ファイルを設定し、書き込むために使用されます。ファイル内の既存データを上書きし、bio が解放される際にファイルを閉じるよう設定されます。
long	wolfSSL_BIO_set_mem_eof_return (WOLFSSL_BIO * bio, int v) ファイル終端値を設定するために使用されます。一般的な値は-1 で、期待される正の値と混同しないようにします。
long	wolfSSL_BIO_get_mem_ptr (WOLFSSL_BIO * bio, WOLFSSL_BUF_MEM ** m) WOLFSSL_BIO メモリポインタの getter 関数です。
char *	wolfSSL_X509_NAME_online (WOLFSSL_X509_NAME * name, char * in, int sz) この関数は x509 の名前をバッファにコピーします。
WOLFSSL_X509_NAME *	wolfSSL_X509_get_issuer_name (WOLFSSL_X509 * cert) この関数は証明書の発行者名を返します。
WOLFSSL_X509_NAME *	wolfSSL_X509_get_subject_name (WOLFSSL_X509 * cert) この関数は WOLFSSL_X509 構造体の subject メンバーを返します。
int	wolfSSL_X509_get_isCA (WOLFSSL_X509 * x509) WOLFSSL_X509 構造体の isCa メンバーをチェックし、その値を返します。
int	wolfSSL_X509_NAME_get_text_by_NID (WOLFSSL_X509_NAME * name, int nid, char * buf, int len) この関数は渡された NID 値に関連するテキストを取得します。
int	wolfSSL_X509_get_signature_type (WOLFSSL_X509 * x509) この関数は WOLFSSL_X509 構造体の sigOID メンバーに格納されている値を返します。
void	wolfSSL_X509_free (WOLFSSL_X509 * x509) この関数は WOLFSSL_X509 構造体を解放します。
int	wolfSSL_X509_get_signature (WOLFSSL_X509 * x509, unsigned char * buf, int * bufSz) X509 署名を取得し、バッファに格納します。
int	wolfSSL_X509_STORE_add_cert (WOLFSSL_X509_STORE * store, WOLFSSL_X509 * x509) この関数は WOLFSSL_X509_STORE 構造体に証明書を追加します。

	Name
WOLFSSL_STACK *	wolfSSL_X509_STORE_CTX_get_chain (WOLFSSL_X509_STORE_CTX * ctx) この関数は WOLFSSL_X509_STORE_CTX 構造体の chain 変数の getter 関数です。現在、chain は入力されていません。
int	wolfSSL_X509_STORE_set_flags (WOLFSSL_X509_STORE * store, unsigned long flag) この関数は渡された WOLFSSL_X509_STORE 構造体の動作を変更するためのフラグを受け取ります。使用されるフラグの例として WOLFSSL_CRL_CHECK があります。
const byte *	wolfSSL_X509_notBefore (WOLFSSL_X509 * x509) この関数は、バイト配列としてエンコードされた証明書の「not before」有効期限を返します。
const byte *	wolfSSL_X509_notAfter (WOLFSSL_X509 * x509) この関数は、バイト配列としてエンコードされた証明書の「not after」有効期限を返します。
WOLFSSL_BIGNUM *	wolfSSL_ASN1_INTEGER_to_BN (const WOLFSSL_ASN1_INTEGER * ai, WOLFSSL_BIGNUM * bn) この関数は、WOLFSSL_ASN1_INTEGER 値を WOLFSSL_BIGNUM 構造体にコピーするために使用されます。
long	wolfSSL_CTX_add_extra_chain_cert (WOLFSSL_CTX * ctx, WOLFSSL_X509 * x509) この関数は、WOLFSSL_CTX 構造体内で構築中の内部チェーンに証明書を追加します。
int	wolfSSL_CTX_get_read_ahead (WOLFSSL_CTX * ctx) この関数は、WOLFSSL_CTX 構造体からリードアヘッドフラグを取得して返します。
int	wolfSSL_CTX_set_read_ahead (WOLFSSL_CTX * ctx, int v) この関数は、WOLFSSL_CTX 構造体内のリードアヘッドフラグを設定します。
long	wolfSSL_CTX_set_tlsext_status_arg (WOLFSSL_CTX * ctx, void * arg) この関数は、OCSP で使用するオプション引数を設定します。
void	wolfSSL_CTX_set_client_cert_cb (WOLFSSL_CTX * ctx, client_cert_cb cb) クライアント証明書と秘密鍵を選択するコールバックを設定します。
void	wolfSSL_CTX_set_cert_cb (WOLFSSL_CTX * ctx, CertSetupCallback cb, void * arg) 汎用証明書セットアップコールバックを設定します。
int	wolfSSL_CTX_set_tlsext_status_cb (WOLFSSL_CTX * ctx, tlsextStatusCb cb) OCSP ステータス要求 (OCSP ステージング) を処理するために使用されるコールバックを設定します。
int	wolfSSL_CTX_get_tlsext_status_cb (WOLFSSL_CTX * ctx, tlsextStatusCb * cb) コンテキストに現在設定されている OCSP ステータスコールバックを取得します。
long	wolfSSL_get_tlsext_status_ocsp_resp (WOLFSSL * ssl, unsigned char ** resp) ピアに送信 (ステップル) される OCSP レスポンスを取得します。

	Name
long	wolfSSL_set_tlsext_status_ocsp_resp (WOLFSSL * ssl, unsigned char * resp, int len) ピアに送信 (ステープル) される OCSP レスポンスを設定します。
int	wolfSSL_set_tlsext_status_ocsp_resp_multi (WOLFSSL * ssl, unsigned char * resp, int len, word32 idx) TLS マルチ証明書チェーン用の複数の OCSP レスポンスを設定します。
void	wolfSSL_CTX_set_ocsp_status_verify_cb (WOLFSSL_CTX * ctx, ocspVerifyStatusCb cb, void * cbArg) OCSP ステータスレスポンスを検証するコールバックを設定します。
long	wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg (WOLFSSL_CTX * ctx, void * arg) この関数は、PRF コールバックに渡されるオプション引数を設定します。
long	**wolfSSL_CTX_set_options を使用してください。OpenSSL 形式の "SSL_OP_" マクロは対応する "WOLFSSL_OP_" 値の別名であり、どちらの表記でも使用できます。
long	**wolfSSL_set_options の説明を参照してください。親 WOLFSSL_CTX から継承したフラグはそのまま保持されます。本関数で SSL_OP_NO_TLSv1_3 を設定した場合、当該セッションのネゴシエーション可能なバージョンは TLS 1.2 まで引き下げられます。
long	wolfSSL_get_options (const WOLFSSL * s) この関数は、現在のオプションマスクを返します。
long	wolfSSL_set_tlsext_debug_arg (WOLFSSL * s, void * arg) これは、渡されるデバッグ引数を設定するために使用されます。
long	wolfSSL_set_tlsext_status_type (WOLFSSL * s, int type) この関数は、クライアントアプリケーションがサーバに OCSP ステータスレスポンス (OCSP ステープリングとも呼ばれます) を返送するよう要求する際に呼び出されます。現在、サポートされている唯一のタイプは TLSEXT_STATUSTYPE_ocsp です。
long	wolfSSL_get_verify_result (const WOLFSSL * ssl) これは、ピアの証明書の検証を試みた後の結果を取得するために使用されます。
void	wolfSSL_ERR_print_errors_fp (XFILE fp, int err) この関数は、wolfSSL_get_error() によって返されたエラーコードを、より人間が読みやすいエラー文字列に変換し、その文字列を出力ファイル fp に出力します。err は wolfSSL_get_error() によって返されたエラーコードであり、fp はエラー文字列が配置されるファイルです。

	Name
void	wolfSSL_ERR_print_errors_cb (int)(const char str, size_t len, void u) cb, void u) この関数は、提供されたコールバックを使用してエラーレポートを処理します。コールバック関数は、各エラー行に対して実行されます。文字列、長さ、およびユーザデータがコールバックパラメータに渡されます。
void	wolfSSL_CTX_set_psk_client_callback (WOLFSSL_CTX * ctx, wc_psk_client_callback cb) この関数は、WOLFSSL_CTX 構造体の client_psk_cb メンバを設定します。
void	wolfSSL_set_psk_client_callback (WOLFSSL * ssl, wc_psk_client_callback cb) PSK クライアント側コールバックを設定します。
const char *	wolfSSL_get_psk_identity_hint (const WOLFSSL *) この関数は、PSK アイデンティティヒントを返します。
const char *	wolfSSL_get_psk_identity (const WOLFSSL *) この関数は、Arrays 構造体の client_identity メンバへの定数ポインタを返します。
int	wolfSSL_CTX_use_psk_identity_hint (WOLFSSL_CTX * ctx, const char * hint) この関数は、hint 引数を WOLFSSL_CTX 構造体の server_hint メンバに格納します。
int	wolfSSL_use_psk_identity_hint (WOLFSSL * ssl, const char * hint) この関数は、hint 引数を WOLFSSL 構造体内の Arrays 構造体の server_hint メンバに格納します。
void	wolfSSL_CTX_set_psk_server_callback (WOLFSSL_CTX * ctx, wc_psk_server_callback cb) この関数は、WOLFSSL_CTX 構造体にサーバ側の psk コールバックを設定します。
void	wolfSSL_set_psk_server_callback (WOLFSSL * ssl, wc_psk_server_callback cb) WOLFSSL 構造体の options メンバを設定することにより、サーバ側の psk コールバックを設定します。
int	wolfSSL_set_psk_callback_ctx (WOLFSSL * ssl, void * psk_ctx) WOLFSSL 構造体の options メンバに PSK ユーザコンテキストを設定します。
int	wolfSSL_CTX_set_psk_callback_ctx (WOLFSSL_CTX * ctx, void * psk_ctx) WOLFSSL_CTX 構造体に PSK ユーザコンテキストを設定します。
void *	wolfSSL_get_psk_callback_ctx (WOLFSSL * ssl) WOLFSSL 構造体の options メンバから PSK ユーザコンテキストを取得します。
void *	wolfSSL_CTX_get_psk_callback_ctx (WOLFSSL_CTX * ctx) WOLFSSL_CTX 構造体から PSK ユーザコンテキストを取得します。
int	wolfSSL_CTX_allow_anon_cipher (WOLFSSL_CTX * ctx) この関数は、コンパイル時に HAVE_ANON が定義されている場合、CTX 構造体の havAnon メンバを有効にします。

	Name
WOLFSSL_METHOD *	**wolfSSLv23_server_method 関数は、アプリケーションがサーバであり、SSL 3.0 から TLS 1.3 までのプロトコルバージョンで接続するクライアントをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい WOLFSSL_METHOD 構造体のためのメモリを割り当て、初期化します。
int	wolfSSL_state (WOLFSSL * ssl) これは、WOLFSSL 構造体の内部エラー状態を取得するために使用されます。
WOLFSSL_X509 *	wolfSSL_get_peer_certificate (WOLFSSL * ssl) この関数は、ピアの証明書を取得します。
int	wolfSSL_want_read (WOLFSSL * ssl) この関数は、wolfSSL_get_error() を呼び出して SSL_ERROR_WANT_READ が返される場合と同様です。基礎となるエラー状態が SSL_ERROR_WANT_READ の場合、この関数は 1 を返し、それ以外の場合は 0 を返します。
int	wolfSSL_want_write (WOLFSSL * ssl) この関数は、wolfSSL_get_error() を呼び出して SSL_ERROR_WANT_WRITE が返される場合と同様です。基礎となるエラー状態が SSL_ERROR_WANT_WRITE の場合、この関数は 1 を返し、それ以外の場合は 0 を返します。
int	wolfSSL_check_domain_name (WOLFSSL * ssl, const char * dn)wolfSSL はデフォルトでピア証明書の有効な日付範囲と検証済み署名をチェックします。wolfSSL_connect() または wolfSSL_accept() の前にこの関数を呼び出すと、実行するチェックのリストにドメイン名チェックが追加されます。dn は、受信したピア証明書に対してチェックするドメイン名を保持します。
int	**wolfSSL_check_ip_address の前にこの関数を呼ぶと、ピア証明書の SAN ipAddress エントリに対する IP アドレス検証を追加します。
int	wolfSSL_Init (void) 使用するために wolfSSL ライブラリを初期化します。アプリケーションごとに 1 回、ライブラリへの他の呼び出しの前に呼び出す必要があります。
int	wolfSSL_Cleanup (void)wolfSSL ライブラリをこれ以上使用しないように初期化を解除します。呼び出す必要はありませんが、ライブラリが使用したリソースを解放します。
const char *	wolfSSL_lib_version (void) この関数は現在のライブラリバージョンを返します。
word32	wolfSSL_lib_version_hex (void) この関数は現在のライブラリバージョンを 16 進表記で返します。

	Name
int	wolfSSL_negotiate (WOLFSSL * ssl) SSL メソッドの側面に基づいて実際の接続または受け入れを実行します。クライアント側から呼び出された場合は wolfSSL_connect() が実行され、サーバー側から呼び出された場合は wolfSSL_accept() が実行されます。
int	wolfSSL_set_compression (WOLFSSL * ssl) SSL 接続で圧縮を使用する機能をオンにします。両側で圧縮をオンにする必要があります。そうでない場合、圧縮は使用されません。zlib ライブラリが実際のデータ圧縮を実行します。ライブラリにコンパイルするには、configure システムで -with-libz を使用し、HAVE_LIBZ を定義してください。送信前にデータを圧縮すると送受信されるメッセージの実際のサイズは減少しますが、圧縮によって節約されるデータの量は、最も遅いネットワークを除いて、生データで送信するよりも解析に時間がかかることに注意してください。
int	wolfSSL_set_timeout (WOLFSSL * ssl, unsigned int to) この関数は SSL セッションのタイムアウト値を秒単位で設定します。
int	wolfSSL_CTX_set_timeout (WOLFSSL_CTX * ctx, unsigned int to) この関数は、指定された SSL コンテキストの SSL セッションのタイムアウト値を秒単位で設定します。
WOLFSSL_X509_CHAIN *	wolfSSL_get_peer_chain (WOLFSSL * ssl) ピアの証明書チェーンを取得します。
int	wolfSSL_get_chain_count (WOLFSSL_X509_CHAIN * chain) ピアの証明書チェーン数を取得します。
int	wolfSSL_get_chain_length (WOLFSSL_X509_CHAIN * chain, int idx) インデックス (idx) におけるピアの ASN1.DER 証明書の長さをバイト単位で取得します。
unsigned char *	wolfSSL_get_chain_cert (WOLFSSL_X509_CHAIN * chain, int idx) インデックス (idx) におけるピアの ASN1.DER 証明書を取得します。
WOLFSSL_X509 *	wolfSSL_get_chain_X509 (WOLFSSL_X509_CHAIN * chain, int idx) この関数は証明書チェーンからインデックス (idx) にあるピアの wolfSSL_X509_certificate を取得します。
int	wolfSSL_get_chain_cert_pem (WOLFSSL_X509_CHAIN * chain, int idx, unsigned char * buf, int inLen, int * outLen) インデックス (idx) におけるピアの PEM 証明書を取得します。
const unsigned char *	wolfSSL_get_sessionID (const WOLFSSL_SESSION * s) セッション ID を取得します。セッション ID は常に 32 バイト長です。

	Name
int	wolfSSL_X509_get_serial_number (WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) ピアの証明書シリアル番号を取得します。シリアル番号バッファ (in) は少なくとも 32 バイト長であり、入力として <i>inOutSz</i> 引数として提供される必要があります。関数を呼び出した後、 <i>inOutSz</i> には in バッファに書き込まれた実際の長さ (バイト単位) が保持されます。
char *	wolfSSL_X509_get_subjectCN (WOLFSSL_X509 *) 証明書からサブジェクトのコモンネームを返します。
const unsigned char *	wolfSSL_X509_get_der (WOLFSSL_X509 * x509, int * outSz) この関数は、WOLFSSL_X509 構造体内の DER エンコードされた証明書を取得します。
WOLFSSL_ASN1_TIME *	wolfSSL_X509_get_notAfter (WOLFSSL_X509 *) この関数は、x509 が NULL であるかどうかをチェックし、そうでない場合は x509 構造体の <i>notAfter</i> メンバを返します。
int	wolfSSL_X509_version (WOLFSSL_X509 * x509) この関数は、X509 証明書のバージョンを取得します。
WOLFSSL_X509 *	wolfSSL_X509_d2i_fp (WOLFSSL_X509 ** x509, FILE * file) NO_STDIO_FILESYSTEM が定義されている場合、この関数はヒープメモリを割り当て、WOLFSSL_X509 構造体を初期化し、そのポインタを返します。
WOLFSSL_X509 *	wolfSSL_X509_load_certificate_file (const char * fname, int format) この関数は、x509 証明書をメモリにロードします。
unsigned char *	wolfSSL_X509_get_device_type (WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は、x509 構造体からデバイスタイプをバッファにコピーします。
unsigned char *	wolfSSL_X509_get_hw_type (WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は、WOLFSSL_X509 構造体の <i>hwType</i> メンバをバッファにコピーします。
unsigned char *	wolfSSL_X509_get_hw_serial_number (WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は、x509 オブジェクトの <i>hwSerialNum</i> メンバを返します。

	Name
int	wolfSSL_connect_cert(WOLFSSL * ssl) この関数はクライアント側で呼び出され、ピアの証明書チェーンを取得するのに十分な長さだけサーバーとの SSL/TLS ハンドシェイクを開始します。この関数が呼び出されるとき、基礎となる通信チャネルはすでに設定されています。 wolfSSL_connect_cert() は、ブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、wolfSSL_connect_cert() は、基礎となる I/O が wolfSSL_connect_cert() がハンドシェイクを続行するために必要な処理を満たせないときに戻ります。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。呼び出しプロセスは、基礎となる I/O が準備できたときに wolfSSL_connect_cert() の呼び出しを繰り返す必要があり、wolfSSL は中断した場所から再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件をチェックできます。基礎となる I/O がブロッキング I/O の場合、wolfSSL_connect_cert() はピアの証明書チェーンが受信されたときにのみ戻ります。
WC_PKCS12 *	wolfSSL_d2i_PKCS12_bio(WOLFSSL_BIO * bio, WC_PKCS12 ** pkcs12)wolfSSL_d2i_PKCS12_bio(d2i_PKCS12_bio) は、WOLFSSL_BIO から WC_PKCS12 構造体へ PKCS12 情報をコピーします。情報は、構造体内でコンテンツ情報のリストとして分割され、オプションの MAC 情報を保持する構造体も含まれます。情報が WC_PKCS12 構造体内でチャンク (ただし復号されていない) に分割された後、呼び出すことによって解析および復号できます。
WC_PKCS12 *	wolfSSL_i2d_PKCS12_bio(WOLFSSL_BIO * bio, WC_PKCS12 * pkcs12)wolfSSL_i2d_PKCS12_bio(i2d_PKCS12_bio) は、WC_PKCS12 構造体から WOLFSSL_BIO へ証明書情報をコピーします。

	Name
int	wolfSSL_PKCS12_parse(WOLFSSL_X509) ca)PKCS12 は、configure コマンドに <code>-enable_opensslextra</code> を追加することで有効にできます。復号にトリプル DES と RC4 を使用できるため、<code>opensslextra</code> を有効にする際にこれらの機能も有効にすることを推奨します (<code>-enable_des3 -enable_arc4</code>)。wolfSSL は現在 RC2 をサポートしていないため、RC2 での復号は現在利用できません。これは、OpenSSL コマンドラインで <code>p12</code> ファイルを作成する際に使用されるデフォルトの暗号化スキームで顕著になる可能性があります。 <code>wolfSSL_PKCS12_parse(PKCS12_parse)</code>。この関数が最初に行うことは、存在する場合に MAC が正しいかどうかを確認することです。MAC が失敗した場合、関数は返され、保存されているコンテンツ情報の復号を試みません。この関数は、各コンテンツ情報を解析してバッグタイプを探し、バッグタイプが既知の場合、必要に応じて復号され、構築中の証明書リストまたは見つかった鍵として保存されます。すべてのバッグを解析した後、見つかった鍵は証明書リストと比較され、一致するペアが見つかります。この一致するペアは、鍵と証明書として返されます。オプションで、見つかった証明書リストは <code>STACK_OF</code> の証明書として返されます。現時点では、CRL、Secret、または SafeContents バッグはスキップされ、解析されません。これらまたは他の「不明な」バッグがスキップされているかどうかは、デバッグ出力を表示することで確認できます。フレンドリ名などの追加属性は、PKCS12 ファイルを解析する際にスキップされます。
int	wolfSSL_SetTmpDH(WOLFSSL * ssl, const unsigned char * p, int pSz, const unsigned char * g, int gSz) サーバ Diffie-Hellman エフェメラルパラメータの設定。この関数は、サーバが DHE を使用する暗号スイートをネゴシエートする場合に使用されるグループパラメータを設定します。
int	wolfSSL_SetTmpDH_buffer(WOLFSSL * ssl, const unsigned char * b, long sz, int format) この関数は、Diffie-Hellman パラメータのラッパーである <code>wolfSSL_SetTMpDH_buffer_wrapper</code> を呼び出します。
int	wolfSSL_SetTmpDH_file(WOLFSSL * ssl, const char * f, int format) この関数は、<code>wolfSSL_SetTmpDH_file_wrapper</code> を呼び出してサーバ Diffie-Hellman パラメータを設定します。
int	wolfSSL_CTX_SetTmpDH(WOLFSSL_CTX * ctx, const unsigned char * p, int pSz, const unsigned char * g, int gSz) サーバ CTX Diffie-Hellman のパラメータを設定します。

	Name
int	wolfSSL_CTX_SetTmpDH_buffer (WOLFSSL_CTX * ctx, const unsigned char * b, long sz, int format)wolfSSL_SetTmpDH_buffer_wrapper を呼び出すラッパー関数です。
int	wolfSSL_CTX_SetTmpDH_file (WOLFSSL_CTX * ctx, const char * f, int format) この関数は、wolfSSL_SetTmpDH_file_wrapper を呼び出してサーバ Diffie-Hellman パラメータを設定します。
int	wolfSSL_CTX_SetMinDhKey_Sz (WOLFSSL_CTX * ctx, word16 keySz_bits) この関数は、WOLFSSL_CTX 構造体の minDhKeySz メンバにアクセスして、Diffie-Hellman 鍵サイズの最小サイズ (ビット単位) を設定します。
int	wolfSSL_SetMinDhKey_Sz (WOLFSSL * ssl, word16 keySz_bits)WOLFSSL 構造体内の Diffie-Hellman 鍵の最小サイズ (ビット単位) を設定します。
int	wolfSSL_CTX_SetMaxDhKey_Sz (WOLFSSL_CTX * ctx, word16 keySz_bits) この関数は、WOLFSSL_CTX 構造体の maxDhKeySz メンバにアクセスして、Diffie-Hellman 鍵サイズの最大サイズ (ビット単位) を設定します。
int	wolfSSL_SetMaxDhKey_Sz (WOLFSSL * ssl, word16 keySz_bits)WOLFSSL 構造体内の Diffie-Hellman 鍵の最大サイズ (ビット単位) を設定します。
int	wolfSSL_GetDhKey_Sz (WOLFSSL * ssl)options 構造体のメンバである dhKeySz の値 (ビット単位) を返します。この値は、Diffie-Hellman 鍵サイズをバイト単位で表します。
int	wolfSSL_CTX_SetMinRsaKey_Sz (WOLFSSL_CTX * ctx, short keySz)WOLFSSL_CTX 構造体と WOLFSSL_CERT_MANAGER 構造体の両方で最小 RSA 鍵サイズを設定します。
int	wolfSSL_SetMinRsaKey_Sz (WOLFSSL * ssl, short keySz)WOLFSSL 構造体にある RSA の最小許容鍵サイズをビット単位で設定します。
int	wolfSSL_CTX_SetMinEccKey_Sz (WOLFSSL_CTX * ctx, short keySz)WOLF_CTX 構造体と WOLFSSL_CERT_MANAGER 構造体で ECC 鍵の最小サイズをビット単位で設定します。
int	wolfSSL_SetMinEccKey_Sz (WOLFSSL * ssl, short keySz)options 構造体の minEccKeySz メンバの値を設定します。options 構造体は WOLFSSL 構造体のメンバであり、ssl パラメータを通じてアクセスされます。
int	wolfSSL_make_eap_keys (WOLFSSL * ssl, void * key, unsigned int len, const char * label) この関数は、マスターシークレットから鍵材料を導出するために EAP-TLS と EAP-TTLS によって使用されます。

	Name
int	wolfSSL_writev (WOLFSSL * ssl, const struct iovec * iov, int iovcnt) writev のセマンティクスをシミュレートしますが、SSL_write() の動作とフロント追加が小さい場合があるため、実際には一度にブロック単位では行いません。writev を使用するソフトウェアへの移植を容易にします。
int	wolfSSL_CTX_UnloadCAs (WOLFSSL_CTX * ctx) この関数は、CA 署名者リストをアンロードし、署名者テーブル全体を解放します。
int	wolfSSL_CTX_UnloadIntermediateCerts (WOLFSSL_CTX * ctx) この関数は、CA 署名者リストに追加された中間証明書をアンロードし、解放します。
int	wolfSSL_CTX_Unload_trust_peers (WOLFSSL_CTX * ctx) この関数は以前にロードされたすべての信頼されたピア証明書をアンロードするために使用されます。この機能は、マクロ WOLFSSL_TRUST_PEER_CERT を定義することで有効になります。
int	wolfSSL_CTX_trust_peer_buffer (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は、TLS/SSL ハンドシェイクを実行する際にピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されるピア証明書は、利用可能な場合は SKID と署名を使用して比較されます。これら 2 つが一致しない場合は、ロードされた CA が使用されます。ファイルではなくバッファからの入力である点を除いて、wolfSSL_CTX_trust_peer_cert と同じ機能です。この機能は、マクロ WOLFSSL_TRUST_PEER_CERT を定義することで有効になります。適切な使用方法については例を参照してください。
int	wolfSSL_CTX_load_verify_buffer (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は CA 証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。適切な使用方法については例を参照してください。

	Name
int	wolfSSL_CTX_load_verify_buffer_ex (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format, int userChain, word32 flags) この関数は CA 証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。_ex 版は PR 2413 で追加され、userChain と flags の追加引数をサポートします。
int	wolfSSL_CTX_load_verify_chain_buffer_format (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は CA 証明書チェーンバッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。適切な使用方法については例を参照してください。
int	wolfSSL_CTX_use_certificate_buffer (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。
int	wolfSSL_CTX_use_PrivateKey_buffer (WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は秘密鍵バッファを SSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。

	Name
int	wolfSSL_CTX_use_certificate_chain_buffer (WOLFSSL_CTX * ctx, const unsigned char * in, long sz) この関数は証明書チェーンバッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。バッファは PEM 形式で、サブジェクトの証明書から始まり、ルート証明書で終わる必要があります。適切な使用方法については例を参照してください。
int	wolfSSL_use_certificate_buffer (WOLFSSL * ssl, const unsigned char * in, long sz, int format) この関数は、証明書バッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。
int	wolfSSL_use_PrivateKey_buffer (WOLFSSL * ssl, const unsigned char * in, long sz, int format) この関数は、秘密鍵バッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。
int	wolfSSL_use_certificate_chain_buffer (WOLFSSL * ssl, const unsigned char * in, long sz) この関数は、証明書チェーンバッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができる点が異なります。バッファはサイズ sz の in 引数によって提供されます。バッファは PEM フォーマットである必要があります、サブジェクトの証明書から始まり、ルート証明書で終わる必要があります。適切な使用方法については例を参照してください。
int	wolfSSL_UnloadCertsKeys (WOLFSSL * ssl) この関数は、SSL が所有する証明書または鍵をアンロードします。
int	wolfSSL_CTX_set_group_messages (WOLFSSL_CTX * ctx) この関数は、可能な場合にハンドシェイクメッセージのグループ化をオンにします。
int	wolfSSL_set_group_messages (WOLFSSL * ssl) この関数は、可能な場合にハンドシェイクメッセージのグループ化をオンにします。

	Name
void	wolfSSL_SetFuzzerCb (WOLFSSL * ssl, CallbackFuzzer cbf, void * fCtx) この関数はファザーコールバックを設定します。
int	wolfSSL_DTLS_SetCookieSecret (WOLFSSL * ssl, const byte * secret, word32 secretSz) この関数は新しい dtls cookie シークレットを設定します。
WC_RNG *	wolfSSL_GetRNG (WOLFSSL * ssl) この関数は乱数を取得します。
int	wolfSSL_CTX_SetMinVersion (WOLFSSL_CTX * ctx, int version) この関数は、許可される最小ダウングレードバージョンを設定します。ダウングレードを許可する接続 (wolfSSLv23_client_method または wolfSSLv23_server_method) を使用する場合にのみ適用されます。
int	wolfSSL_SetMinVersion (WOLFSSL * ssl, int version) この関数は、許可される最小ダウングレードバージョンを設定します。ダウングレードを許可する接続 (wolfSSLv23_client_method または wolfSSLv23_server_method) を使用する場合にのみ適用されます。
int	wolfSSL_GetObjectSize (void) この関数は WOLFSSL オブジェクトのサイズを返し、ビルドオプションと設定に依存します。wolfSSL をビルドする際に SHOW_SIZES が定義されている場合、この関数は WOLFSSL オブジェクト内の個々のオブジェクト (Suites、Ciphers など) のサイズも stdout に出力します。
int	wolfSSL_GetOutputSize (WOLFSSL * ssl, int inSz) 平文入力のリコード層サイズを返します。これは、アプリケーションが指定された平文入力サイズに対して、トランスポート層を介して送信されるバイト数を知りたい場合に役立ちます。この関数は、SSL/TLS ハンドシェイクが完了した後に呼び出す必要があります。
int	wolfSSL_GetMaxOutputSize (WOLFSSL * ssl) 平文データの最大レコード層サイズを返します。これは、プロトコル標準で指定されている最大 SSL/TLS レコードサイズ、または TLS 最大フラグメント長拡張によって設定された最大 TLS フラグメントサイズのいずれかに対応します。この関数は、アプリケーションが wolfSSL_GetOutputSize() を呼び出して INPUT_SIZE_E エラーを受け取った場合に役立ちます。この関数は、SSL/TLS ハンドシェイクが完了した後に呼び出す必要があります。

	Name
int	wolfSSL_SetVersion (WOLFSSL * ssl, int version) この関数は、version で指定されたバージョンを使用して、指定された SSL セッション (WOLFSSL オブジェクト) の SSL/TLS プロトコルバージョンを設定します。これにより、SSL セッション (ssl) のプロトコル設定が上書きされます。これは元々 SSL コンテキスト (wolfSSL_CTX_new()) のメソッドタイプによって定義および設定されたものです。
void	wolfSSL_CTX_SetMacEncryptCb (WOLFSSL_CTX * ctx, CallbackMacEncrypt cb) 呼び出し元がアトミックユーザレコード処理 Mac/暗号化コールバックを設定できるようにします。コールバックは、成功時には 0、エラー時には <0 を返す必要があります。ssl と ctx ポインタは、ユーザの利便性のために利用可能です。macOut は、mac の結果を格納する出力バッファです。macIn は mac 入力バッファであり、macInSz はバッファのサイズを示します。macContent と macVerify は、wolfSSL_SetTlsHmacInner() に必要であり、そのまま渡す必要があります。encOut は、暗号化の結果を格納する出力バッファです。encIn は暗号化する入力バッファであり、encSz は入力のサイズです。コールバックの例は、wolfssl/test.h の myMacEncryptCb() にあります。
void	wolfSSL_SetMacEncryptCtx (WOLFSSL * ssl, void * ctx) 呼び出し元がアトミックユーザレコード処理 Mac/暗号化コールバックコンテキストを ctx に設定できるようにします。
void *	wolfSSL_GetMacEncryptCtx (WOLFSSL * ssl) 呼び出し元が、wolfSSL_SetMacEncryptCtx() で以前に保存されたアトミックユーザレコード処理 Mac/暗号化コールバックコンテキストを取得できるようにします。
void	wolfSSL_CTX_SetDecryptVerifyCb (WOLFSSL_CTX * ctx, CallbackDecryptVerify cb) 呼び出し元がアトミックユーザレコード処理復号/検証コールバックを設定できるようにします。コールバックは、成功時には 0、エラー時には <0 を返す必要があります。ssl と ctx ポインタは、ユーザの利便性のために利用可能です。decOut は、復号の結果を格納する出力バッファです。decIn は暗号化された入力バッファであり、decInSz はバッファのサイズを示します。content と verify は、wolfSSL_SetTlsHmacInner() に必要であり、そのまま渡す必要があります。padSz は、パディングの合計値を設定する必要がある出力変数です。つまり、mac サイズに加えて、パディングとパディングバイトです。コールバックの例は、wolfssl/test.h の myDecryptVerifyCb() にあります。

	Name
void	wolfSSL_SetDecryptVerifyCtx (WOLFSSL * ssl, void * ctx) 呼び出し元がアトミックユーザレコード処理復号/検証コールバックコンテキストを ctx に設定できるようにします。
void *	wolfSSL_GetDecryptVerifyCtx (WOLFSSL * ssl) 呼び出し元が、wolfSSL_SetDecryptVerifyCtx() で以前に保存されたアトミックユーザレコード処理復号/検証コールバックコンテキストを取得できるようにします。
const unsigned char *	wolfSSL_GetMacSecret (WOLFSSL * ssl, int verify) ハンドシェイクプロセスから Hmac/Mac シークレットの取得を可能にします。verify パラメータは、これがピアメッセージの検証用であるかどうかを指定します。
const unsigned char *	wolfSSL_GetClientWriteKey (WOLFSSL *) ハンドシェイクプロセスからクライアント書き込み鍵の取得を可能にします。
const unsigned char *	wolfSSL_GetClientWriteIV (WOLFSSL *) ハンドシェイクプロセスからクライアント書き込み IV(初期化ベクトル) の取得を可能にします。
const unsigned char *	wolfSSL_GetServerWriteKey (WOLFSSL *) ハンドシェイクプロセスからサーバ書き込み鍵の取得を可能にします。
const unsigned char *	wolfSSL_GetServerWriteIV (WOLFSSL *) ハンドシェイクプロセスからサーバ書き込み IV(初期化ベクトル) の取得を可能にします。
int	wolfSSL_GetKeySize (WOLFSSL * ssl) ハンドシェイクプロセスから鍵サイズの取得を可能にします。
int	wolfSSL_GetIVSize (WOLFSSL * ssl) WOLFSSL 構造体に保持されている specs 構造体の iv_size メンバを返します。
int	wolfSSL_GetSide (WOLFSSL * ssl) この WOLFSSL 接続のサイド (側) の取得を可能にします。
int	wolfSSL_IsTLSv1_1 (WOLFSSL * ssl) 呼び出し元が、ネゴシエートされたプロトコルバージョンが少なくとも TLS バージョン 1.1 以上であるかどうかを判断できるようにします。
int	wolfSSL_GetBulkCipher (WOLFSSL * ssl) 呼び出し元がハンドシェイクからネゴシエートされたバルク暗号アルゴリズムを判定できるようにします。
int	wolfSSL_GetCipherBlockSize (WOLFSSL * ssl) 呼び出し元がハンドシェイクからネゴシエートされた暗号ブロックサイズを判定できるようにします。
int	wolfSSL_GetAeadMacSize (WOLFSSL * ssl) 呼び出し元がハンドシェイクからネゴシエートされた aead mac サイズを判定できるようにします。暗号タイプ WOLFSSL_AEAD_TYPE 用。

	Name
int	wolfSSL_GetHmacSize (WOLFSSL * ssl) 呼び出し元がハンドシェイクからネゴシエートされた (h)mac サイズを判定できるようにします。WOLFSSL_AEAD_TYPE 以外の暗号タイプ用。
int	wolfSSL_GetHmacType (WOLFSSL * ssl) 呼び出し元がハンドシェイクからネゴシエートされた (h)mac タイプを判定できるようにします。WOLFSSL_AEAD_TYPE 以外の暗号タイプ用。
int	wolfSSL_GetCipherType (WOLFSSL * ssl) 呼び出し元がハンドシェイクからネゴシエートされた暗号タイプを判定できるようにします。
int	wolfSSL_SetTlsHmacInner (WOLFSSL * ssl, byte * inner, word32 sz, int content, int verify) 呼び出し元がメッセージの送受信のために Hmac Inner ベクトルを設定できるようにします。結果は inner に書き込まれ、少なくとも wolfSSL_GetHmacSize() バイトである必要があります。メッセージのサイズは sz で指定され、content はメッセージのタイプ、verify はこれがピアメッセージの検証であるかを指定します。WOLFSSL_AEAD_TYPE を除く暗号タイプで有効です。
void	wolfSSL_CTX_SetEccSignCb (WOLFSSL_CTX * ctx, CallbackEccSign cb) 呼び出し元が ECC 署名用の公開鍵コールバックを設定できるようにします。コールバックは成功時に 0、エラー時に 0 未満を返す必要があります。ssl と ctx ポインタはコーナーの便宜のために利用可能です。in は署名する入力バッファで、inSz は入力の長さを示します。out は署名の結果を格納する出力バッファです。outSz は入出力変数で、呼び出し時の出力バッファのサイズを指定し、返す前に署名の実際のサイズをそこに格納する必要があります。keyDer は ASN1 形式の ECC 秘密鍵で、keySz は鍵の長さをバイト単位で表します。コールバックの例は wolfssl/test.h myEccSign() にあります。
void	wolfSSL_SetEccSignCtx (WOLFSSL * ssl, void * ctx) 呼び出し元が公開鍵 ECC 署名コールバックコンテキストを ctx に設定できるようにします。
void *	wolfSSL_GetEccSignCtx (WOLFSSL * ssl) 呼び出し元が以前に wolfSSL_SetEccSignCtx() で格納された公開鍵 ECC 署名コールバックコンテキストを取得できるようにします。
void	wolfSSL_CTX_SetEccSignCtx (WOLFSSL_CTX * ctx, void * userCtx) 呼び出し元が公開鍵 ECC 署名コールバックコンテキストを ctx に設定できるようにします。
void *	wolfSSL_CTX_GetEccSignCtx (WOLFSSL_CTX * ctx) 呼び出し元が以前に wolfSSL_SetEccSignCtx() で格納された公開鍵 ECC 署名コールバックコンテキストを取得できるようにします。

	Name
void	wolfSSL_CTX_SetEccVerifyCb (WOLFSSL_CTX * ctx, CallbackEccVerify cb) 呼び出し元が ECC 検証用の公開鍵コールバックを設定できるようにします。コールバックは成功時に 0、エラー時に 0 未満を返す必要があります。ssl と ctx ポインタはユーザの便宜のために利用可能です。sig は検証する署名で、sigSz は署名の長さを示します。hash はメッセージのダイジェストを含む入力バッファで、hashSz はハッシュの長さをバイト単位で示します。result は検証の結果を格納する出力変数で、成功時は 1、失敗時は 0 です。keyDer は ASN1 形式の ECC 秘密鍵で、keySz は鍵の長さをバイト単位で表します。コールバックの例は wolfssl/test.h myEccVerify() にあります。
void	wolfSSL_SetEccVerifyCtx (WOLFSSL * ssl, void * ctx) 呼び出し元が公開鍵 ECC 検証コールバックコンテキストを ctx に設定できるようにします。
void *	wolfSSL_GetEccVerifyCtx (WOLFSSL * ssl) 呼び出し元が以前に wolfSSL_SetEccVerifyCtx() で格納された公開鍵 ECC 検証コールバックコンテキストを取得できるようにします。
void	wolfSSL_CTX_SetRsaSignCb (WOLFSSL_CTX * ctx, CallbackRsaSign cb) 呼び出し元が RSA 署名用の公開鍵コールバックを設定できるようにします。コールバックは成功時に 0、エラー時に 0 未満を返す必要があります。ssl と ctx ポインタはユーザの便宜のために利用可能です。in は署名する入力バッファで、inSz は入力の長さを示します。out は署名の結果を格納する出力バッファです。outSz は入出力変数で、呼び出し時の出力バッファのサイズを指定し、返す前に署名の実際のサイズをそこに格納する必要があります。keyDer は ASN1 形式の RSA 秘密鍵で、keySz は鍵の長さをバイト単位で表します。コールバックの例は wolfssl/test.h myRsaSign() にあります。
void	wolfSSL_SetRsaSignCtx (WOLFSSL * ssl, void * ctx) 呼び出し元が公開鍵 RSA 署名コールバックコンテキストを ctx に設定できるようにします。
void *	wolfSSL_GetRsaSignCtx (WOLFSSL * ssl) 呼び出し元が以前に wolfSSL_SetRsaSignCtx() で格納された公開鍵 RSA 署名コールバックコンテキストを取得できるようにします。

	Name
void	wolfSSL_CTX_SetRsaVerifyCb (WOLFSSL_CTX * ctx, CallbackRsaVerify cb) 呼び出し元が RSA 検証用の公開鍵コールバックを設定できるようにします。コールバックは成功時に平文のバイト数、エラー時に 0 未満を返す必要があります。ssl と ctx ポインタはユーザの便宜のために利用可能です。sig は検証する署名で、sigSz は署名の長さを示します。out は復号プロセスとパディングの後、検証バッファの先頭に設定する必要があります。keyDer は ASN1 形式の RSA 公開鍵で、keySz は鍵の長さをバイト単位で表します。コールバックの例は wolfssl/test.h myRsaVerify() にあります。
void	wolfSSL_SetRsaVerifyCtx (WOLFSSL * ssl, void * ctx) 呼び出し元が公開鍵 RSA 検証コールバックコンテキストを ctx に設定できるようにします。
void *	wolfSSL_GetRsaVerifyCtx (WOLFSSL * ssl) 呼び出し元が wolfSSL_SetRsaVerifyCtx() で以前に保存された公開鍵 RSA 検証コールバックコンテキストを取得できるようにします。
void	wolfSSL_CTX_SetRsaEncCb (WOLFSSL_CTX * ctx, CallbackRsaEnc cb) 呼び出し元が RSA 公開暗号化のための公開鍵コールバックを設定できるようにします。コールバックは成功の場合は 0 を、エラーの場合は 0 未満を返す必要があります。ssl と ctx ポインタはユーザの利便性のために利用可能です。in は暗号化する入力バッファで、inSz は入力の長さを示します。out は暗号化の結果を格納する出力バッファです。outSz は入出力変数で、呼び出し時に出力バッファのサイズを指定し、戻る前に暗号化の実際のサイズを格納する必要があります。keyDer は ASN1 形式の RSA 公開鍵で、keySz はバイト単位の鍵の長さです。コールバックの例は wolfssl/test.h の myRsaEnc() にあります。
void	wolfSSL_SetRsaEncCtx (WOLFSSL * ssl, void * ctx) 呼び出し元が公開鍵 RSA 公開暗号化コールバックコンテキストを ctx に設定できるようにします。
void *	wolfSSL_GetRsaEncCtx (WOLFSSL * ssl) 呼び出し元が wolfSSL_SetRsaEncCtx() で以前に保存された公開鍵 RSA 公開暗号化コールバックコンテキストを取得できるようにします。

	Name
void	wolfSSL_CTX_SetRsaDecCb (WOLFSSL_CTX * ctx, CallbackRsaDec cb) 呼び出し元が RSA 秘密復号のための公開鍵コールバックを設定できるようにします。コールバックは成功の場合は平文のバイト数を、エラーの場合は 0 未満を返す必要があります。ssl と ctx ポインタはユーザーの利便性のために利用可能です。in は復号する入力バッファで、inSz は入力の長さを示します。out は復号処理とパディングの後、復号バッファの先頭に設定する必要があります。keyDer は ASN1 形式の RSA 秘密鍵で、keySz はバイト単位の鍵の長さです。コールバックの例は wolfssl/test.h の myRsaDec() にあります。
void	wolfSSL_SetRsaDecCtx (WOLFSSL * ssl, void * ctx) 呼び出し元が公開鍵 RSA 秘密復号コールバックコンテキストを ctx に設定できるようにします。
void *	wolfSSL_GetRsaDecCtx (WOLFSSL * ssl) 呼び出し元が wolfSSL_SetRsaDecCtx() で以前に保存された公開鍵 RSA 秘密復号コールバックコンテキストを取得できるようにします。
void	wolfSSL_CTX_SetCACb (WOLFSSL_CTX * ctx, CallbackCACache cb) この関数は、新しい CA 証明書が wolfSSL にロードされたときに呼び出されるコールバックを SSL コンテキスト (WOLFSSL_CTX) に登録します。コールバックには DER エンコードされた証明書を含むバッファが渡されます。
WOLFSSL_CERT_MANAGER *	wolfSSL_CertManagerNew_ex (void * heap) 新しい証明書マネージャーコンテキストを割り当て、初期化します。このコンテキストは SSL の必要性とは独立して使用できます。証明書のロード、証明書の検証、失効ステータスのチェックに使用できます。
WOLFSSL_CERT_MANAGER *	wolfSSL_CertManagerNew (void) 新しい証明書マネージャーコンテキストを割り当て、初期化します。このコンテキストは SSL の必要性とは独立して使用できます。証明書のロード、証明書の検証、失効ステータスのチェックに使用できます。
void	wolfSSL_CertManagerFree (WOLFSSL_CERT_MANAGER * cm) 証明書マネージャーコンテキストに関連するすべてのリソースを解放します。証明書マネージャーを使用する必要がなくなったときにこれを呼び出します。
int	wolfSSL_CertManagerLoadCA (WOLFSSL_CERT_MANAGER * cm, const char * f, const char * d) マネージャーコンテキストへの CA 証明書ロードの場所を指定します。PEM 証明書 CAfile には複数の信頼された CA 証明書を含めることができます。CApath が NULL でない場合、PEM 形式の CA 証明書を含むディレクトリを指定します。

	Name
int	wolfSSL_CertManagerLoadCABuffer (WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format)wolfSSL_CTX_load_verify_buffer を呼び出し、関数に渡された cm の情報を失わないように一時的な cm を使用してその結果を返すことで、CA バッファをロードします。
int	wolfSSL_CertManagerUnloadCAs (WOLFSSL_CERT_MANAGER * cm) この関数は CA 署名者リストをアンロードします。
int	wolfSSL_CertManagerUnloadIntermediateCerts (WOLFSSL_CERT_MANAGER * cm) この関数は CA 署名者リストに追加された中間証明書をアンロードします。
int	wolfSSL_CertManagerUnload_trust_peers (WOLFSSL_CERT_MANAGER * cm) この関数は、Trusted Peer リンクリストを解放し、トラステッドピアリストのロックを解除します。
int	wolfSSL_CertManagerVerify (WOLFSSL_CERT_MANAGER * cm, const char * f, int format)Certificate Manager コンテキストで検証する証明書を指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 を指定できます。
int	wolfSSL_CertManagerVerifyBuffer (WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format)Certificate Manager コンテキストで検証する証明書バッファを指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 を指定できます。
void	wolfSSL_CertManagerSetVerify (WOLFSSL_CERT_MANAGER * cm, VerifyCallback verify_callback) この関数は、Certificate Manager 内に verifyCallback 関数を設定します。存在する場合、ロードされた各証明書に対して呼び出されます。検証エラーがある場合、verify callback を使用してエラーをオーバーライドできます。
int	wolfSSL_CertManagerCheckCRL (WOLFSSL_CERT_MANAGER * cm, const unsigned char * der, int sz) オプションが有効な場合、CRL をチェックし、証明書を CRL リストと比較します。
int	wolfSSL_CertManagerEnableCRL (WOLFSSL_CERT_MANAGER * cm, int options)Certificate Manager で証明書を検証する際に、Certificate Revocation List チェックをオンにします。デフォルトでは、CRL チェックはオフです。options には、リーフ証明書のみ(デフォルト)ではなくチェーン内の各証明書に対して CRL チェックを実行する WOLFSSL_CRL_CHECKALL が含まれます。

	Name
int	wolfSSL_CertManagerDisableCRL (WOLFSSL_CERT_MANAGER * cm) Certificate Manager で証明書を検証する際に、Certificate Revocation List チェックをオフにします。デフォルトでは、CRL チェックはオフです。この関数を使用して、以前に CRL チェックを有効にしていたこの Certificate Manager コンテキストに対して、CRL チェックを一時的または永続的に無効にできます。
int	wolfSSL_CertManagerLoadCRL (WOLFSSL_CERT_MANAGER * cm, const char * path, int type, int monitor) エラーチェックを行い、LoadCRL() に渡して失効チェックのために CRL に証明書をロードします。更新された CRL をロードするには、まず wolfSSL_CertManagerFreeCRL を呼び出してから、新しい CRL をロードします。
int	wolfSSL_CertManagerLoadCRLBuffer (WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int type) この関数は、BufferLoadCRL を呼び出して CRL ファイルをロードします。
int	wolfSSL_CertManagerSetCRL_Cb (WOLFSSL_CERT_MANAGER * cm, CbMissingCRL cb) この関数は、CRL Certificate Manager コールバックを設定します。HAVE_CRL が定義されており、一致する CRL レコードが見つからない場合、cbMissingCRL が呼び出されます (wolfSSL_CertManagerSetCRL_Cb で設定)。これにより、外部から CRL を取得してロードできます。
int	wolfSSL_CertManagerSetCRLUpdate_Cb (WOLFSSL_CERT_MANAGER * cm, CbUpdateCRL cb) この関数は、CRL 更新コールバックを設定します。HAVE_CRL と HAVE_CRL_UPDATE_CB が定義されており、CRL が追加されるときに同じ発行者でより低い CRL 番号を持つエントリが存在する場合、既存のエントリと、それを置き換える新しいエントリの詳細と共に CbUpdateCRL が呼び出されます。
int	wolfSSL_CertManagerGetCRLInfo (WOLFSSL_CERT_MANAGER * cm, CrlInfo * info, const byte * buff, long sz, int type) この関数は、エンコードされた CRL バッファから解析された CRL 情報を含む構造体を生成します。
int	wolfSSL_CertManagerCheckOCSP (WOLFSSL_CERT_MANAGER * cm, const unsigned char * der, int sz) この関数は、WOLFSSL_CERT_MANAGER のメンバである ocsEnabled を有効にして、OCSP チェックオプションが有効になっていることを示します。
int	wolfSSL_CertManagerEnableOCSP (WOLFSSL_CERT_MANAGER * cm, int options) OCSP がオフになっている場合にオンにし、設定オプションでコンパイルされている場合に有効にします。
int	wolfSSL_CertManagerDisableOCSP (WOLFSSL_CERT_MANAGER * cm) OCSP 証明書失効を無効にします。

	Name
int	wolfSSL_CertManagerSetOCSPOverrideURL (WOLFSSL_CERT_MANAGER * cm, const char * url) この関数は、URL を WOLFSSL_CERT_MANAGER 構造体の ocsOverrideURL メンバにコピーします。
int	wolfSSL_CertManagerSetOCSP_Cb (WOLFSSL_CERT_MANAGER * cm, CbOCSPio ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx) この関数は、WOLFSSL_CERT_MANAGER 内の OCSP コールバックを設定します。
int	wolfSSL_CertManagerEnableOCSPStapling (WOLFSSL_CERT_MANAGER * cm) この関数は、OCSP ステープリングがオンになっていない場合にオンにし、オプションを設定します。
int	wolfSSL_EnableCRL (WOLFSSL * ssl, int options)CRL 証明書失効を有効にします。
int	wolfSSL_DisableCRL (WOLFSSL * ssl)CRL 証明書失効を無効にします。
int	wolfSSL_LoadCRL (WOLFSSL * ssl, const char * path, int type, int monitor) 失効チェックのための証明書を読み込むために LoadCRL を最終的に呼び出すラッパー関数です。
int	wolfSSL_SetCRL_Cb (WOLFSSL * ssl, CbMissingCRL cb)WOLFSSL_CERT_MANAGER 構造体内の CRL コールバックを設定します。
int	wolfSSL_EnableOCSP (WOLFSSL * ssl, int options) この関数は、OCSP 証明書検証を有効にします。options の値は、以下のオプションの 1 つ以上を OR 演算することで形成されます。WOLFSSL_OCSP_URL_OVERRIDE - 証明書内の URL の代わりにオーバーライド URL を使用します。オーバーライド URL は wolfSSL_CTX_SetOCSP_OverrideURL() 関数を使用して指定されます。WOLFSSL_OCSP_CHECKALL - すべての OCSP チェックをオンに設定します。WOLFSSL_OCSP_NO_NONCE - OCSP 要求作成時の nonce オプションを設定します。
int	wolfSSL_DisableOCSP (WOLFSSL * ssl)OCSP 証明書失効オプションを無効にします。
int	wolfSSL_SetOCSP_OverrideURL (WOLFSSL * ssl, const char * url) この関数は、WOLFSSL_CERT_MANAGER 構造体の ocsOverrideURL メンバを設定します。
int	wolfSSL_SetOCSP_Cb (WOLFSSL * ssl, CbOCSPio ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx) この関数は、WOLFSSL_CERT_MANAGER 構造体に OCSP コールバックを設定します。
int	wolfSSL_CTX_EnableCRL (WOLFSSL_CTX * ctx, int options)CTX を通じて CRL 証明書検証を有効にします。

	Name
int	wolfSSL_CTX_DisableCRL (WOLFSSL_CTX * ctx) この関数は、CTX 構造体で CRL 検証を無効にします。
int	wolfSSL_CTX_LoadCRL (WOLFSSL_CTX * ctx, const char * path, int type, int monitor) この関数は、wolfSSL_CertManagerLoadCRL() を通じて CRL を WOLFSSL_CTX 構造体にロードします。
int	wolfSSL_CTX_SetCRL_Cb (WOLFSSL_CTX * ctx, CbMissingCRL cb) この関数は、wolfSSL_CertManagerSetCRL_Cb を呼び出すことにより、コールバック引数を WOLFSSL_CERT_MANAGER 構造体の cbMissingCRL メンバに設定します。
int	wolfSSL_CTX_EnableOCSP (WOLFSSL_CTX * ctx, int options) この関数は、wolfSSL の OCSP 機能の動作を設定するオプションを設定します。options の値は、以下のオプションの 1 つ以上を OR 演算することで形成されます。WOLFSSL_OCSP_URL_OVERRIDE - 証明書内の URL の代わりにオーバーライド URL を使用します。オーバーライド URL は wolfSSL_CTX_SetOCSP_OverrideURL() 関数を使用して指定されます。WOLFSSL_OCSP_CHECKALL - すべての OCSP チェックをオンに設定します。WOLFSSL_OCSP_NO_NONCE - OCSP 要求作成時の nonce オプションを設定します。
int	wolfSSL_CTX_DisableOCSP (WOLFSSL_CTX * ctx) この関数は、WOLFSSL_CERT_MANAGER 構造体の ocsEnabled メンバに影響を与えることにより、OCSP 証明書失効チェックを無効にします。
int	wolfSSL_CTX_SetOCSP_OverrideURL (WOLFSSL_CTX * ctx, const char * url) この関数は、OCSP が使用する URL を手動で設定します。デフォルトでは、wolfSSL_CTX_EnableOCSP を使用して WOLFSSL_OCSP_URL_OVERRIDE オプションが設定されていない限り、OCSP は個々の証明書で見つかった URL を使用します。
int	wolfSSL_CTX_SetOCSP_Cb (WOLFSSL_CTX * ctx, CbOCSP_IO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx) WOLFSSL_CTX 構造体に OCSP のコールバックを設定します。
int	wolfSSL_CTX_EnableOCSPStapling (WOLFSSL_CTX * ctx) この関数は、wolfSSL_CertManagerEnableOCSPStapling() を呼び出して OCSP ステープリングを有効にします。

	Name
void	wolfSSL_KeepArrays (WOLFSSL * ssl) 通常、SSL ハンドシェイクの終了時に、wolfSSL は一時配列を解放します。ハンドシェイクが始まる前にこの関数を呼び出すと、wolfSSL は一時配列を解放しなくなります。一時配列は wolfSSL_get_keys() や PSK ヒントなどに必要になる場合があります。ユーザーが一時配列を使い終わったら、wolfSSL_FreeArrays() を呼び出してリソースを即座に解放するか、あるいは関連する SSL オブジェクトが解放されるときにリソースが解放されます。
void	wolfSSL_FreeArrays (WOLFSSL * ssl) 通常、SSL ハンドシェイクの終了時に、wolfSSL は一時配列を解放します。ハンドシェイクの前に wolfSSL_KeepArrays() が呼び出されていた場合、wolfSSL は一時配列を解放しません。この関数は一時配列を明示的に解放し、ユーザーが一時配列を使い終わり、SSL オブジェクトが解放されるのを待たずにこれらのリソースを解放したい場合に呼び出す必要があります。
int	wolfSSL_UseSNI (WOLFSSL * ssl, unsigned char type, const void * data, unsigned short size) この関数は、'ssl' パラメータで渡された SSL オブジェクトで Server Name Indication の使用を有効にします。つまり、wolfSSL クライアントは ClientHello で SNI 拡張を送信し、wolfSSL サーバーは SNI 不一致の場合、ClientHello + SNI に対して ServerHello + 空の SNI または fatal アラートで応答します。
int	wolfSSL_CTX_UseSNI (WOLFSSL_CTX * ctx, unsigned char type, const void * data, unsigned short size) この関数は、'ctx' パラメータで渡された SSL コンテキストから作成された SSL オブジェクトに対して Server Name Indication の使用を有効にします。つまり、wolfSSL クライアントは ClientHello で SNI 拡張を送信し、wolfSSL サーバーは SNI 不一致の場合、ClientHello + SNI に対して ServerHello + 空の SNI または fatal アラートで応答します。
void	wolfSSL_SNI_SetOptions (WOLFSSL * ssl, unsigned char type, unsigned char options) この関数は、'ssl' パラメータで渡された SSL オブジェクトで Server Name Indication を使用する SSL セッションの動作を設定するために、サーバー側で呼び出されます。オプションについては以下で説明します。

	Name
void	wolfSSL_CTX_SNI_SetOptions (WOLFSSL_CTX * ctx, unsigned char type, unsigned char options) この関数は、'ctx' パラメータで渡された SSL コンテキストから作成された SSL オブジェクトに対して Server Name Indication を使用する SSL セッションの動作を設定するために、サーバー側で呼び出されます。オプションについては以下で説明します。
int	wolfSSL_SNI_GetFromBuffer (const unsigned char * clientHello, unsigned int helloSz, unsigned char type, unsigned char * sni, unsigned int * inOutSz) この関数は、クライアントがセッションを開始するために送信した Client Hello メッセージからクライアントによって提供された Server Name Indication を取得するために、サーバー側で呼び出されます。SNI を取得するためにコンテキストやセッションのセットアップは必要ありません。
unsigned char	wolfSSL_SNI_Status (WOLFSSL * ssl, unsigned char type) この関数は SNI オブジェクトのステータスを取得します。
unsigned short	wolfSSL_SNI_GetRequest (WOLFSSL * ssl, unsigned char type, void ** data) この関数は、SSL セッションでクライアントによって提供された Server Name Indication を取得するために、サーバー側で呼び出されます。
int	wolfSSL_UseALPN (WOLFSSL * ssl, char * protocol_name_list, unsigned int protocol_name_listSz, unsigned char options) wolfSSL セッションで ALPN の使用をセットアップします。
int	wolfSSL_ALPN_GetProtocol (WOLFSSL * ssl, char ** protocol_name, unsigned short * size) この関数は、サーバーによって設定されたプロトコル名を取得します。
int	wolfSSL_ALPN_GetPeerProtocol (WOLFSSL * ssl, char ** list, unsigned short * listSz) この関数は、SSL オブジェクトから alpn_client_list データをバッファにコピーします。
int	wolfSSL_UseMaxFragment (WOLFSSL * ssl, unsigned char mfl) この関数は、'ssl' パラメータで渡された SSL オブジェクトで Maximum Fragment Length の使用を有効にするために、クライアント側で呼び出されます。これは、wolfSSL クライアントによって ClientHello で Maximum Fragment Length 拡張が送信されることを意味します。

	Name
int	wolfSSL_CTX_UseMaxFragment (WOLFSSL_CTX * ctx, unsigned char mfl) この関数は、'ctx' パラメータで渡された SSL コンテキストから作成された SSL オブジェクトに対して Maximum Fragment Length の使用を有効にするために、クライアント側で呼び出されます。これは、wolfSSL クライアントによって ClientHello で Maximum Fragment Length 拡張が送信されることを意味します。
int	wolfSSL_UseTruncatedHMAC (WOLFSSL * ssl) この関数は、'ssl' パラメータで渡された SSL オブジェクトで Truncated HMAC の使用を有効にするために、クライアント側で呼び出されます。これは、wolfSSL クライアントによって ClientHello で Truncated HMAC 拡張が送信されることを意味します。
int	wolfSSL_CTX_UseTruncatedHMAC (WOLFSSL_CTX * ctx) この関数は、'ctx' パラメータで渡された SSL コンテキストから作成された SSL オブジェクトに対して Truncated HMAC の使用を有効にするために、クライアント側で呼び出されます。これは、wolfSSL クライアントによって ClientHello で Truncated HMAC 拡張が送信されることを意味します。
int	wolfSSL_UseOCSPStapling (WOLFSSL * ssl, unsigned char status_type, unsigned char options) ステープリングは CA に連絡する必要性を排除します。ステープリングは、OCSP で提示される証明書失効チェックのコストを削減します。
int	wolfSSL_CTX_UseOCSPStapling (WOLFSSL_CTX * ctx, unsigned char status_type, unsigned char options) この関数は、ハンドシェイク中に証明書ステータスを要求します。
int	wolfSSL_UseOCSPStaplingV2 (WOLFSSL * ssl, unsigned char status_type, unsigned char options) この関数は、OCSP のステータスタイプとオプションを設定します。
int	wolfSSL_CTX_UseOCSPStaplingV2 (WOLFSSL_CTX * ctx, unsigned char status_type, unsigned char options) OCSP ステープリング用の証明書ステータス要求を作成して初期化します。
int	wolfSSL_UseSupportedCurve (WOLFSSL * ssl, word16 name) この関数は、'ssl' パラメータで渡された SSL オブジェクトで Supported Elliptic Curves 拡張の使用を有効にするために、クライアント側で呼び出されます。これは、wolfSSL クライアントによって ClientHello で有効化されたサポートされる曲線が送信されることを意味します。この関数は、複数の曲線を有効にするために複数回呼び出すことができます。

	Name
int	wolfSSL_CTX_UseSupportedCurve (WOLFSSL_CTX * ctx, word16 name) この関数は、ctx パラメータで渡された SSL コンテキストから作成された SSL オブジェクトに対して、サポートされる楕円曲線拡張の使用を有効にするためにクライアント側で呼び出されます。これは、有効にされたサポートされる曲線が wolfSSL クライアントによって ClientHello で送信されることを意味します。この関数は、複数の曲線を有効にするために複数回呼び出すことができます。
int	wolfSSL_UseSecureRenegotiation (WOLFSSL * ssl) この関数は、提供された WOLFSSL 構造体に対して安全な再ネゴシエーションを強制します。これは推奨されません。
int	wolfSSL_Rehandshake (WOLFSSL * ssl) この関数は安全な再ネゴシエーションハンドシェイクを実行します。wolfSSL はこの機能を推奨していないため、これはユーザが強制するものです。
int	wolfSSL_UseSessionTicket (WOLFSSL * ssl) 提供された WOLFSSL 構造体にセッションチケットを使用するよう強制します。定数 HAVE_SESSION_TICKET が定義されており、定数 NO_WOLFSSL_CLIENT が定義されていない必要があります。
int	wolfSSL_CTX_UseSessionTicket (WOLFSSL_CTX * ctx) この関数は、wolfSSL コンテキストにセッションチケットを使用するよう設定します。
int	wolfSSL_get_SessionTicket (WOLFSSL * ssl, unsigned char * buf, word32 * bufSz) この関数は、Session 構造体の ticket メンバをバッファにコピーします。buf が NULL で bufSz が非 NULL の場合、bufSz はチケット長に設定されます。
int	wolfSSL_set_SessionTicket (WOLFSSL * ssl, const unsigned char * buf, word32 bufSz) この関数は、WOLFSSL 構造体内の WOLFSSL_SESSION 構造体の ticket メンバを設定します。関数に渡されたバッファはメモリにコピーされます。
int	wolfSSL_set_SessionTicket_cb (WOLFSSL * ssl, CallbackSessionTicket cb, void * ctx) この関数は、セッションチケットコールバックを設定します。CallbackSessionTicket 型は、次のシグネチャを持つ関数ポインタです: int (CallbackSessionTicket)(WOLFSSL, const unsigned char, int, void)
int	wolfSSL_send_SessionTicket (WOLFSSL * ssl) この関数は、TLS v1.3 ハンドシェイクが確立された後、クライアントにセッションチケットを送信します。

	Name
int	wolfSSL_CTX_set_TicketEncCb (WOLFSSL_CTX * ctx, SessionTicketEncCb cb) この関数は、RFC 5077 で規定されているセッションチケットをサポートするサーバのセッションチケット鍵暗号化コールバック関数を設定します。
int	wolfSSL_CTX_set_TicketHint (WOLFSSL_CTX * ctx, int hint) この関数は、クライアントに中継されるセッションチケットヒントを設定します。サーバ側での使用。
int	wolfSSL_CTX_set_TicketEncCtx (WOLFSSL_CTX * ctx, void * userCtx) この関数は、コールバック用のセッションチケット暗号化ユーザコンテキストを設定します。サーバ側で使います。
void *	wolfSSL_CTX_get_TicketEncCtx (WOLFSSL_CTX * ctx) この関数は、コールバック用のセッションチケット暗号化ユーザコンテキストを取得します。サーバ側で使います。
int	wolfSSL_SetHsDoneCb (WOLFSSL * ssl, HandShakeDoneCb cb, void * user_ctx) この関数は、ハンドシェイク完了コールバックを設定します。WOLFSSL 構造体の hsDoneCb と hsDoneCtx メンバがこの関数で設定されます。
int	wolfSSL_PrintSessionStats (void) この関数は、セッションの統計情報を出力します。
int	wolfSSL_get_session_stats (unsigned int * active, unsigned int * total, unsigned int * peak, unsigned int * maxSessions) この関数は、セッションの統計情報を取得します。
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) この関数は、cr と sr の値をコピーし、wc_PRF(疑似乱数関数) に渡し、その値を返します。
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) TLS 鍵を導出するための外部向けラッパー。

	Name
int	wolfSSL_connect_ex (WOLFSSL * ssl, HandShakeCallBack hsCb, TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout) wolfSSL_connect_ex() は、HandShake コールバックを設定できる拡張です。これは、デバッガが利用できず、スニффイングが実用的でない場合に、組み込みシステムのデバッグサポートに役立ちます。HandShake コールバックは、ハンドシェイクエラーが発生したかどうかに関わらず呼び出されます。SSL パケットの最大数が既知であるため、動的メモリは使用されません。パケット名は packetNames[] を通じてアクセスできます。接続拡張はまた、タイムアウト値と共に Timeout コールバックを設定できます。これは、ユーザが TCP スタックのタイムアウトを待ちたくない場合に便利です。この拡張は、いずれか、両方、またはどちらのコールバックもなしで呼び出すことができます。
int	wolfSSL_accept_ex (WOLFSSL * ssl, HandShakeCallBack hsCb, TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout) wolfSSL_accept_ex() は、HandShake コールバックを設定できる拡張です。これは、デバッガが利用できず、スニффイングが実用的でない場合に、組み込みシステムのデバッグサポートに役立ちます。HandShake コールバックは、ハンドシェイクエラーが発生したかどうかに関わらず呼び出されます。SSL パケットの最大数が既知であるため、動的メモリは使用されません。パケット名は packetNames[] を通じてアクセスできます。接続拡張はまた、タイムアウト値と共に Timeout コールバックを設定できます。これは、ユーザが TCP スタックのタイムアウトを待ちたくない場合に便利です。この拡張は、いずれか、両方、またはどちらのコールバックもなしで呼び出すことができます。
long	wolfSSL_BIO_set_fp (WOLFSSL_BIO * bio, XFILE fp, int c) これは、BIO の内部ファイルポインタを設定するために使用されます。
long	wolfSSL_BIO_get_fp (WOLFSSL_BIO * bio, XFILE * fp) これは、BIO の内部ファイルポインタを取得するために使用されます。
int	wolfSSL_check_private_key (const WOLFSSL * ssl) この関数は、秘密鍵が使用されている証明書と一致していることを確認します。
int	wolfSSL_X509_get_ext_by_NID (const WOLFSSL_X509 * x, int nid, int lastpos) この関数は、渡された NID 値に一致する拡張インデックスを検索して返します。

	Name
void *	wolfSSL_X509_get_ext_d2i (const WOLFSSL_X509 * x509, int nid, int * c, int * idx) この関数は渡された NID 値に一致する拡張を検索して返します。
int	wolfSSL_X509_digest (const WOLFSSL_X509 * x509, const WOLFSSL_EVP_MD * digest, unsigned char * buf, unsigned int * len) この関数は DER 証明書のハッシュを返します。
int	wolfSSL_use_certificate (WOLFSSL * ssl, WOLFSSL_X509 * x509) ハンドシェイク中に使用する WOLFSSL 構造体の証明書を設定するために使用されます。
int	wolfSSL_use_certificate_ASN1 (WOLFSSL * ssl, const unsigned char * der, int derSz) ハンドシェイク中に使用する WOLFSSL 構造体の証明書を設定するために使用されます。DER 形式のバッファが必要です。
int	wolfSSL_use_PrivateKey (WOLFSSL * ssl, WOLFSSL_EVP_PKEY * pkey) WOLFSSL 構造体の秘密鍵を設定するために使用されます。
int	wolfSSL_use_PrivateKey_ASN1 (int pri, WOLFSSL * ssl, const unsigned char * der, long derSz) WOLFSSL 構造体の秘密鍵を設定するために使用されます。DER 形式の鍵バッファが必要です。
int	wolfSSL_use_RSAPrivateKey_ASN1 (WOLFSSL * ssl, unsigned char * der, long derSz) WOLFSSL 構造体の秘密鍵を設定するために使用されます。DER 形式の RSA 鍵バッファが必要です。
WOLFSSL_DH *	wolfSSL_DSA_dup_DH (const WOLFSSL_DSA * r) この関数は dsa のパラメータを新しく作成された WOLFSSL_DH 構造体に複製します。
int	wolfSSL_SESSION_get_master_key (const WOLFSSL_SESSION * ses, unsigned char * out, int outSz) ハンドシェイク完了後にマスターキーを取得するために使用されます。
int	wolfSSL_SESSION_get_master_key_length (const WOLFSSL_SESSION * ses) マスターシークレットキーの長さを取得するために使用されます。
void	wolfSSL_CTX_set_cert_store (WOLFSSL_CTX * ctx, WOLFSSL_X509_STORE * str) ctx の WOLFSSL_X509_STORE 構造体の setter 関数です。
WOLFSSL_X509 *	wolfSSL_d2i_X509_bio (WOLFSSL_BIO * bio, WOLFSSL_X509 ** x509) この関数は bio から DER バッファを取得し、それを WOLFSSL_X509 構造体に変換します。
WOLFSSL_X509_STORE *	wolfSSL_CTX_get_cert_store (WOLFSSL_CTX * ctx) ctx の WOLFSSL_X509_STORE 構造体の getter 関数です。

	Name
size_t	wolfSSL_BIO_ctrl_pending (WOLFSSL_BIO * b) 読み取り保留中のバイト数を取得します。BIO タイプが BIO_BIO の場合、ペアから読み取るバイト数です。BIO が SSL オブジェクトを含む場合、SSL オブジェクトからの保留中のデータです (wolfSSL_pending(ssl))。BIO_MEMORY タイプの場合、メモリバッファのサイズを返します。
size_t	wolfSSL_get_server_random (const WOLFSSL * ssl, unsigned char * out, size_t outlen) ハンドシェイク中にサーバーから送信されたランダムデータを取得するために使用されます。
size_t	wolfSSL_get_client_random (const WOLFSSL * ssl, unsigned char * out, size_t outSz) ハンドシェイク中にクライアントから送信されたランダムデータを取得するために使用されます。
wc_pem_password_cb *	wolfSSL_CTX_get_default_passwd_cb (WOLFSSL_CTX * ctx) ctx 内に設定されたパスワードコールバックの getter 関数です。
void *	wolfSSL_CTX_get_default_passwd_cb_userdata (WOLFSSL_CTX * ctx) ctx 内に設定されたパスワードコールバックユーザーデータの getter 関数です。
WOLFSSL_X509 *	wolfSSL_PEM_read_bio_X509_AUX (WOLFSSL_BIO * bp, WOLFSSL_X509 ** x, wc_pem_password_cb * cb, void * u) この関数は wolfSSL_PEM_read_bio_X509 と同じように動作します。AUX は、信頼された/拒否されたユースケースや人間が読みやすいフレンドリ名などの追加情報を含むことを意味します。
long	wolfSSL_CTX_set_tmp_dh (WOLFSSL_CTX * ctx, WOLFSSL_DH * dh) WOLFSSL_CTX 構造体の dh メンバを Diffie-Hellman パラメータで初期化します。
WOLFSSL_DSA *	wolfSSL_PEM_read_bio_DSAParams (WOLFSSL_BIO * bp, WOLFSSL_DSA ** x, wc_pem_password_cb * cb, void * u) この関数は、bio 内の PEM バッファから DSA パラメータを取得します。
unsigned long	wolfSSL_ERR_peek_last_error (void) この関数は、WOLFSSL_ERROR から発生した最後のエラーの絶対値を返します。
long	WOLF_STACK_OF (WOLFSSL_X509) const この関数は、ピアの証明書チェーンを取得します。
long	wolfSSL_CTX_clear_options (WOLFSSL_CTX * ctx, long opt) この関数は、WOLFSSL_CTX オブジェクトのオプションビットをリセットします。
int	wolfSSL_set_jobject (WOLFSSL * ssl, void * objPtr) この関数は、WOLFSSL 構造体の jobjectRef メンバを設定します。
void *	wolfSSL_get_jobject (WOLFSSL * ssl) この関数は、WOLFSSL 構造体の jobjectRef メンバを返します。

	Name
int	wolfSSL_set_msg_callback (WOLFSSL * ssl, SSL_Msg_Cb cb) この関数は、ssl 内にコールバックを設定します。コールバックはハンドシェイクメッセージを監視するためのものです。cb の NULL 値はコールバックをリセットします。
int	wolfSSL_set_msg_callback_arg (WOLFSSL * ssl, void * arg) この関数は、ssl 内に関連するコールバックコンテキスト値を設定します。値はコールバック引数に渡されます。
char *	wolfSSL_X509_get_next_altname (WOLFSSL_X509 *) この関数は、ピア証明書から次の代替名 (もしあれば) を返します。
WOLFSSL_ASN1_TIME *	wolfSSL_X509_get_notBefore (WOLFSSL_X509 *) この関数は、x509 が NULL かどうかを確認し、NULL でない場合は x509 構造体の notBefore メンバを返します。
int	wolfSSL_connect (WOLFSSL * ssl) この関数はクライアント側で呼び出され、サーバとの SSL/TLS ハンドシェイクを開始します。この関数が呼び出されると、基礎となる通信チャネルはすでに設定されています。wolfSSL_connect() は、ブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、wolfSSL_connect() は、基礎となる I/O が wolfSSL_connect() がハンドシェイクを続行するために必要とするものを満たすことができないときに返されます。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE を返します。呼び出しプロセスは、基礎となる I/O の準備ができたときに wolfSSL_connect() の呼び出しを繰り返す必要があります。wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキングの場合、wolfSSL_connect() はハンドシェイクが完了するかエラーが発生するまで返されません。wolfSSL は、OpenSSL とは異なるアプローチで証明書検証を行います。クライアントのデフォルトポリシーはサーバを検証することです。つまり、サーバを検証するための CA を読み込まない場合、接続エラー、検証不可 (155) が発生します。サーバの検証が失敗しても SSL_connect が成功し、セキュリティを低下させる OpenSSL の動作を模倣したい場合は、SSL_new() を呼び出す前に SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); を呼び出すことでこれを実現できます。ただし、推奨されません。

	Name
int	wolfSSL_send_hrr_cookie (WOLFSSL * ssl, const unsigned char * secret, unsigned int secretSz) この関数は、HelloRetryRequest メッセージに Cookie を含める必要があることを示すために、サーバ側で呼び出されます。また、プロトコル DTLS v1.3 を使用する場合、ハンドシェイクには常に Cookie 交換が含まれることを示します。プロトコル DTLS v1.3 を使用する場合、Cookie 交換はデフォルトで有効になっていることに注意してください。Cookie は現在のトランスクリプトのハッシュを保持しており、別のサーバプロセスが応答の ClientHello を処理できるようにします。secret は、Cookie データの完全性チェックを生成する際に使用されます。
int	wolfSSL_disable_hrr_cookie (WOLFSSL * ssl) この関数は、HelloRetryRequest メッセージに Cookie を含めてはならないこと、およびプロトコル DTLS v1.3 を使用している場合、ハンドシェイクに Cookie 交換を含めないことを示すために、サーバ側で呼び出されます。プロトコル DTLS v1.3 を使用する際に Cookie 交換を行わないと、サーバが DoS/増幅攻撃に対して脆弱になる可能性があることに注意してください。
int	wolfSSL_CTX_no_ticket_TLSv13 (WOLFSSL_CTX * ctx) この関数は、ハンドシェイクが完了した後、再開セッションチケットの送信を停止するために、サーバで呼び出されます。
int	wolfSSL_no_ticket_TLSv13 (WOLFSSL * ssl) この関数は、ハンドシェイクが完了した後、再開セッションチケットの送信を停止するために、サーバで呼び出されます。
int	wolfSSL_CTX_no_dhe_psk (WOLFSSL_CTX * ctx) この関数は、事前共有鍵を認証に使用するハンドシェイクの際に、Diffie_Hellman(DH) スタイルの鍵交換を禁止するために、TLS v1.3 wolfSSL コンテキストで呼び出されます。
int	wolfSSL_no_dhe_psk (WOLFSSL * ssl) この関数は、事前共有鍵を認証に使用するハンドシェイクの際に、Diffie_Hellman(DH) スタイルの鍵交換を禁止するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。
int	wolfSSL_update_keys (WOLFSSL * ssl) この関数は、鍵のロールオーバーを強制するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。KeyUpdate メッセージがピアに送信され、暗号化用の新しい鍵が計算されます。ピアは KeyUpdate メッセージを送り返し、その後新しい復号鍵が計算されます。この関数は、ハンドシェイクが完了した後にのみ呼び出すことができます。

	Name
int	wolfSSL_key_update_response (WOLFSSL * ssl, int * required) この関数は、鍵のロールオーバーが進行中かどうかを判断するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。wolfSSL_update_keys() が呼び出されると、KeyUpdate メッセージが送信され、暗号化鍵が更新されます。復号鍵は、応答を受信したときに更新されます。
int	wolfSSL_CTX_allow_post_handshake_auth (WOLFSSL_CTX * ctx) この関数は、サーバからの要求に応じてクライアント証明書をハンドシェイク後に送信できるようにするために、TLS v1.3 クライアント wolfSSL コンテキストで呼び出されます。これは、クライアント認証が必要なページとそうでないページを持つ Web サーバに接続する際に便利です。
int	wolfSSL_allow_post_handshake_auth (WOLFSSL * ssl) この関数は、サーバからの要求に応じてクライアント証明書をハンドシェイク後に送信できるようにするために、TLS v1.3 クライアント wolfSSL で呼び出されます。Post-Handshake Client Authentication 拡張が ClientHello で送信されます。これは、クライアント認証が必要なページとそうでないページを持つ Web サーバに接続する際に便利です。
int	wolfSSL_request_certificate (WOLFSSL * ssl) この関数は、TLS v1.3 クライアントからクライアント証明書を要求します。これは、Web サーバがクライアント認証を必要とするページと必要としないページの両方を提供している場合に便利です。接続上で最大 256 回の要求を送信できます。
int	wolfSSL_CTX_set1_sigalgs_list (WOLFSSL_CTX * ctx, const char * list) この関数は、wolfSSL コンテキストで優先順位に従って許可する署名アルゴリズムのリストを設定します。リストは null 終端のテキスト文字列で、コロン区切りのリストです。各要素は "+" の形式 (例: "RSA-PSS+SHA256:ECDSA+SHA384") で指定します。Edwards 曲線アルゴリズムである "ED25519" および "ED448" はハッシュがアルゴリズムに内包されるため、サフィックスを付けずに記述します。コンテキストに既に保持されているリストは置き換えられます。
int	** wolfSSL_set1_sigalgs_list の説明を参照してください。セッションに既に保持されているリストは置き換えられます。

	Name
int	wolfSSL_CTX_set1_groups_list (WOLFSSL_CTX * ctx, const char * list) この関数は、wolfSSL コンテキストで優先順位に従って許可する鍵交換グループ（名前付き楕円曲線および KEM）のリストを設定します。リストは null 終端のテキスト文字列で、グループ名をコロンで区切って指定します（例: "P-384:P-256:X25519"）。TLS v1.3 接続で使用する鍵交換パラメータを設定するために本関数を呼び出します（本関数は HAVE_ECC、WOLFSSL_TLS13、HAVE_SUPPORTED_CURVES がすべて定義された場合にのみ提供されます）。
int	** wolfSSL_set1_groups_list の説明を参照してください。
int	wolfSSL_preferred_group (WOLFSSL * ssl) この関数は、TLS v1.3 ハンドシェイクでクライアントが優先的に使用したい鍵交換グループを返します。ハンドシェイク完了後にこの関数を呼び出して、サーバが優先するグループを判定し、この情報を将来の接続で使用して鍵交換用の鍵ペアを事前生成できます。
int	wolfSSL_CTX_set_groups (WOLFSSL_CTX * ctx, int * groups, int count) この関数は、wolfSSL コンテキストで優先順位に従って許可する鍵交換グループ（名前付き楕円曲線および KEM）のリストを設定します。リストは名前付きグループ識別子（下表参照）の配列で、count に配列内の識別子数を指定します。TLS v1.3 接続で使用する鍵交換パラメータを設定するために本関数を呼び出します。配列の順序がそのまま KeyShare の優先順位として用いられます。
int	** wolfSSL_set_groups の説明を参照してください。

	Name
int	wolfSSL_connect_TLSv13(WOLFSSL * ssl) この関数はクライアント側で呼び出され、サーバとの TLS v1.3 ハンドシェイクを開始します。この関数が呼び出されるとき、基礎となる通信チャネルはすでに設定されています。wolfSSL_connect() はブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、基礎となる I/O が wolfSSL_connect() がハンドシェイクを続行するために必要なものを満たせない場合、wolfSSL_connect() は返されます。この場合、wolfSSL_get_error() を呼び出すと SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、基礎となる I/O の準備ができたときに wolfSSL_connect() の呼び出しを繰り返す必要があり、wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキング I/O の場合、wolfSSL_connect() はハンドシェイクが完了するかエラーが発生するまで返されません。wolfSSL は証明書検証に OpenSSL とは異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバを検証することです。つまり、サーバを検証するための CA をロードしない場合、接続エラー「検証できません (155)」が発生します。サーバの検証が失敗しても SSL_connect が成功するという OpenSSL の動作を模倣し、セキュリティを低下させたい場合は、SSL_new() を呼び出す前に SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0) を呼び出すことでこれを行うことができます。ただし、これは推奨されません。

	Name
	wolfSSL_accept_TLSv13(WOLFSSL * ssl) この関数はサーバ側で呼び出され、SSL/TLS クライアントが SSL/TLS ハンドシェイクを開始するのを待ちます。この関数が呼び出されると、基礎となる通信チャンネルはすでに設定されています。 wolfSSL_accept() はブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、基礎となる I/O が wolfSSL_accept() がハンドシェイクを続行するために必要なものを満たせない場合、wolfSSL_accept() は返されます。この場合、wolfSSL_get_error() を呼び出すと SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、データが読み取り可能になったときに wolfSSL_accept の呼び出しを繰り返す必要があります、wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキング I/O の場合、wolfSSL_accept() はハンドシェイクが完了するかエラーが発生するまで返されません。TLS v1.3 接続を期待する場合にこの関数を呼び出してください。ただし、古いバージョンの ClientHello メッセージもサポートされています。
int	wolfSSL_CTX_set_max_early_data(WOLFSSL_CTX * ctx, unsigned int sz) この関数は、TLS v1.3 クライアントまたはサーバが wolfSSL コンテキストを使用して交換する意思のある早期データの最大量を設定します。リプレイ攻撃を軽減するために処理する早期データの量を制限するには、この関数を呼び出してください。早期データは、セッションチケットが送信された接続の鍵から派生した鍵によって保護されるため、セッションチケットが再開に使用されるたびに同じになります。この値は再開用のセッションチケットに含まれます。サーバの値がゼロの場合、セッションチケットを使用してクライアントが早期データを送信しないことを示します。クライアントの値がゼロの場合、クライアントが早期データを送信しないことを示します。早期データのバイト数は、アプリケーションで実用的に可能な限り低く保つことをお勧めします。

	Name
int	wolfSSL_set_max_early_data (WOLFSSL * ssl, unsigned int sz) この関数は、TLS v1.3 クライアントまたはサーバが交換する意思のある早期データの最大量を設定します。リプレイ攻撃を軽減するために処理する早期データの量を制限するには、この関数を呼び出してください。早期データは、セッションチケットが送信された接続の鍵から派生した鍵によって保護されるため、セッションチケットが再開に使用されるたびに同じになります。この値は再開用のセッションチケットに含まれます。サーバの値がゼロの場合、セッションチケットを使用してクライアントが早期データを送信しないことを示します。クライアントの値がゼロの場合、クライアントが早期データを送信しないことを示します。早期データのバイト数は、アプリケーションで実用的に可能な限り低く保つことをお勧めします。
int	wolfSSL_write_early_data (WOLFSSL * ssl, const void * data, int sz, int * outSz) この関数は再開時にサーバーにアーリーデータを書き込みます。wolfSSL_connect() または wolfSSL_connect_TLSv13() の前にこの関数を呼び出してください。この関数はクライアントでのみ使用されます。
int	wolfSSL_read_early_data (WOLFSSL * ssl, void * data, int sz, int * outSz) この関数は再開時にクライアントからのアーリーデータを読み取ります。クライアントを受け入れ、ハンドシェイクでアーリーデータを読み取るには、wolfSSL_accept() または wolfSSL_accept_TLSv13() の代わりにこの関数を呼び出します。wolfSSL_is_init_finished() が true を返すまで関数を呼び出す必要があります。アーリーデータは複数のメッセージでクライアントから送信される場合があります。アーリーデータがない場合、ハンドシェイクは通常通り処理されます。この関数はサーバーでのみ使用されます。
int	wolfSSL_inject (WOLFSSL * ssl, const void * data, int sz) この関数は WOLFSSL オブジェクトにデータを注入するために呼び出されます。これは、データを単一の場所から読み取り、複数の接続に分割する必要がある場合に便利です。呼び出し元は wolfSSL_read() を呼び出して WOLFSSL オブジェクトから平文データを抽出する必要があります。

	Name
void	wolfSSL_CTX_set_psk_client_tls13_callback (WOLFSSL_CTX * ctx, wc_psk_client_tls13_callback cb) この関数は TLS v1.3 接続のための事前共有鍵 (PSK) クライアント側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL_CTX 構造体の client_psk_tls13_cb メンバーを設定します。
void	wolfSSL_set_psk_client_tls13_callback (WOLFSSL * ssl, wc_psk_client_tls13_callback cb) この関数は TLS v1.3 接続のための事前共有鍵 (PSK) クライアント側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL 構造体の options フィールドの client_psk_tls13_cb メンバーを設定します。
void	wolfSSL_CTX_set_psk_server_tls13_callback (WOLFSSL_CTX * ctx, wc_psk_server_tls13_callback cb) この関数は TLS v1.3 接続のための事前共有鍵 (PSK) サーバー側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL_CTX 構造体の server_psk_tls13_cb メンバーを設定します。
void	wolfSSL_set_psk_server_tls13_callback (WOLFSSL * ssl, wc_psk_server_tls13_callback cb) この関数は TLS v1.3 接続のための事前共有鍵 (PSK) サーバー側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL 構造体の options フィールドの server_psk_tls13_cb メンバーを設定します。
int	wolfSSL_UseKeyShare (WOLFSSL * ssl, word16 group) この関数は鍵ペアの生成を含む、グループから鍵共有エントリを作成します。KeyShare 拡張には鍵交換用に生成されたすべての公開鍵が含まれます。この関数が呼び出されると、指定されたグループのみが含まれます。サーバーに対して優先グループが以前に確立されている場合に、この関数を呼び出します。
int	wolfSSL_NoKeyShares (WOLFSSL * ssl) この関数は ClientHello で鍵共有が送信されないようにするために呼び出されます。これにより、ハンドシェイクで鍵交換が必要な場合、サーバーは HelloRetryRequest で応答することになります。期待される鍵交換グループが不明で、不要な鍵の生成を避けたい場合にこの関数を呼び出します。鍵交換が必要な場合、ハンドシェイクを完了するために追加のラウンドトリップが必要になることに注意してください。

	Name
WOLFSSL_METHOD *	wolfTLSv1_3_server_method_ex (void * heap) この関数はアプリケーションがサーバーであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。
WOLFSSL_METHOD *	wolfTLSv1_3_client_method_ex (void * heap) この関数はアプリケーションがクライアントであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。
WOLFSSL_METHOD *	wolfTLSv1_3_server_method (void) この関数は、アプリケーションがサーバーであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体用のメモリを割り当て、初期化します。
WOLFSSL_METHOD *	wolfTLSv1_3_client_method (void) この関数は、アプリケーションがクライアントであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体用のメモリを割り当て、初期化します。
WOLFSSL_METHOD *	wolfTLSv1_3_method_ex (void * heap) この関数は、どちら側 (サーバー/クライアント) であるかがまだ決定されていないことを除いて、wolfTLSv1_3_client_method と同様の WOLFSSL_METHOD を返します。
WOLFSSL_METHOD *	wolfTLSv1_3_method (void) この関数は、どちら側 (サーバー/クライアント) であるかがまだ決定されていないことを除いて、wolfTLSv1_3_client_method と同様の WOLFSSL_METHOD を返します。
int	wolfSSL_CTX_set_ephemeral_key (WOLFSSL_CTX * ctx, int keyAlgo, const char * key, unsigned int keySz, int format) この関数は、テスト専用の固定/静的エフェメラル鍵を設定します。
int	wolfSSL_set_ephemeral_key (WOLFSSL * ssl, int keyAlgo, const char * key, unsigned int keySz, int format) この関数は、テスト専用の固定/静的エフェメラル鍵を設定します。
int	wolfSSL_CTX_get_ephemeral_key (WOLFSSL_CTX * ctx, int keyAlgo, const unsigned char ** key, unsigned int * keySz) この関数は、ASN.1/DER としてロードされた鍵へのポインタを返します。

	Name
int	wolfSSL_get_ephemeral_key (WOLFSSL * ssl, int keyAlgo, const unsigned char ** key, unsigned int * keySz) この関数は、ASN.1/DER としてロードされた鍵へのポインタを返します。
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) 選択したメッセージダイジェスト、パディング、および RSA 鍵でメッセージに署名します。
int	wolfSSL_dtls13_has_pending_msg (WOLFSSL * ssl) DTLSv1.3 スタックが送信したが、まだ他のピアから確認応答されていないメッセージがあるかどうかをチェックします。
unsigned int	wolfSSL_SESSION_get_max_early_data (const WOLFSSL_SESSION * s) セッションから Early Data の最大サイズを取得します。
int	wolfSSL_CRYPTOP_EX_new_index (int class_index, long argl, void * argp, WOLFSSL_CRYPTOP_EX_new * new_func, WOLFSSL_CRYPTOP_EX_dup * dup_func, WOLFSSL_CRYPTOP_EX_free * free_func) 外部データ用の新しいインデックスを取得します。このエントリは以下の API にも適用されます:
int	wolfSSL_CTX_set_client_cert_type (WOLFSSL_CTX * ctx, const char * buf, int len) この関数がクライアント側で呼び出された場合、ピアに送信できる証明書タイプを設定します。サーバー側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が "Raw public key" 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。
int	wolfSSL_CTX_set_server_cert_type (WOLFSSL_CTX * ctx, const char * buf, int len) この関数がサーバー側で呼び出された場合、ピアに送信できる証明書タイプを設定します。クライアント側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が "Raw public key" 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。

	Name
int	wolfSSL_set_client_cert_type (WOLFSSL * ssl, const char * buf, int len) この関数がクライアント側で呼び出された場合、ピアに送信できる証明書タイプを設定します。サーバー側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が "Raw public key" 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。
int	wolfSSL_set_server_cert_type (WOLFSSL * ssl, const char * buf, int len) この関数がサーバー側で呼び出された場合、ピアに送信できる証明書タイプを設定します。クライアント側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が "Raw public key" 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。
int	wolfSSL_CTX_clear_group_messages (WOLFSSL_CTX * ctx) 指定された WOLFSSL_CTX コンテキストに対してハンドシェイクメッセージグループ化を無効にします。
int	wolfSSL_clear_group_messages (WOLFSSL * ssl) 指定された WOLFSSL オブジェクトに対してハンドシェイクメッセージグループ化を無効にします。
int	wolfSSL_get_negotiated_client_cert_type (WOLFSSL * ssl, int * tp) この関数は、ClientHello と ServerHello で行われたクライアント証明書タイプネゴシエーションの結果を返します。ネゴシエーションが発生しなかった場合、戻り値として WOLFSSL_SUCCESS が返され、証明書タイプとして WOLFSSL_CERT_TYPE_UNKNOWN が返されます。
int	wolfSSL_get_negotiated_server_cert_type (WOLFSSL * ssl, int * tp) この関数は、ClientHello と ServerHello で行われたサーバ証明書タイプネゴシエーションの結果を返します。ネゴシエーションが発生しなかった場合、戻り値として WOLFSSL_SUCCESS が返され、証明書タイプとして WOLFSSL_CERT_TYPE_UNKNOWN が返されます。

	Name
int	wolfSSL_dtls_cid_use (WOLFSSL * ssl) SSL オブジェクトに対して ConnectionID 拡張の使用を有効にします。RFC 9146 および RFC 9147 を参照してください。
int	wolfSSL_dtls_cid_is_enabled (WOLFSSL * ssl) ハンドシェイク完了後に呼び出された場合、SSL オブジェクトに対して ConnectionID が正常にネゴシエートされたかどうかを確認します。RFC 9146 および RFC 9147 を参照してください。
int	wolfSSL_dtls_cid_set (WOLFSSL * ssl, unsigned char * cid, unsigned int size) この接続でレコードを送信する際に相手ピアが使用する ConnectionID を設定します。RFC 9146 および RFC 9147 を参照してください。ConnectionID は最大 DTLS_CID_MAX_SIZE(調整可能なコンパイル時定義) である必要があり、255 バイトを超えることはできません。
int	wolfSSL_dtls_cid_get_rx_size (WOLFSSL * ssl, unsigned int * size) この接続でレコードを送信する際に相手ピアが使用する ConnectionID のサイズを取得します。RFC 9146 および RFC 9147 を参照してください。サイズはパラメータ size に格納されます。
int	wolfSSL_dtls_cid_get_rx (WOLFSSL * ssl, unsigned char * buffer, unsigned int bufferSz) この接続でレコードを送信する際に相手ピアが使用する ConnectionID を、パラメータ buffer が指すバッファにコピーします。RFC 9146 および RFC 9147 を参照してください。bufferSz でバッファ内の利用可能なスペースを提供する必要があります。
int	wolfSSL_dtls_cid_get0_rx (WOLFSSL * ssl, unsigned char ** cid) 相手ピアが使用する ConnectionID を取得します。RFC 9146 および RFC 9147 を参照してください。
int	wolfSSL_dtls_cid_get_tx_size (WOLFSSL * ssl, unsigned int * size) この接続でレコードを送信する際に使用する ConnectionID のサイズを取得します。RFC 9146 および RFC 9147 を参照してください。サイズはパラメータ size に格納されます。
int	wolfSSL_dtls_cid_get_tx (WOLFSSL * ssl, unsigned char * buffer, unsigned int bufferSz) この接続でレコードを送信する際に使用する ConnectionID を、パラメータ buffer が指すバッファにコピーします。RFC 9146 および RFC 9147 を参照してください。bufferSz で利用可能なサイズを提供する必要があります。
int	wolfSSL_dtls_cid_get0_tx (WOLFSSL * ssl, unsigned char ** cid) この接続でレコードを送信する際に使用する ConnectionID を取得します。RFC 9146 および RFC 9147 を参照してください。

	Name
const unsigned char *	wolfSSL_dtls_cid_parse (const unsigned char * msg, unsigned int msgSz, unsigned int cidSz) レコードデータグラム/メッセージから ConnectionID を抽出します。RFC 9146 および RFC 9147 を参照してください。
void	** wolfSSL_CTX_set_client_CA_list (WOLFSSL_X509_NAME * names) サーバ側では、この関数は証明書要求でクライアントに送信される CA 名のリストを設定します。これは、サーバがサポートする CA のヒントとして機能します。
WOLFSSL_STACK *	wolfSSL_CTX_get_client_CA_list (const WOLFSSL_CTX * ctx) これは、wolfSSL_CTX_set_client_CA_list を介して以前に設定されたリストを取得します。リストが設定されていない場合は NULL を返します。
void	** wolfSSL_set_client_CA_list (WOLFSSL_X509_NAME * names)wolfSSL_CTX_set_client_CA_list と同じですが、セッション固有です。CA リストがコンテキストとセッションの両方に設定されている場合、セッション上のリストが使用されます。
WOLFSSL_STACK *	wolfSSL_get_client_CA_list (const WOLFSSL * ssl) サーバ側では、wolfSSL_set_client_CA_list を介して以前に設定されたリストを取得します。何も設定されていない場合は、wolfSSL_CTX_set_client_CA_list を介して以前に設定されたリストを返します。リストが全く設定されていない場合は、NULL を返します。
void	** wolfSSL_CTX_set0_CA_list (WOLFSSL_X509_NAME * names) この関数は、ピアの認証でサポートされている CA のヒントとしてピアに送信される CA 名のリストを設定します。
WOLFSSL_STACK *	wolfSSL_CTX_get0_CA_list (const WOLFSSL_CTX * ctx) これは、wolfSSL_CTX_set0_CA_list を介して以前に設定されたリストを取得します。リストが設定されていない場合は NULL を返します。
void	** wolfSSL_set0_CA_list (WOLFSSL_X509_NAME * names)wolfSSL_CTX_set0_CA_list と同じですが、セッション固有です。CA リストがコンテキストとセッションの両方に設定されている場合、セッション上のリストが使用されます。
WOLFSSL_STACK *	wolfSSL_get0_CA_list (const WOLFSSL * ssl) これは、wolfSSL_set0_CA_list を介して以前に設定されたリストを取得します。何も設定されていない場合は、wolfSSL_CTX_set0_CA_list を介して以前に設定されたリストを返します。リストが全く設定されていない場合は、NULL を返します。
WOLFSSL_STACK *	wolfSSL_get0_peer_CA_list (const WOLFSSL * ssl) これは、ピアから受信した CA リストを返します。

	Name
void	wolfSSL_CTX_set_cert_cb (WOLFSSL_CTX * ctx, int)(WOLFSSL , void) cb, void arg) この関数は、証明書が使用される直前に呼び出されるコールバックを設定し、アプリケーションが証明書を検査、設定、またはクリアできるようにします。例えば、ピアから送信された CA リストに反応することができます。
int	wolfSSL_get_client_suites_sigalgs (const WOLFSSL * ssl, const byte ** suites, word16 * suiteSz, const byte ** hashSigAlgo, word16 * hashSigAlgoSz) この関数は、クライアントが提供する暗号スイートと署名アルゴリズムの生のリストを返します。リストは、wolfSSL_CTX_set_cert_cb() で設定されたコールバック内でのみ保存され、返されます。これは、利用可能な暗号スイートと署名アルゴリズムに基づいて証明書と鍵を動的にロードできるようにするのに便利です。
WOLFSSL_CIPHERSUITE_INFO	wolfSSL_get_ciphersuite_info (byte first, byte second) これは、生の暗号スイートバイトから直接暗号スイートに関する情報を返します。
int	wolfSSL_get_sigalg_info (byte first, byte second, int * hashAlgo, int * sigAlgo) これは、生の暗号スイートバイトから直接ハッシュおよび署名アルゴリズムに関する情報を返します。

C.52.2 Functions Documentation

C.52.2.1 function wolfDTLSv1_2_client_method_ex

```
WOLFSSL_METHOD * wolfDTLSv1_2_client_method_ex(
    void * heap
)
```

この関数は DTLS v1.2 クライアントメソッドを初期化します。

Parameters:

- **none** パラメータはありません。

See:

- **wolfSSL_Init**
- **wolfSSL_CTX_new**

Return: pointer この関数は新しい WOLFSSL_METHOD 構造体へのポインタを返します。

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfDTLSv1_2_client_method());
...
WOLFSSL* ssl = wolfSSL_new(ctx);
...
```

C.52.2.2 function wolfSSLv23_method

```
WOLFSSL_METHOD * wolfSSLv23_method(  
    void  
)
```

この関数は wolfSSLv23_client_method に似た WOLFSSL_METHOD を返しますが、どちら側 (サーバ/クライアント) であるかがまだ決定されていない点が異なります。

Parameters:

- **none** パラメータはありません。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- WOLFSSL_METHOD* 正常に作成された場合、WOLFSSL_METHOD ポインタを返します
- NULL メモリ割り当てエラーまたはメソッド作成失敗の場合は Null を返します

Example

```
WOLFSSL* ctx;  
ctx = wolfSSL_CTX_new(wolfSSLv23_method());  
// ret 値をチェック
```

C.52.2.3 function wolfSSLv3_server_method

```
WOLFSSL_METHOD * wolfSSLv3_server_method(  
    void  
)
```

[wolfSSLv3_server_method\(\)](#)関数は、アプリケーションがサーバであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>
```

```
WOLFSSL_METHOD* method;
```



```

WOLFSSL_CTX* ctx;

method = wolfSSLv3_server_method();
if (method == NULL) {
    メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

C.52.2.4 function wolfSSLv3_client_method

```

WOLFSSL_METHOD * wolfSSLv3_client_method(
    void
)

```

wolfSSLv3_client_method()関数は、アプリケーションがクライアントであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```

#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_client_method();
if (method == NULL) {
    メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

C.52.2.5 function wolfTLSv1_server_method

```
WOLFSSL_METHOD * wolfTLSv1_server_method(
    void
)
```

`wolfTLSv1_server_method()`関数は、アプリケーションがサーバであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、`wolfSSL_CTX_new()` で SSL/TLS コンテキストを作成する際に使用される新しい `wolfSSL_METHOD` 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された `WOLFSSL_METHOD` 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる `malloc()` 実装の失敗値が返されます (通常は NULL で、`errno` が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_server_method();
if (method == NULL) {
    メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.6 function wolfTLSv1_client_method

```
WOLFSSL_METHOD * wolfTLSv1_client_method(
    void
)
```

`wolfTLSv1_client_method()`関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、`wolfSSL_CTX_new()` で SSL/TLS コンテキストを作成する際に使用される新しい `wolfSSL_METHOD` 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>
```

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
```

```
method = wolfTLSv1_client_method();
if (method == NULL) {
    メソッドを取得できません
}
```

```
ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.7 function wolfTLSv1_1_server_method

```
WOLFSSL_METHOD * wolfTLSv1_1_server_method(
    void
)
```

wolfTLSv1_1_server_method()関数は、アプリケーションがサーバであり、TLS 1.1 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- none パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>
```

```

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_server_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

C.52.2.8 function wolfTLSv1_1_client_method

```

WOLFSSL_METHOD * wolfTLSv1_1_client_method(
    void
)

```

wolfTLSv1_1_client_method()関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```

#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_client_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

C.52.2.9 function wolfTLSv1_2_server_method

```
WOLFSSL_METHOD * wolfTLSv1_2_server_method(
    void
)
```

`wolfTLSv1_2_server_method()`関数は、アプリケーションがサーバであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、`wolfSSL_CTX_new()` で SSL/TLS コンテキストを作成する際に使用される新しい `wolfSSL_METHOD` 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された `WOLFSSL_METHOD` 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる `malloc()` 実装の失敗値が返されます (通常は NULL で、`errno` が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_server_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.10 function wolfTLSv1_2_client_method

```
WOLFSSL_METHOD * wolfTLSv1_2_client_method(
    void
)
```

`wolfTLSv1_2_client_method()`関数は、アプリケーションがクライアントであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、`wolfSSL_CTX_new()` で SSL/TLS コンテキストを作成する際に使用される新しい `wolfSSL_METHOD` 構造体のためのメモリを割り当てて初期化します。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>
```

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
```

```
method = wolfTLSv1_2_client_method();
if (method == NULL) {
    // メソッドを取得できません
}
```

```
ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.11 function wolfDTLSv1_client_method

```
WOLFSSL_METHOD * wolfDTLSv1_client_method(
    void
)
```

wolfDTLSv1_client_method()関数は、アプリケーションがクライアントであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポート付きでコンパイルされている場合 (-enable_dtls、または wolfSSL_DTLS を定義することによって) にのみ利用可能です。

Parameters:

- none パラメータはありません。

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:

- * 成功した場合、この呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_client_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

C.52.2.12 function wolfDTLSv1_server_method

```

WOLFSSL_METHOD * wolfDTLSv1_server_method(
    void
)

```

wolfDTLSv1_server_method()関数は、アプリケーションがサーバーであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です (-enable-dtls、または wolfSSL_DTLS を定義することによって)。

Parameters:

- **none** パラメータはありません。

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:

- * 成功した場合、この呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL_XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_server_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...

```

C.52.2.13 function wolfDTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_server_method(  
    void  
)
```

`wolfDTLSv1_3_server_method()`関数は、アプリケーションがサーバーであり、DTLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、`wolfSSL_CTX_new()` で SSL/TLS コンテキストを作成する際に使用される新しい `wolfSSL_METHOD` 構造体のメモリを割り当て、初期化します。この関数は、`wolfSSL` が DTLSv1.3 サポートを有効にしてコンパイルされている場合にのみ使用可能です (`-enable-dtls13`、または `wolfSSL_DTLS13` を定義することによって)。

Parameters:

- **none** パラメータはありません。

See: `wolfDTLSv1_3_client_method`

Return:

- * 成功した場合、この呼び出しは新しく作成された `WOLFSSL_METHOD` 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる `malloc()` 実装の失敗値が返されます (通常は NULL で、`errno` が ENOMEM に設定されます)。

Example

```
WOLFSSL_METHOD* method;  
WOLFSSL_CTX* ctx;  
  
method = wolfDTLSv1_3_server_method();  
if (method == NULL) {  
    // メソッドを取得できません  
}  
  
ctx = wolfSSL_CTX_new(method);  
...
```

C.52.2.14 function wolfDTLSv1_3_client_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_client_method(  
    void  
)
```

`wolfDTLSv1_3_client_method()`関数は、アプリケーションがクライアントであり、DTLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、`wolfSSL_CTX_new()` で SSL/TLS コンテキストを作成する際に使用される新しい `wolfSSL_METHOD` 構造体のメモリを割り当て、初期化します。この関数は、`wolfSSL` が DTLSv1.3 サポートを有効にしてコンパイルされている場合にのみ使用可能です (`-enable-dtls13`、または `wolfSSL_DTLS13` を定義することによって)。

Parameters:

- **none** パラメータはありません。

See: `wolfDTLSv1_3_server_method`

Return:

- * 成功した場合、この呼び出しは新しく作成された `WOLFSSL_METHOD` 構造体へのポインタを返します。
- FAIL XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる `malloc()` 実装の失敗値が返されます (通常は NULL で、`errno` が ENOMEM に設定されます)。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_client_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.15 function wolfDTLS_server_method

```
WOLFSSL_METHOD * wolfDTLS_server_method(
    void
)
```

wolfDTLS_server_method()関数は、アプリケーションがサーバーであり、利用可能な最新バージョンの DTLS および許可される最小バージョンまでのすべてのバージョンをサポートすることを示すために使用されます。デフォルトの許可される最小バージョンは、WOLFSSL_MIN_DTLS_DOWNGRADE の定義に基づいており、wolfSSL_SetMinVersion() を使用して実行時に変更できます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。この関数は、wolfSSL が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です（-enable-dtls、または wolfSSL_DTLS を定義することによって）。

Parameters:

- **none** パラメータはありません。

See:

- **wolfDTLS_client_method**
- **wolfSSL_SetMinVersion**

Return:

- * 成功した場合、この呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL_XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます（通常は NULL で、errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_server_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.16 function wolfDTLS_client_method

```
WOLFSSL_METHOD * wolfDTLS_client_method(
    void
)
```

`wolfDTLS_client_method()`関数は、アプリケーションがクライアントであり、利用可能な最新バージョンの DTLS および許可される最小バージョンまでのすべてのバージョンをサポートすることを示すために使用されます。デフォルトの許可される最小バージョンは、WOLFSSL_MIN_DTLS_DOWNGRADE の定義に基づいており、`wolfSSL_SetMinVersion()` を使用して実行時に変更できます。この関数は、`wolfSSL_CTX_new()` で SSL/TLS コンテキストを作成する際に使用される新しい `wolfSSL_METHOD` 構造体のメモリを割り当て、初期化します。この関数は、`wolfSSL` が DTLS サポートを有効にしてコンパイルされている場合にのみ使用可能です（`-enable-dtls`、または `wolfSSL_DTLS` を定義することによって）。

Parameters:

- **none** パラメータはありません。

See:

- `wolfDTLS_server_method`
- `wolfSSL_SetMinVersion`

Return:

- * 成功した場合、この呼び出しは新しく作成された `WOLFSSL_METHOD` 構造体へのポインタを返します。
- FAIL_XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる `malloc()` 実装の失敗値が返されます（通常は NULL で、`errno` が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_client_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.17 function wolfDTLSv1_2_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_2_server_method(
    void
)
```

この関数は、サーバー側用の `WOLFSSL_METHOD` を作成し、初期化します。

Parameters:

- **none** パラメータはありません。

See: `wolfSSL_CTX_new`

Return: この関数は `WOLFSSL_METHOD` ポインタを返します。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfDTLSv1_2_server_method());
WOLFSSL* ssl = wolfSSL_new(ctx);
...
```

C.52.2.18 function wolfSSL_use_old_poly

```
int wolfSSL_use_old_poly(
    WOLFSSL * ssl,
    int value
)
```

chacha-poly AEAD 構築の最初のリリースと新しいバージョンの間にはいくつかの違いがあるため、古いバージョンを使用するサーバー/クライアントと通信するためのオプションを追加しました。デフォルトでは、wolfSSL は新しいバージョンを使用します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **value** poly1305 の情報を設定する際に古いバージョンを使用するかどうか。フラグ値 1 を渡すと古い poly AEAD を使用することを示し、新しいバージョンの使用に戻すにはフラグ値 0 を渡します。

See: none

Return: 0 成功時

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_use_old_poly(ssl, 1);
if (ret != 0) {
    // poly1305 AEAD バージョンの設定に失敗しました
}
```

C.52.2.19 function wolfSSL_dtls_import

```
int wolfSSL_dtls_import(
    WOLFSSL * ssl,
    const unsigned char * buf,
    unsigned int sz
)
```

wolfSSL_dtls_import() 関数は、シリアル化されたセッション状態を解析するために使用されます。これにより、ハンドシェイクが完了した後に接続を再開できます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** インポートするシリアル化されたセッション。
- **sz** シリアル化されたセッションバッファのサイズ。

See:

- [wolfSSL_new](#)
- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_dtls_set_export](#)

Return:

- Success 成功した場合、読み取られたバッファの量が返されます。
- Failure すべての失敗した戻り値は 0 未満になります。
- VERSION_ERROR バージョンの不一致が見つかった場合、つまり DTLS v1 で ctx が DTLS v1.2 用に設定されている場合、VERSION_ERROR が返されます。

Example

```

WOLFSSL* ssl;
int ret;
unsigned char buf[MAX];
bufSz = MAX;
...
// wc_dtls_export 関数から送信された情報を取得し、buf に配置します
fread(buf, 1, bufSz, input);
ret = wolfSSL_dtls_import(ssl, buf, bufSz);
if (ret < 0) {
    // エラーケースを処理します
}
// ハンドシェイクが既に完了しているため、wolfSSL_accept は不要です
...
ret = wolfSSL_write(ssl) および wolfSSL_read(ssl);
...

```

C.52.2.20 function wolfSSL_tls_import

```

int wolfSSL_tls_import(
    WOLFSSL * ssl,
    const unsigned char * buf,
    unsigned int sz
)

```

シリアル化された TLS セッションをインポートするために使用されます。この関数は、接続の状態をインポートするためのものです。警告：buf には状態に関する機密情報が含まれており、保存する場合は暗号化してから保存するのが最善です。マクロ WOLFSSL_SESSION_EXPORT_DEBUG を定義することで、追加のデバッグ情報を表示できます。

Parameters:

- **ssl** セッションをインポートする WOLFSSL 構造体
- **buf** シリアル化されたセッション
- **sz** バッファ'buf' のサイズ

See:

- [wolfSSL_dtls_import](#)
- [wolfSSL_tls_export](#)

Return: バッファ'buf' から読み取られたバイト数

C.52.2.21 function wolfSSL_CTX_dtls_set_export

```

int wolfSSL_CTX_dtls_set_export(
    WOLFSSL_CTX * ctx,
    wc_dtls_export func
)

```

wolfSSL_CTX_dtls_set_export() 関数は、セッションをエクスポートするためのコールバック関数を設定するために使用されます。以前に保存されたエクスポート関数をクリアするために、パラメータ func として NULL を渡すことができます。サーバー側で使用され、ハンドシェイクが完了した直後に呼び出されます。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された WOLFSSL_CTX 構造体へのポインタ。
- **func** セッションをエクスポートする際に使用する wc_dtls_export 関数。

See:

- [wolfSSL_new](#)
- [wolfSSL_CTX_new](#)
- [wolfSSL_dtls_set_export](#)
- Static buffer use

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG null または期待されない引数が渡された場合

Example

```
int send_session(WOLFSSL* ssl, byte* buf, word32 sz, void* userCtx);
// buf (シリアル化されたセッション) を宛先に渡す send_session(wc_dtls_export) の本体
WOLFSSL_CTX* ctx;
int ret;
...
ret = wolfSSL_CTX_dtls_set_export(ctx, send_session);
if (ret != SSL_SUCCESS) {
    // エラーケースを処理します
}
...
ret = wolfSSL_accept(ssl);
...
```

C.52.22 function wolfSSL_dtls_set_export

```
int wolfSSL_dtls_set_export(
    WOLFSSL * ssl,
    wc_dtls_export func
)
```

wolfSSL_dtls_set_export() 関数は、セッションをエクスポートするためのコールバック関数を設定するために使用されます。以前に保存されたエクスポート関数をクリアするために、パラメータ func として NULL を渡すことができます。サーバー側で使用され、ハンドシェイクが完了した直後に呼び出されます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **func** セッションをエクスポートする際に使用する wc_dtls_export 関数。

See:

- [wolfSSL_new](#)
- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_dtls_set_export](#)

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG null または期待されない引数が渡された場合

Example

```
int send_session(WOLFSSL* ssl, byte* buf, word32 sz, void* userCtx);
// send_session の本体 (wc_dtls_export)
// buf(シリアル化されたセッション) を宛先に渡す
WOLFSSL* ssl;
int ret;
```

```

...
ret = wolfSSL_dtls_set_export(ssl, send_session);
if (ret != SSL_SUCCESS) {
    // エラーケースの処理
}
...
ret = wolfSSL_accept(ssl);
...

```

C.52.2.23 function wolfSSL_dtls_export

```

int wolfSSL_dtls_export(
    WOLFSSL * ssl,
    unsigned char * buf,
    unsigned int * sz
)

```

wolfSSL_dtls_export() 関数は、WOLFSSL セッションを提供されたバッファにシリアル化するために使用されます。セッションを送信するための関数コールバックを使用するよりもメモリオーバーヘッドが少なく、セッションをシリアル化するタイミングを選択できます。関数に渡されたときに buffer が NULL の場合、sz は WOLFSSL セッションをシリアル化するために必要なバッファのサイズに設定されます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** シリアル化されたセッションを保持するバッファ。
- **sz** バッファのサイズ。

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_CTX_dtls_set_export
- wolfSSL_dtls_import

Return:

- Success 成功した場合、使用されたバッファの量が返されます。
- Failure すべての失敗時の戻り値は 0 未満になります。

Example

```

WOLFSSL* ssl;
int ret;
unsigned char buf[MAX];
bufSz = MAX;
...
ret = wolfSSL_dtls_export(ssl, buf, bufSz);
if (ret < 0) {
    // エラーケースの処理
}
...

```

C.52.2.24 function wolfSSL_tls_export

```

int wolfSSL_tls_export(
    WOLFSSL * ssl,
    unsigned char * buf,

```

```
    unsigned int * sz
)
```

シリアル化された TLS セッションをエクスポートするために使用されます。この関数は接続のシリアル化された状態をエクスポートするためのものです。ほとんどの場合、wolfSSL_tls_export の代わりに wolfSSL_get1_session を使用する必要があります。追加のデバッグ情報は、マクロ WOLFSSL_SESSION_EXPORT_DEBUG が定義されている場合に表示できます。警告: buf には状態に関する機密情報が含まれているため、保存する場合は暗号化してから保存することが最善です。

Parameters:

- **ssl** セッションをエクスポートする WOLFSSL 構造体
- **buf** シリアル化されたセッションの出力
- **sz** 'buf' に設定されたバイト単位のサイズ

See:

- wolfSSL_dtls_import
- wolfSSL_tls_import

Return: バッファ'buf' に書き込まれたバイト数

C.52.2.25 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
)
```

この関数は、CTX 用の静的メモリを確保するために使用されます。確保されたメモリは、CTX のライフタイム中および CTX から作成された SSL オブジェクトに使用されます。NULL の ctx ポインタと wolfSSL_method_func 関数を渡すことで、CTX 自体の作成も静的メモリを使用します。wolfSSL_method_func は、WOLFSSL_METHOD* (wolfSSL_method_func)(void heap); という関数シグネチャを持ちます。max に 0 を渡すと、設定されていないかのように動作し、最大同時使用制限は適用されません。渡された flag 値は、メモリの使用方法と動作中の挙動を決定します。利用可能なフラグは次のとおりです: 0 - デフォルトの一般メモリ、WOLFMEM_IO_POOL - メッセージの送受信時の入出力バッファに使用され、一般メモリをオーバーライドするため、渡されたバッファ内のすべてのメモリが IO に使用されます、WOLFMEM_IO_FIXED - WOLFMEM_IO_POOL と同じですが、各 SSL がライフタイム中に 2 つのバッファを保持します、WOLFMEM_TRACK_STATS - 各 SSL が実行中にメモリ統計を追跡します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタのアドレス。
- **method** プロトコルを作成する関数。(ctx も NULL でない場合は NULL であるべきです)
- **buf** すべての操作に使用するメモリ。
- **sz** 渡されるメモリバッファのサイズ。
- **flag** メモリのタイプ。
- **max** 最大同時操作数。

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_is_static_memory
- wolfSSL_is_static_memory

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE 失敗時。

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;
...
// 静的メモリを使用して ctx も作成、使用する一般メモリから開始
ctx = NULL;
ret = wolfSSL_CTX_load_static_memory(&ctx, wolfSSLv23_server_method_ex,
memory, memorySz, 0, MAX_CONCURRENT_HANDSHAKES);
if (ret != SSL_SUCCESS) {
    // エラーケースの処理
}
// IO で使用するメモリをロード
ret = wolfSSL_CTX_load_static_memory(&ctx, NULL, IO, IOSz, flag,
MAX_CONCURRENT_IO);
if (ret != SSL_SUCCESS) {
    // エラーケースの処理
}
...
```

C.52.2.26 function wolfSSL_CTX_is_static_memory

```
int wolfSSL_CTX_is_static_memory(
    WOLFSSL_CTX * ctx,
    WOLFSSL_MEM_STATS * mem_stats
)
```

この関数は接続の動作を変更せず、静的メモリの使用状況に関する情報を収集するためにのみ使用されます。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **mem_stats** 静的メモリの使用状況に関する情報を保持する構造体。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_load_static_memory](#)
- [wolfSSL_is_static_memory](#)

Return:

- 1 CTX が静的メモリを使用している場合に true として返されます。
- 0 静的メモリを使用していない場合に返されます。

Example

```
WOLFSSL_CTX* ctx;
int ret;
WOLFSSL_MEM_STATS mem_stats;
```



```

...
//CTX による静的メモリに関する情報を取得
ret = wolfSSL_CTX_is_static_memory(ctx, &mem_stats);
if (ret == 1) {
    // 静的メモリを使用しているケースの処理
    // mem_stats の要素を出力または検査
}
if (ret == 0) {
    //ctx が静的メモリを使用していないケースの処理
}
...

```

C.52.2.27 function wolfSSL_is_static_memory

```

int wolfSSL_is_static_memory(
    WOLFSSL * ssl,
    WOLFSSL_MEM_CONN_STATS * mem_stats
)

```

wolfSSL_is_static_memory は、SSL の静的メモリ使用状況に関する情報を収集するために使用されます。戻り値は静的メモリが使用されているかどうかを示し、WOLFSSL_MEM_CONN_STATS は、静的メモリをロードする際に親 CTX に WOLFMEM_TRACK_STATS フラグが渡された場合にのみ入力されます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **mem_stats** 静的メモリの使用状況を含む構造体。

See:

- wolfSSL_new
- wolfSSL_CTX_is_static_memory

Return:

- 1 CTX が静的メモリを使用している場合に true として返されます。
- 0 静的メモリを使用していない場合に返されます。

Example

```

WOLFSSL* ssl;
int ret;
WOLFSSL_MEM_CONN_STATS mem_stats;
...
ret = wolfSSL_is_static_memory(ssl, mem_stats);
if (ret == 1) {
    // 静的メモリの場合のケースを処理
    // WOLFMEM_TRACK_STATS フラグがある場合、mem_stats 内の要素を調査
}
...

```

C.52.2.28 function wolfSSL_CTX_use_certificate_file

```

int wolfSSL_CTX_use_certificate_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)

```

この関数は、証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは file 引数によって提供されます。format 引数はファイルのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM のいずれかです。適切な使用方法については、example を参照してください。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインタ
- **file** wolfSSL SSL コンテキストにロードされる証明書を含むファイルの名前へのポインタ。
- **format** - file が指す証明書のフォーマット。可能なオプションは SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE 関数呼び出しが失敗した場合、考えられる原因には、ファイルが間違ったフォーマットである、または "format" 引数を使用して間違ったフォーマットが指定されている、ファイルが存在しない、読み取れない、または破損している、メモリ不足の状態が発生している、ファイルの Base16 デコードが失敗したことなどが含まれます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_file(ctx, "../client-cert.pem",
                                     SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // 証明書ファイルのロードエラー
}
...
```

C.52.2.29 function wolfSSL_CTX_use_PrivateKey_file

```
int wolfSSL_CTX_use_PrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

この関数は、秘密鍵ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは file 引数によって提供されます。format 引数はファイルのフォーマットタイプを指定します - SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用方法については、example を参照してください。

Parameters:

- **none** パラメータなし。

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_CTX_SetDevId

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE ファイルが間違ったフォーマットである、または “format” 引数を使用して間違ったフォーマットが指定されている。ファイルが存在しない、読み取れない、または破損している。メモリ不足の状態が発生している。ファイルの Base16 デコードが失敗した。キーファイルが暗号化されているが、パスワードが提供されていない。

外部キーストアを使用していて秘密鍵を持っていない場合、代わりに公開鍵を提供し、署名を処理するための暗号コールバックを登録できます。これには、暗号コールバックまたは PK コールバックのいずれかでビルドできます。暗号コールバックを有効にするには、`-enable-cryptocb` または `WOLF_CRYPTO_CB` でビルドし、`wc_CryptoCb_RegisterDevice` を使用して暗号コールバックを登録し、`wolfSSL_CTX_SetDevId` を使用して関連する `devId` を設定します。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_PrivateKey_file(ctx, "./server-key.pem",
                                     SSL_FILETYPE_PEM);

if (ret != SSL_SUCCESS) {
    // 鍵ファイルのロードエラー
}
...
```

C.52.2.30 function wolfSSL_CTX_load_verify_locations

```
int wolfSSL_CTX_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    const char * path
)
```

この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである場合があります。同じファイルに複数の CA 証明書が含まれている場合、wolfSSL はファイル内に提示された順序でそれらをロードします。path 引数は、信頼されたルート CA の証明書をを含むディレクトリの名前へのポインタです。file の値が NULL でない場合、必要であれば path を NULL として指定できます。path が指定され、ライブラリのビルド時に `NO_WOLFSSL_DIR` が定義されていない場合、wolfSSL は指定されたディレクトリにあるすべての CA 証明書をロードします。この関数はディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダー “--BEGIN CERTIFICATE--” を持つ PEM 形式の CERT_TYPE ファイルを想定しています。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **file** PEM 形式の CA 証明書をを含むファイルの名前へのポインタ。
- **path** PEM 形式の証明書をロードするディレクトリの名前へのポインタ。

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` 成功時。
- `SSL_FAILURE` ctx が NULL の場合、または file と path の両方が NULL の場合に返されます。
- `SSL_BAD_FILETYPE` ファイルが間違ったフォーマットの場合に返されます。
- `SSL_BAD_FILE` ファイルが存在しない、読み取れない、または破損している場合に返されます。
- `MEMORY_E` メモリ不足の状態が発生した場合に返されます。
- `ASN_INPUT_E` ファイルの Base16 デコードが失敗した場合に返されます。
- `ASN_BEFORE_DATE_E` 現在の日付が before date より前の場合に返されます。
- `ASN_AFTER_DATE_E` 現在の日付が after date より後の場合に返されます。
- `BUFFER_E` チェーンバッファが受信バッファより大きい場合に返されます。
- `BAD_PATH_ERROR` path を開こうとしたときに `opendir()` が失敗した場合に返されます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_verify_locations(ctx, "./ca-cert.pem", NULL);
if (ret != WOLFSSL_SUCCESS) {
    // CA 証明書のロードエラー
}
...
```

C.52.2.31 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
)
```

この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである場合があります。同じファイルに複数の CA 証明書が含まれている場合、wolfSSL はファイル内に提示された順序でそれらをロードします。path 引数は、信頼されたルート CA の証明書を含むディレクトリの名前へのポインタです。file の値が NULL でない場合、必要であれば path を NULL として指定できます。path が指定され、ライブラリのビルド時に `NO_WOLFSSL_DIR` が定義されていない場合、wolfSSL は指定されたディレクトリにあるすべての CA 証明書をロードします。この関数は、指定されたフラグに基づいてディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダー “—BEGIN CERTIFICATE—” を持つ PEM 形式の `CERT_TYPE` ファイルを想定しています。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **file** PEM 形式の CA 証明書を含むファイル名へのポインタ。
- **path** PEM 形式の証明書を読み込むディレクトリ名へのポインタ。
- **flags** 指定可能なマスク値: `WOLFSSL_LOAD_FLAG_IGNORE_ERR`、`WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY`、`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 成功時。
- SSL_FAILURE ctx が NULL の場合、または file と path の両方が NULL の場合に返されます。少なくとも 1 つの証明書が正常にロードされたが、1 つ以上が失敗した場合もこれが返されます。理由についてはエラースタックを確認してください。
- SSL_BAD_FILETYPE ファイルが間違ったフォーマットの場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファより大きい場合に返されます。
- BAD_PATH_ERROR path を開こうとしたときに opendir() が失敗した場合に返されます。

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) {
    // CA 証明書の読み込みエラー
}
...
```

C.52.2.32 function wolfSSL_get_system_CA_dirs

```
const char ** wolfSSL_get_system_CA_dirs(
    word32 * num
)
```

この関数は、wolfSSL_CTX_load_system_CA_certs が呼び出されたときに wolfSSL がシステム CA 証明書を検索するディレクトリを表す文字列の配列へのポインタを返します。証明書をアクセス可能なシステムディレクトリに保存しないシステム (Apple プラットフォームなど) では、この関数は常に NULL を返します。

Parameters:

- **num** 文字列配列の長さが格納される word32 へのポインタ。

See:

- wolfSSL_CTX_load_system_CA_certs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- 成功時は有効なポインタ。
- 失敗時は NULL ポインタ。

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.52.2.33 function wolfSSL_CTX_load_system_CA_certs

```
int wolfSSL_CTX_load_system_CA_certs(
    WOLFSSL_CTX * ctx
)
```

ほとんどのプラットフォーム (Linux および Windows を含む) において、この関数は OS 依存の CA 証明書ストアから WOLFSSL_CTX へ CA 証明書を読み込もうと試みます。読み込まれた証明書は信頼されます。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。

See:

- wolfSSL_get_system_CA_dirs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- 成功時は WOLFSSL_SUCCESS。
- システム CA 証明書が読み込まれなかった場合は WOLFSSL_BAD_PATH。
- その他の失敗タイプの場合は WOLFSSL_FAILURE (例: Windows 証明書ストアが適切に閉じられなかった場合)。

Apple プラットフォーム (macOS を除く) では、証明書をシステムから取得できないため、wolfSSL 証明書マネージャに読み込むことができません。これらのプラットフォームでは、この関数は、WOLFSSL_CTX にバインドされた TLS 接続において、ユーザーによって読み込まれた証明書に対してピア証明書の信頼性を最初に認証できない場合に、ネイティブシステムの信頼 API を使用してピア証明書チェーンの信頼性を検証できるようにします。

サポートおよびテストされているプラットフォームは、Linux (Debian、Ubuntu、Gentoo、Fedora、RHEL)、Windows 10/11、Android、macOS、および iOS です。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_system_CA_certs(ctx,);
if (ret != WOLFSSL_SUCCESS) {
    // システム CA 証明書の読み込みエラー
}
...
```

C.52.2.34 function wolfSSL_CTX_trust_peer_cert

```
int wolfSSL_CTX_trust_peer_cert(
    WOLFSSL_CTX * ctx,
    const char * file,
```

```
    int type
)
```

この関数は、TLS/SSL ハンドシェイクを実行する際にピアを検証するために使用する証明書を読み込みます。ハンドシェイク中に送信されるピア証明書は、利用可能な場合は SKID を使用し、署名を使用して比較されます。これら 2 つが一致しない場合は、読み込まれた CA が使用されます。この機能はマクロ WOLFSSL_TRUST_PEER_CERT を定義することで有効になります。適切な使用方法については、examples を参照してください。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。
- **file** 証明書を含むファイル名へのポインタ。
- **type** 読み込まれる証明書のタイプ、すなわち SSL_FILETYPE_ASN1 または 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:

- 成功時は SSL_SUCCESS。
- ctx が NULL の場合、または file と type の両方が無効な場合は SSL_FAILURE が返されます。
- ファイルの形式が間違っている場合は SSL_BAD_FILETYPE が返されます。
- ファイルが存在しない、読み取れない、または破損している場合は SSL_BAD_FILE が返されます。
- メモリ不足状態が発生した場合は MEMORY_E が返されます。
- ファイルに対する Base16 デコードが失敗した場合は ASN_INPUT_E が返されます。

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) {
    // 信頼されたピア証明書の読み込みエラー
}
...
```

C.52.2.35 function wolfSSL_CTX_use_certificate_chain_file

```
int wolfSSL_CTX_use_certificate_chain_file(
    WOLFSSL_CTX * ctx,
    const char * file
)
```

この関数は、証明書チェーンを SSL コンテキスト (WOLFSSL_CTX) に読み込みます。証明書チェーンを含むファイルは file 引数によって提供され、PEM 形式の証明書を含む必要があります。この関数は、

MAX_CHAIN_DEPTH(デフォルト = 9、internal.h で定義) 個までの証明書と、サブジェクト証明書を処理します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **file** wolfSSL SSL コンテキストに読み込まれる証明書チェーンを含むファイルの名前へのポインタ。証明書は PEM 形式である必要があります。

See:

- `wolfSSL_CTX_use_certificate_file`
- `wolfSSL_CTX_use_certificate_buffer`
- `wolfSSL_use_certificate_file`
- `wolfSSL_use_certificate_buffer`

Return:

- 成功時は SSL_SUCCESS。
- 関数呼び出しが失敗した場合は SSL_FAILURE。考えられる原因には、ファイルの形式が間違っている、または「format」引数を使用して間違った形式が指定されている、ファイルが存在しない、読み取れない、または破損している、メモリ不足状態が発生した、などがあります。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_chain_file(ctx, "./cert-chain.pem");
if (ret != SSL_SUCCESS) {
    // 証明書ファイルの読み込みエラー
}
...
```

C.52.2.36 function wolfSSL_CTX_use_RSAPrivateKey_file

```
int wolfSSL_CTX_use_RSAPrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

この関数は、SSL 接続で使用されるプライベート RSA 鍵を SSL コンテキスト (WOLFSSL_CTX) に読み込みます。この関数は、wolfSSL が OpenSSL 互換レイヤーを有効にしてコンパイルされた場合 (-enable-opensslExtra、#define OPENSSL_EXTRA) にのみ利用可能であり、より一般的に使用される wolfSSL_CTX_use_PrivateKey_file() 関数と同一です。file 引数には、format 引数で指定された形式の RSA プライベート鍵ファイルへのポインタが含まれます。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **file** wolfSSL SSL コンテキストに読み込まれる RSA プライベート鍵を含むファイルの名前へのポインタ。形式は format 引数で指定されます。
- **format** file 引数で指定された RSA プライベート鍵のエンコーディングタイプ。指定可能な値には、SSL_FILETYPE_PEM および SSL_FILETYPE_ASN1 が含まれます。

See:

- `wolfSSL_CTX_use_PrivateKey_buffer`
- `wolfSSL_CTX_use_PrivateKey_file`
- `wolfSSL_use_RSAPrivateKey_file`

- `wolfSSL_use_PrivateKey_buffer`
- `wolfSSL_use_PrivateKey_file`

Return:

- 成功時は `SSL_SUCCESS`。
- 関数呼び出しが失敗した場合は `SSL_FAILURE`。考えられる原因には、入力鍵ファイルの形式が間違っている、または「format」引数を使用して間違った形式が指定されている、ファイルが存在しない、読み取れない、または破損している、メモリ不足状態が発生した、などがあります。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_RSAPrivateKey_file(ctx, "./server-key.pem",
                                         SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // プライベート鍵ファイルの読み込みエラー
}
...
```

C.52.2.37 function wolfSSL_get_verify_depth

```
long wolfSSL_get_verify_depth(
    WOLFSSL * ssl
)
```

この関数は、有効なセッション (つまり、非 NULL のセッションオブジェクト (ssl) が存在する場合) に対して許可される最大チェーン深度を返します。デフォルトは 9 です。

Parameters:

- `ssl` `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_CTX_get_verify_depth`

Return:

- WOLFSSL 構造体が NULL でない場合は `MAX_CHAIN_DEPTH` が返されます。デフォルトでは値は 9 です。
- WOLFSSL 構造体が NULL の場合は `BAD_FUNC_ARG` が返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
long sslDep = wolfSSL_get_verify_depth(ssl);

if(sslDep > EXPECTED){
    // 検証された深度は期待値よりも大きい
} else {
    // 検証された深度は期待値以下
}
```

C.52.2.38 function wolfSSL_CTX_get_verify_depth

```
long wolfSSL_CTX_get_verify_depth(
    WOLFSSL_CTX * ctx
)
```

この関数は、CTX 構造体を使用して証明書チェーンの深度を取得します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- `wolfSSL_CTX_use_certificate_chain_file`
- `wolfSSL_get_verify_depth`

Return:

- CTX 構造体が NULL でない場合は MAX_CHAIN_DEPTH が返されます。最大証明書チェーンピア深度の定数表現です。
- CTX 構造体が NULL の場合は BAD_FUNC_ARG が返されます。

Example

```
WOLFSSL_METHOD method; // プロトコルメソッド
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
...
long ret = wolfSSL_CTX_get_verify_depth(ctx);

if(ret == EXPECTED){
    // 期待値を取得しました
} else {
    // 予期しない深度を処理
}
```

C.52.2.39 function wolfSSL_use_certificate_file

```
int wolfSSL_use_certificate_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

この関数は、証明書ファイルを SSL セッション (WOLFSSL 構造体) に読み込みます。証明書ファイルは file 引数によって提供されます。format 引数は、ファイルの形式タイプ (SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM) を指定します。

Parameters:

- **ssl** `wolfSSL_new()`で作成された WOLFSSL 構造体へのポインタ。
- **file** wolfSSL SSL セッションに読み込まれる証明書を含むファイルの名前へのポインタ。形式は format 引数で指定されます。
- **format** file 引数で指定された証明書のエンコーディングタイプ。指定可能な値には、SSL_FILETYPE_PEM および SSL_FILETYPE_ASN1 が含まれます。

See:

- `wolfSSL_CTX_use_certificate_buffer`
- `wolfSSL_CTX_use_certificate_file`
- `wolfSSL_use_certificate_buffer`

Return:

- 成功時は SSL_SUCCESS。
- 関数呼び出しが失敗した場合は SSL_FAILURE。考えられる原因には、ファイルの形式が間違っている、または「format」引数を使用して間違った形式が指定されている、ファイルが存在しない、読み取れない、または破損している、メモリ不足状態が発生した、ファイルに対する Base16 デコードが失敗した、などがあります。

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_certificate_file(ssl, "./client-cert.pem",
                                  SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // 証明書ファイルの読み込みエラー
}
...
```

C.52.2.40 function wolfSSL_use_PrivateKey_file

```
int wolfSSL_use_PrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

この関数は、プライベート鍵ファイルを SSL セッション (WOLFSSL 構造体) に読み込みます。鍵ファイルは file 引数によって提供されます。format 引数は、ファイルの形式タイプ (SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM) を指定します。

Parameters:

- **ssl** wolfSSL_new() で作成された WOLFSSL 構造体へのポインタ。
- **file** wolfSSL SSL セッションに読み込まれる鍵ファイルを含むファイルの名前へのポインタ。形式は format 引数で指定されます。
- **format** file 引数で指定された鍵のエンコーディングタイプ。指定可能な値には、SSL_FILETYPE_PEM および SSL_FILETYPE_ASN1 が含まれます。

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_SetDevId

Return:

- 成功時は SSL_SUCCESS。
- 関数呼び出しが失敗した場合は SSL_FAILURE。考えられる原因には、ファイルの形式が間違っている、または「format」引数を使用して間違った形式が指定されている、ファイルが存在しない、読み取れない、または破損している、メモリ不足状態が発生した、ファイルに対する Base16 デコードが失敗した、鍵ファイルが暗号化されているがパスワードが提供されていない、などがあります。

外部鍵ストアを使用していてプライベート鍵を持っていない場合は、代わりに公開鍵を提供し、署名を処理するための暗号コールバックを登録することができます。これには、暗号コールバックまたは PK コールバックのいずれかを有効にしてビルドします。暗号コールバックを有効にするには、-enable-cryptocb または WOLF_CRYPTOCB を使用し、wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_SetDevId を使用して関連する devId を設定します。

Example

```

int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_PrivateKey_file(ssl, "./server-key.pem",
                                SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // 鍵ファイルの読み込みエラー
}
...

```

C.52.2.41 function wolfSSL_use_certificate_chain_file

```

int wolfSSL_use_certificate_chain_file(
    WOLFSSL * ssl,
    const char * file
)

```

この関数は、証明書のチェーンを SSL セッション (WOLFSSL 構造体) にロードします。証明書チェーンを含むファイルは file 引数で指定され、PEM 形式の証明書を含んでいる必要があります。この関数は、MAX_CHAIN_DEPTH(デフォルト = 9、internal.h で定義) 個までの証明書と、サブジェクト証明書を処理します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ
- **file** wolfSSL SSL セッションにロードする証明書のチェーンを含むファイルの名前へのポインタ。証明書は PEM 形式でなければなりません。

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE 関数呼び出しが失敗した場合、考えられる原因は次のとおりです: ファイルの形式が間違っている、または "format" 引数で与えられた形式が間違っている、ファイルが存在しない、読み取れない、または破損している、メモリ不足の状態が発生した

Example

```

int ret = 0;
WOLFSSL* ctx;
...
ret = wolfSSL_use_certificate_chain_file(ssl, "./cert-chain.pem");
if (ret != SSL_SUCCESS) {
    // 証明書ファイルのロードエラー
}
...

```

C.52.2.42 function wolfSSL_use_RSAPrivateKey_file

```

int wolfSSL_use_RSAPrivateKey_file(
    WOLFSSL * ssl,
    const char * file,

```

```
    int format
)
```

この関数は、SSL 接続で使用されるプライベート RSA 鍵を SSL セッション (WOLFSSL 構造体) にロードします。この関数は、wolfSSL が OpenSSL 互換レイヤーを有効にしてコンパイルされている場合 (-enable-opensslExtra、#define OPENSSL_EXTRA) にのみ利用可能であり、より一般的に使用される wolfSSL_use_PrivateKey_file() 関数と同一です。file 引数は、format で指定された形式の RSA プライベート鍵ファイルへのポインタを含みます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ
- **file** wolfSSL SSL セッションにロードする RSA プライベート鍵を含むファイルの名前へのポインタ、形式は format で指定されます。
- **format** file で指定された RSA プライベート鍵のエンコーディングタイプ。指定可能な値は SSL_FILETYPE_PEM と SSL_FILETYPE_ASN1 です。

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 成功時
- SSL_FAILURE 関数呼び出しが失敗した場合、考えられる原因は次のとおりです: 入力鍵ファイルの形式が間違っている、または "format" 引数で与えられた形式が間違っている、ファイルが存在しない、読み取れない、または破損している、メモリ不足の状態が発生した

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_RSAPrivateKey_file(ssl, "./server-key.pem",
                                     SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // プライベート鍵ファイルのロードエラー
}
...
```

C.52.2.43 function wolfSSL_CTX_der_load_verify_locations

```
int wolfSSL_CTX_der_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

この関数は wolfSSL_CTX_load_verify_locations に似ていますが、DER 形式の CA ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードできます。PEM 形式の CA ファイルをロードするためにも使用できます。これらの証明書は信頼されたルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。file 引数で指定されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルである可能性があります。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイル内で提示された順序でそれらをロードします。format 引数は、証明書の形式が SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1(DER) のいずれかであることを指定します。

wolfSSL_CTX_load_verify_locations とは異なり、この関数は指定されたディレクトリパスからの CA 証明書のロードを許可しません。この関数は、wolfSSL ライブラリが WOLFSSL_DER_LOAD を定義してコンパイルされている場合にのみ利用可能です。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインタ
- **file** wolfSSL SSL コンテキストにロードする CA 証明書を含むファイルの名前へのポインタ、形式は format で指定されます。
- **format** file で指定された証明書のエンコーディングタイプ。指定可能な値は SSL_FILETYPE_PEM と SSL_FILETYPE_ASN1 です。

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE 失敗時。

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) {
    // CA 証明書のロードエラー
}
...
```

C.52.2.44 function wolfSSL_CTX_new

```
WOLFSSL_CTX * wolfSSL_CTX_new(
    WOLFSSL_METHOD *
)
```

この関数は、入力として希望する SSL/TLS プロトコルメソッドを取り、新しい SSL コンテキストを作成します。

Parameters:

- **method** SSL コンテキストに使用する希望の WOLFSSL_METHOD へのポインタ。これは、SSL/TLS/DTLS プロトコルレベルを指定するための wolfSSLvXX_XXXX_method() 関数のいずれかを使用して作成されます。

See: wolfSSL_new

Return:

- pointer 成功した場合、呼び出しは新しく作成された WOLFSSL_CTX へのポインタを返します。
- NULL 失敗時。

Example

```
WOLFSSL_CTX* ctx = 0;
WOLFSSL_METHOD* method = 0;

method = wolfSSLv3_client_method();
```

```

if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキストの作成に失敗しました
}

```

C.52.2.45 function wolfSSL_new

```

WOLFSSL * wolfSSL_new(
    WOLFSSL_CTX *
)

```

この関数は、既に作成された SSL コンテキストを入力として取り、新しい SSL セッションを作成します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された SSL コンテキストへのポインタ。

See: [wolfSSL_CTX_new](#)

Return:

- * 成功した場合、呼び出しは新しく作成された wolfSSL 構造体へのポインタを返します。
- NULL 失敗時。

Example

```

#include <wolfssl/ssl.h>

WOLFSSL*      ssl = NULL;
WOLFSSL_CTX*  ctx = 0;

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキストの作成に失敗しました
}

ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // SSL オブジェクトの作成に失敗しました
}

```

C.52.2.46 function wolfSSL_set_fd

```

int wolfSSL_set_fd(
    WOLFSSL * ssl,
    int fd
)

```

この関数は、ファイルディスクリプタ (fd) を SSL 接続の入出力機能として割り当てます。通常、これはソケットファイルディスクリプタになります。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)で作成された SSL セッションへのポインタ。
- **fd** SSL/TLS 接続で使用するファイルディスクリプタ。

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG 失敗時。

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_fd(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // SSL ファイルディスクリプタの設定に失敗しました
}
```

C.52.2.47 function wolfSSL_set_dtls_fd_connected

```
int wolfSSL_set_dtls_fd_connected(
    WOLFSSL * ssl,
    int fd
)
```

この関数は、ファイルディスクリプタ (fd) を SSL 接続の入出力機能として割り当てます。通常、これはソケットファイルディスクリプタになります。これは、ソケットが接続されていることをマークするため、DTLS 固有の API です。この fd に対する recvfrom および sendto 呼び出しは、addr および addr_len パラメータが NULL に設定されます。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **fd** SSL/TLS 接続で使用するファイルディスクリプタ。

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfDTLS_SetChGoodCb

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG 失敗時。

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
if (connect(sockfd, peer_addr, peer_addr_len) != 0) {
    // 接続エラーを処理
}
```



```
...
ret = wolfSSL_set_dtls_fd_connected(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // SSL ファイルディスクリプタの設定に失敗しました
}
```

C.52.2.48 function wolfDTLS_SetChGoodCb

```
int wolfDTLS_SetChGoodCb(
    WOLFSSL * ssl,
    ClientHelloGoodCb cb,
    void * user_ctx
)
```

正しく処理および検証された DTLS クライアント hello に対するコールバックを設定できます。クッキー交換メカニズム (DTLS 1.2 の HelloVerifyRequest またはクッキー拡張を伴う DTLS 1.3 の HelloRetryRequest) を使用する場合、このコールバックはクッキー交換が成功した後に呼び出されます。これは、1 つの WOLFSSL オブジェクトを新しい接続のリスナーとして使用し、ClientHello が検証された後 (クッキー交換を通じて、または ClientHello が正しい形式であるかをチェックするだけで) に WOLFSSL オブジェクトを分離できるようにするのに役立ちます。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** `wolfSSL_new()` で作成された SSL セッションへのポインタ。
- **fd** SSL/TLS 接続で使用するファイルディスクリプタ。

See: `wolfSSL_set_dtls_fd_connected`

Return:

- `SSL_SUCCESS` 成功時。
- `BAD_FUNC_ARG` 失敗時。

Example

```
// 接続を検証したときに呼び出されます
static int chGoodCb(WOLFSSL* ssl, void* arg)
{
    // ピアとファイルディスクリプタを設定
}

if (wolfDTLS_SetChGoodCb(ssl, chGoodCb, NULL) != WOLFSSL_SUCCESS) {
    // コールバック設定エラー
}
```

C.52.2.49 function wolfSSL_get_cipher_list

```
char * wolfSSL_get_cipher_list(
    int priority
)
```

渡された優先度レベルでの暗号の名前を取得します。

Parameters:

- **priority** 暗号の優先度レベルを表す整数。

See:

- `wolfSSL_CIPHER_get_name`
- `wolfSSL_get_current_cipher`

Return:

- string 成功
- 0 優先度が範囲外または無効です。

Example

```
printf("The cipher at 1 is %s", wolfSSL_get_cipher_list(1));
```

C.52.2.50 function `wolfSSL_get_ciphers`

```
int wolfSSL_get_ciphers(
    char * buf,
    int len
)
```

この関数は、wolfSSL で有効になっている暗号を取得します。

Parameters:

- **buf** バッファを表す char ポインタ。
- **len** バッファの長さ。

See:

- GetCipherNames
- `wolfSSL_get_cipher_list`
- ShowCiphers

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- BAD_FUNC_ARG buf パラメータが NULL または len 引数がゼロ以下の場合に返されます。
- BUFFER_E バッファが十分に大きくなくオーバーフローする場合に返されます。

Example

```
static void ShowCiphers(void){
    char* ciphers;
    int ret = wolfSSL_get_ciphers(ciphers, (int)sizeof(ciphers));

    if(ret == SSL_SUCCESS){
        printf("%s\n", ciphers);
    }
}
```

C.52.2.51 function `wolfSSL_get_cipher_name`

```
const char * wolfSSL_get_cipher_name(
    WOLFSSL * ssl
)
```

この関数は、引数を `wolfSSL_get_cipher_name_internal` に渡すことによって、DHE-RSA 形式で暗号名を取得します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_CIPHER_get_name`
- `wolfSSL_get_current_cipher`
- `wolfSSL_get_cipher_name_internal`

Return:

- string この関数は、マッチした暗号スイートの文字列表現を返します。
- NULL エラーまたは暗号が見つかりません。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
char* cipher = wolfSSL_get_cipher_name(ssl);

if(cipher == NULL){
    // 暗号スイートがマッチしませんでした
} else {
    // 暗号スイートがマッチしました
    printf("%s\n", cipher);
}
```

C.52.2.52 function wolfSSL_get_fd

```
int wolfSSL_get_fd(
    const WOLFSSL * ssl
)
```

この関数は、SSL 接続の入力機能として使用される読み取りファイル記述子 (fd) を返します。通常、これはソケットファイル記述子になります。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL セッションへのポインタ。

See:

- `wolfSSL_set_fd`
- `wolfSSL_set_read_fd`
- `wolfSSL_set_write_fd`

Return: fd 成功した場合、この関数は SSL セッションファイルディスクリプタを返します。

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_fd(ssl);
...
```

C.52.2.53 function wolfSSL_get_wfd

```
int wolfSSL_get_wfd(
    const WOLFSSL * ssl
)
```

この関数は、SSL 接続の出力機能として使用される書き込みファイルディスクリプタ (fd) を返します。通常はソケットファイルディスクリプタになります。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL セッションへのポインタ。

See:

- `wolfSSL_set_fd`
- `wolfSSL_set_read_fd`
- `wolfSSL_set_write_fd`

Return: fd 成功した場合、この関数は SSL セッションファイルディスクリプタを返します。

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_wfd(ssl);
...
```

C.52.2.54 function wolfSSL_set_using_nonblock

```
void wolfSSL_set_using_nonblock(
    WOLFSSL * ssl,
    int nonblock
)
```

この関数は、WOLFSSL オブジェクトに対して、下層の I/O がノンブロッキングであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキングソケットと共に使用する場合は、`wolfSSL_set_using_nonblock()` を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、`recvfrom` 呼び出しがブロックすることを意味すると認識できます。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL セッションへのポインタ。
- **nonblock** WOLFSSL オブジェクトのノンブロッキングフラグを設定するための値。ノンブロッキングを指定する場合は 1 を、そうでない場合は 0 を使用してください。

See:

- `wolfSSL_get_using_nonblock`
- `wolfSSL_dtls_got_timeout`
- `wolfSSL_dtls_get_current_timeout`

Return: none 戻り値はありません。

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_using_nonblock(ssl, 1);
```

C.52.2.55 function wolfSSL_get_using_nonblock

```
int wolfSSL_get_using_nonblock(
    WOLFSSL *
)
```

この関数により、アプリケーションは `wolfSSL` がノンブロッキング I/O を使用しているかどうかを判定できます。`wolfSSL` がノンブロッキング I/O を使用している場合、この関数は 1 を返し、それ以外の場合は 0 を返します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキングソケット

と共に使用する場合は、wolfSSL_set_using_nonblock() を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、recvfrom 呼び出しがブロックすることを意味すると認識できます。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。

See: wolfSSL_set_session

Return:

- 0 下層の I/O がブロッキングです。
- 1 下層の I/O がノンブロッキングです。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_get_using_nonblock(ssl);
if (ret == 1) {
    // 下層の I/O はノンブロッキング
}
...
```

C.52.2.56 function wolfSSL_write

```
int wolfSSL_write(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

この関数は、バッファ data から sz バイトを SSL 接続 ssl に書き込みます。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_write() は SSL/TLS セッションをネゴシエートします。(D)TLSv1.3 を使用していて early data 機能がコンパイルされている場合、この関数はデータ送信が可能になるまでハンドシェイクを進めます。次の wolfSSL_Connect()、wolfSSL_Accept()、wolfSSL_read() の呼び出しでハンドシェイクが完了します。wolfSSL_write() は、ブロッキングとノンブロッキングの両方の I/O で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_write() は、下層の I/O が wolfSSL_write() を継続するために必要な要求を満たせなくなった時点で返されます。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、下層の I/O の準備ができれば、wolfSSL_write() の呼び出しを繰り返す必要があります。下層の I/O がブロッキングの場合、wolfSSL_write() は、サイズ sz のバッファデータが完全に書き込まれるか、エラーが発生するまで返されません。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **data** ピアに送信されるデータバッファ。
- **sz** ピアに送信するデータ (data) のサイズ (バイト単位)。

See:

- wolfSSL_send
- wolfSSL_read
- wolfSSL_recv

Return:

- > 0 成功時に書き込まれたバイト数。

- 0 失敗時に返されます。具体的なエラーコードについては、wolfSSL_get_error() を呼び出してください。
- SSL_FATAL_ERROR エラーが発生した場合、またはノンブロッキングソケットを使用している場合に SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーを受け取り、アプリケーションが wolfSSL_write() を再度呼び出す必要がある場合に返されます。具体的なエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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() が失敗、wolfSSL_get_error() を呼び出す
}
```

C.52.2.57 function wolfSSL_read

```
int wolfSSL_read(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファ data に読み込みます。読み取られたバイトは内部受信バッファから削除されます。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_read() は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズが 16kB の SSL レコードを使用します (最大レコードサイズは、/wolfssl/internal.h 内の MAX_RECORD_SIZE 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号できるようになる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_read() の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。内部 wolfSSL 受信バッファには、まだ復号されていない追加データが待機している可能性があり、これは次の wolfSSL_read() 呼び出しで取得および復号されます。sz が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_read() は内部読み取りバッファ内で利用可能なバイトを返します。内部読み取りバッファにまだバイトがバッファリングされていない場合、wolfSSL_read() の呼び出しは次のレコードの処理をトリガします。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **data** wolfSSL_read() が読み取ったデータを格納するバッファ。
- **sz** data に読み込むバイト数。

See:

- wolfSSL_recv
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- > 0 成功時に読み取られたバイト数。

- 0 失敗時に返されます。これは、クリーンシャットダウン (close notify アラート) またはピアが接続を閉じたことが原因である可能性があります。具体的なエラーコードについては、wolfSSL_get_error() を呼び出してください。
- SSL_FATAL_ERROR エラーが発生した場合、またはノンブロッキングソケットを使用している場合に SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーを受け取り、アプリケーションが wolfSSL_read() を再度呼び出す必要がある場合に返されます。具体的なエラーコードを取得するには、wolfSSL_get_error() を使用してください。

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_read(ssl, reply, sizeof(reply));
if (input > 0) {
    // バッファ"reply" に"input" バイトが返された
}
```

wolfSSL_read() のより完全な例については、wolfSSL の例 (client、server、echoclient、
↪ echoserver) を参照してください。

C.52.2.58 function wolfSSL_peek

```
int wolfSSL_peek(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファ data にコピーします。この関数は、内部 SSL セッション受信バッファ内のデータが削除または変更されない点を除いて、wolfSSL_read() と同じです。wolfSSL_read() と同様に、必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_peek() は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズが 16kB の SSL レコードを使用します (最大レコードサイズは、/wolfssl/internal.h 内の MAX_RECORD_SIZE 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号できるようになる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_peek() の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。内部 wolfSSL 受信バッファには、まだ復号されていない追加データが待機している可能性があり、これは次の wolfSSL_peek() または wolfSSL_read() 呼び出しで取得および復号されます。sz が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_peek() は内部読み取りバッファ内で利用可能なバイトを返します。内部読み取りバッファにまだバイトがバッファリングされていない場合、wolfSSL_peek() の呼び出しは次のレコードの処理をトリガします。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **data** wolfSSL_peek() が読み取ったデータを格納するバッファ。
- **sz** data に読み込むバイト数。

See: wolfSSL_read**Return:**

- > 0 成功時に読み取られたバイト数。
- 0 失敗時に返されます。これは、クリーンシャットダウン (close notify アラート) またはピアが接続を閉じたことが原因である可能性があります。具体的なエラーコードについては、wolfSSL_get_error() を呼び出してください。

- SSL_FATAL_ERROR エラーが発生した場合、またはノンブロッキングソケットを使用している場合に SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーを受け取り、アプリケーションが wolfSSL_peek() を再度呼び出す必要がある場合に返されます。具体的なエラーコードを取得するには、wolfSSL_get_error() を使用してください。

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_peek(ssl, reply, sizeof(reply));
if (input > 0) {
    // バッファ"reply" に"input" バイトが返された
}
```

C.52.2.59 function wolfSSL_accept

```
int wolfSSL_accept(
    WOLFSSL * ssl
)
```

この関数はサーバ側で呼び出され、SSL クライアントが SSL/TLS ハンドシェイクを開始するのを待機します。この関数が呼び出されるとき、下層の通信チャンネルはすでに設定されています。wolfSSL_accept() は、ブロッキングとノンブロッキングの両方の I/O で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_accept() は、下層の I/O が wolfSSL_accept がハンドシェイクを継続するために必要な要求を満たせなくなった時点で返されます。この場合、wolfSSL_get_error() を呼び出すと、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、データの読み取りが可能になったら wolfSSL_accept() の呼び出しを繰り返す必要があり、wolfSSL は中断した箇所から再開します。ノンブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。下層の I/O がブロッキングの場合、wolfSSL_accept() は、ハンドシェイクが完了するか、エラーが発生するまで返されません。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_connect

Return:

- SSL_SUCCESS 成功時。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出してください。

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.52.2.60 function wolfDTLS_accept_stateless

```
int wolfDTLS_accept_stateless(
    WOLFSSL * ssl
)
```

この関数はサーバ側で呼び出され、SSL クライアントが DTLS ハンドシェイクを開始するのをステートレスに待機します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_accept`
- `wolfSSL_get_error`
- `wolfSSL_connect`

Return:

- WOLFSSL_SUCCESS 有効なクッキーを含む ClientHello が受信されました。wolfSSL_accept() で接続を続行できます。
- WOLFSSL_FAILURE I/O 層が WANT_READ を返しました。これは、読み取るデータがなくノンブロッキングソケットを使用しているか、クッキーリクエストを送信して応答を待っているためです。ユーザは、I/O 層でデータが利用可能になった後、wolfDTLS_accept_stateless を再度呼び出す必要があります。
- WOLFSSL_FATAL_ERROR 致命的なエラーが発生しました。ssl オブジェクトを解放して再割り当てしてから続行する必要があります。

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
...
do {
    ret = wolfDTLS_accept_stateless(ssl);
    if (ret == WOLFSSL_FATAL_ERROR)
        // wolfSSL_free() と wolfSSL_new() で ssl オブジェクトを再割り当て
} 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.52.2.61 function wolfSSL_CTX_free

```
void wolfSSL_CTX_free(
    WOLFSSL_CTX * ctx
)
```

この関数は、割り当てられた WOLFSSL_CTX オブジェクトを解放します。この関数は CTX 参照カウントをデクリメントし、参照カウントが 0 になった場合にのみコンテキストを解放します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された SSL コンテキストへのポインタ。

See:

- `wolfSSL_CTX_new`
- `wolfSSL_new`
- `wolfSSL_free`

Return: none 戻り値はありません。

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_free(ctx);
```

C.52.2.62 function `wolfSSL_free`

```
void wolfSSL_free(
    WOLFSSL * ssl
)
```

この関数は、割り当てられた `wolfSSL` オブジェクトを解放します。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL オブジェクトへのポインタ。

See:

- `wolfSSL_CTX_new`
- `wolfSSL_new`
- `wolfSSL_CTX_free`

Return: none 戻り値はありません。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL* ssl = 0;
...
wolfSSL_free(ssl);
```

C.52.2.63 function `wolfSSL_shutdown`

```
int wolfSSL_shutdown(
    WOLFSSL * ssl
)
```

この関数は、SSL セッション `ssl` を使用してアクティブな SSL/TLS 接続をシャットダウンします。この関数は、ピアに「close notify」アラートを送信しようと試みます。呼び出し側のアプリケーションは、ピアからの応答として「close notify」アラートが送信されるのを待つか、`wolfSSL_shutdown()` を直接呼び出した後に基盤となる接続をシャットダウンするか (リソースを節約するため) を選択できます。TLS 仕様では、どちらのオプションも許可されています。基盤となる接続を将来再び使用する場合は、ピア間の同期を維持するために、完全な双方向シャットダウン手順を実行する必要があります。`wolfSSL_shutdown()` は、ブロッキング I/O とノンブロッキング I/O の両方で動作します。基盤となる I/O がノンブロッキングの場合、基盤となる I/O が `wolfSSL_shutdown()` の継続に必要な要求を満たせなかった場合、`wolfSSL_shutdown()` はエラーを返します。この場合、`wolfSSL_get_error()` を呼び出すと、`SSL_ERROR_WANT_READ` または `SSL_ERROR_WANT_WRITE` のいずれかが返されます。呼び出し側プロセスは、基盤となる I/O の準備ができたときに、`wolfSSL_shutdown()` の呼び出しを繰り返す必要があります。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL セッションへのポインタ。

See:

- `wolfSSL_free`
- `wolfSSL_CTX_free`

Return:

- `SSL_SUCCESS` が成功時に返されます。
- `SSL_SHUTDOWN_NOT_DONE` は、シャットダウンが完了していない場合に返され、関数を再度呼び出す必要があります。
- `SSL_FATAL_ERROR` は、失敗時に返されます。より具体的なエラーコードについては、`wolfSSL_get_error()` を呼び出してください。

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_shutdown(ssl);
if (ret != 0) {
    // SSL 接続のシャットダウンに失敗しました
}
```

C.52.2.64 function wolfSSL_send

```
int wolfSSL_send(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int flags
)
```

この関数は、指定されたフラグを使用して、バッファ data から sz バイトを SSL 接続 ssl に書き込みます。必要に応じて、`wolfSSL_connect()` または `wolfSSL_accept()` によってハンドシェイクがまだ実行されていない場合、`wolfSSL_send()` は SSL/TLS セッションをネゴシエートします。`wolfSSL_send()` は、ブロッキング I/O とノンブロッキング I/O の両方で動作します。基盤となる I/O がノンブロッキングの場合、基盤となる I/O が `wolfSSL_send` の継続に必要な要求を満たせなかった場合、`wolfSSL_send()` は戻ります。この場合、`wolfSSL_get_error()` を呼び出すと、`SSL_ERROR_WANT_READ` または `SSL_ERROR_WANT_WRITE` のいずれかが返されます。呼び出し側プロセスは、基盤となる I/O の準備ができたときに、`wolfSSL_send()` の呼び出しを繰り返す必要があります。基盤となる I/O がブロッキングの場合、`wolfSSL_send()` は、サイズ sz のバッファ data が完全に書き込まれるか、エラーが発生するまで戻りません。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL セッションへのポインタ。
- **data** ピアに送信するデータバッファ。
- **sz** ピアに送信するデータのサイズ (バイト単位)。
- **flags** 基盤となる送信操作に使用する送信フラグ。

See:

- `wolfSSL_write`
- `wolfSSL_read`
- `wolfSSL_recv`

Return:

- > 0 は、成功時に書き込まれたバイト数です。
- 0 は、失敗時に返されます。具体的なエラーコードについては、wolfSSL_get_error() を呼び出してください。
- SSL_FATAL_ERROR は、エラーが発生した場合、またはノンブロッキングソケットを使用している場合に SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが発生し、アプリケーションが wolfSSL_send() を再度呼び出す必要がある場合に返されます。具体的なエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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() が失敗しました
}
```

C.52.2.65 function wolfSSL_recv

```
int wolfSSL_recv(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int flags
)
```

この関数は、指定されたフラグを使用して、SSL セッション(ssl) の内部読み取りバッファから sz バイトをバッファ data に読み込みます。読み取られたバイトは、内部受信バッファから削除されます。この関数は、基盤となる読み取り操作の recv フラグをアプリケーションが設定できる点を除いて、wolfSSL_read() と同じです。必要に応じて、wolfSSL_connect() または wolfSSL_accept() によってハンドシェイクがまだ実行されていない場合、wolfSSL_recv() は SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズ 16kB の SSL レコードを使用します (最大レコードサイズは、wolfssl/internal.h の MAX_RECORD_SIZE 定義で制御できます)。そのため、wolfSSL は、レコードを処理および復号する前に、内部で SSL レコード全体を読み取る必要があります。このため、wolfSSL_recv() の呼び出しは、呼び出し時に復号された最大バッファサイズのみを返すことができます。まだ復号されていない追加データが内部 wolfSSL 受信バッファで待機している可能性があり、次の wolfSSL_recv() 呼び出しで取得および復号されます。sz が内部読み取りバッファのバイト数より大きい場合、SSL_recv() は内部読み取りバッファで利用可能なバイトを返します。内部読み取りバッファにまだバッファリングされているバイトがない場合、wolfSSL_recv() の呼び出しは次のレコードの処理をトリガーします。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **data** wolfSSL_recv() が読み取ったデータを配置するバッファ。
- **sz** data に読み込むバイト数。
- **flags** 基盤となる受信操作に使用する受信フラグ。

See:

- wolfSSL_read
- wolfSSL_write
- wolfSSL_peek

- `wolfSSL_pending`

Return:

- `> 0` は、成功時に読み取られたバイト数です。
- `0` は、失敗時に返されます。これは、クリーンな (close notify アラート) シャットダウンによって引き起こされる場合もあれば、単にピアが接続を閉じた場合もあります。具体的なエラーコードについては、`wolfSSL_get_error()` を呼び出してください。
- `SSL_FATAL_ERROR` は、エラーが発生した場合、またはノンブロッキングソケットを使用している場合に `SSL_ERROR_WANT_READ` または `SSL_ERROR_WANT_WRITE` エラーが発生し、アプリケーションが `wolfSSL_recv()` を再度呼び出す必要がある場合に返されます。具体的なエラーコードを取得するには、`wolfSSL_get_error()` を使用してください。

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
int flags = ... ;
...

input = wolfSSL_recv(ssl, reply, sizeof(reply), flags);
if (input > 0) {
    // バッファ"reply" に"input" バイトが返されました
}
```

C.52.2.66 function wolfSSL_get_error

```
int wolfSSL_get_error(
    WOLFSSL * ssl,
    int ret
)
```

この関数は、前回の API 関数呼び出し (`wolfSSL_connect`、`wolfSSL_accept`、`wolfSSL_read`、`wolfSSL_write` 等) がエラーリターンコード (`SSL_FAILURE`) になった理由を説明する一意のエラーコードを返します。前回の関数の戻り値は、`ret` を通じて `wolfSSL_get_error` に渡されます。`wolfSSL_get_error` が呼び出されて一意のエラーコードを返した後、`wolfSSL_ERR_error_string()` を呼び出して、人間が読めるエラー文字列を取得できます。詳細については、`wolfSSL_ERR_error_string()` を参照してください。

Parameters:

- **ssl** `wolfSSL_new()` で作成された SSL オブジェクトへのポインタ。
- **ret** エラーリターンコードになった前回の関数の戻り値。

See:

- `wolfSSL_ERR_error_string`
- `wolfSSL_ERR_error_string_n`
- `wolfSSL_ERR_print_errors_fp`
- `wolfSSL_load_error_strings`

Return:

- 正常に完了した場合、この関数は前回の API 関数が失敗した理由を説明する一意のエラーコードを返します。
- `SSL_ERROR_NONE` は、`ret > 0` の場合に返されます。`ret <= 0` の場合、前回の API がエラーコードを返したように見えても実際にはエラーが発生していなかった場合に、この値が返されることがあります。例えば、`sz` パラメータがゼロの `wolfSSL_read()` を呼び出した場合です。`wolfSSL_read()` からの `0` の戻り値は通常エラーを示しますが、この場合はエラーは発生していません。その後 `wolfSSL_get_error()` を呼び出すと、`SSL_ERROR_NONE` が返されます。

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.52.2.67 function wolfSSL_get_alert_history

```
int wolfSSL_get_alert_history(
    WOLFSSL * ssl,
    WOLFSSL_ALERT_HISTORY * h
)
```

この関数はアラート履歴を取得します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **h** WOLFSSL 構造体の alert_history メンバーの値を保持する WOLFSSL_ALERT_HISTORY 構造体へのポインタ。

See: [wolfSSL_get_error](#)

Return: SSL_SUCCESS は、関数が正常に完了したときに返されます。アラート履歴があってもなくても、いずれの場合も戻り値は SSL_SUCCESS です。

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 に ssl->alert_history の内容のコピーが格納されました
```

C.52.2.68 function wolfSSL_set_session

```
int wolfSSL_set_session(
    WOLFSSL * ssl,
    WOLFSSL_SESSION * session
)
```

この関数は、SSL オブジェクト ssl が SSL/TLS 接続を確立するために使用されるときに使用されるセッションを設定します。セッション再開の場合、セッションオブジェクトで wolfSSL_shutdown() を呼び出す前に、アプリケーションは wolfSSL_get1_session() を呼び出してオブジェクトからセッション ID を保存する必要があります。これはセッションへのポインタを返します。後で、アプリケーションは新しい WOLFSSL オブジェクトを作成し、wolfSSL_set_session() を使用して保存されたセッションを割り当てる必要があります。この時点で、アプリケーションは wolfSSL_connect() を呼び出すことができ、wolfSSL はセッションの再開を試みます。wolfSSL サーバーコードは、デフォルトでセッション再開を許可します。wolfSSL_get1_session() によって返されたオブジェクトは、アプリケーションが使用を終えた後、wolfSSL_SESSION_free() を呼び出して解放する必要があります。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)で作成された SSL オブジェクトへのポインタ。
- **session** ssl のセッションを設定するために使用される WOLFSSL_SESSION へのポインタ。

See: [wolfSSL_get1_session](#)

Return:

- SSL_SUCCESS は、セッションの設定に成功した場合に返されます。
- SSL_FAILURE は、失敗時に返されます。これは、セッションキャッシュが無効になっている場合、またはセッションがタイムアウトした場合に発生する可能性があります。
- OPENSSL_EXTRA と WOLFSSL_ERROR_CODE_OPENSSL が定義されている場合、セッションがタイムアウトしても SSL_SUCCESS が返されます。

Example

```
int ret;
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get1_session(ssl);
if (session == NULL) {
    // ssl オブジェクトからセッションオブジェクトの取得に失敗しました
}
...
ret = wolfSSL_set_session(ssl, session);
if (ret != SSL_SUCCESS) {
    // SSL セッションの設定に失敗しました
}
wolfSSL_SESSION_free(session);
...
```

C.52.2.69 function wolfSSL_get_session

```
WOLFSSL_SESSION * wolfSSL_get_session(
    WOLFSSL * ssl
)
```

NO_SESSION_CACHE_REF が定義されている場合、この関数は ssl で使用されている現在のセッション (WOLFSSL_SESSION) へのポインタを返します。この関数は、WOLFSSL_SESSION オブジェクトへの非永続的なポインタを返します。返されたポインタは、wolfSSL_free が呼び出されたときに解放されます。この呼び出しは、現在のセッションを検査または変更するためにのみ使用する必要があります。セッション再開には、wolfSSL_get1_session() を使用することをお勧めします。後方互換性のため、NO_SESSION_CACHE_REF が定義されていない場合、この関数はローカルキャッシュに格納されている永続的なセッションオブジェクトポインタを返します。キャッシュサイズは有限であり、アプリケーションが wolfSSL_set_session() を呼び出すまでに、別の ssl 接続によってセッションオブジェクトが上書きされるリスクがあります。アプリケーションで NO_SESSION_CACHE_REF を定義し、セッション再開に wolfSSL_get1_session() を使用することをお勧めします。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) で作成された SSL セッションへのポインタ。

See:

- [wolfSSL_get1_session](#)
- [wolfSSL_set_session](#)

Return:

- pointer 呼び出しが成功した場合、現在の SSL セッションオブジェクトへのポインタを返します。
- NULL は、ssl が NULL の場合、SSL セッションキャッシュが無効になっている場合、wolfSSL がセッション ID を利用できない場合、またはミューテックス関数が失敗した場合に返されます。

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get_session(ssl);
if (session == NULL) {
    // セッションポインタの取得に失敗しました
}
...
```

C.52.2.70 function wolfSSL_flush_sessions

```
void wolfSSL_flush_sessions(
    WOLFSSL_CTX * ctx,
    long tm
)
```

この関数は、期限切れのセッションをセッションキャッシュからフラッシュします。時刻 tm は、時刻比較に使用されます。wolfSSL は現在セッションに静的テーブルを使用しているため、フラッシュは不要です。そのため、この関数は現在スタブにすぎません。この関数は、wolfSSL が OpenSSL 互換レイヤーでコンパイルされている場合、OpenSSL 互換性 (SSL_flush_sessions) を提供します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **tm** セッション有効期限比較に使用される時刻。

See:

- `wolfSSL_get1_session`
- `wolfSSL_set_session`

Return: none 戻り値はありません。

Example

```
WOLFSSL_CTX* ssl;
...
wolfSSL_flush_sessions(ctx, time(0));
```

C.52.2.71 function wolfSSL_SetServerID

```
int wolfSSL_SetServerID(
    WOLFSSL * ssl,
    const unsigned char * id,
    int len,
    int newSession
)
```

この関数は、クライアントセッションをサーバー ID に関連付けます。newSession フラグがオンの場合、既存のセッションは再利用されません。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **id** WOLFSSL_SESSION 構造体の serverID メンバーにコピーされる定数バイトポインタ。
- **len** セッション id パラメータの長さを表す int 型。
- **newSession** セッションを再利用するかどうかを示すフラグを表す int 型。

See: [wolfSSL_set_session](#)

Return:

- SSL_SUCCESS は、関数がエラーなく実行された場合に返されます。
- BAD_FUNC_ARG は、WOLFSSL 構造体または id パラメータが NULL の場合、または len が 0 より大きくない場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol );
WOLFSSL* ssl = WOLFSSL_new(ctx);
const byte id[MAX_SIZE]; // または動的にスペースを作成
int len = 0; // 長さを初期化
int newSession = 0; // 許可するフラグ
...
int ret = wolfSSL_SetServerID(ssl, id, len, newSession);

if (ret == WOLFSSL_SUCCESS) {
    // ID が正常に設定されました
}
```

C.52.2.72 function wolfSSL_GetSessionIndex

```
int wolfSSL_GetSessionIndex(
    WOLFSSL * ssl
)
```

この関数は、WOLFSSL 構造体のセッションインデックスを取得します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See: [wolfSSL_GetSessionAtIndex](#)

Return: int この関数は、WOLFSSL 構造体内の sessionIndex を表す int 型を返します。

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)){
    // インデックス番号が範囲外であり、何かが正しくありません。
}
```

C.52.2.73 function wolfSSL_GetSessionAtIndex

```
int wolfSSL_GetSessionAtIndex(
    int index,
    WOLFSSL_SESSION * session
)
```

この関数はセッションキャッシュの指定されたインデックスにあるセッションを取得し、メモリにコピーします。WOLFSSL_SESSION 構造体はセッション情報を保持します。

Parameters:

- **index** セッションインデックスを表す int 型。
- **session** WOLFSSL_SESSION 構造体へのポインタ。

See:

- UnLockMutex
- LockMutex
- [wolfSSL_GetSessionIndex](#)

Return:

- SSL_SUCCESS 関数が正常に実行され、エラーが発生しなかった場合に返されます。
- BAD_MUTEX_E mutex のアンロックまたはロックエラーがあった場合に返されます。
- SSL_FAILURE 関数が正常に実行されなかった場合に返されます。

Example

```
int idx; // セッションを特定するインデックス。
WOLFSSL_SESSION* session; // コピー先のバッファ。
...
if(wolfSSL_GetSessionAtIndex(idx, session) != SSL_SUCCESS){
    // 失敗ケース。
}
```

C.52.2.74 function wolfSSL_SESSION_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_SESSION_get_peer_chain(
    WOLFSSL_SESSION * session
)
```

WOLFSSL_SESSION 構造体からピア証明書チェーンを返します。

Parameters:

- **session** WOLFSSL_SESSION 構造体へのポインタ。

See:

- [wolfSSL_GetSessionAtIndex](#)
- [wolfSSL_GetSessionIndex](#)
- AddSession

Return: pointer ピア証明書チェーンを含む WOLFSSL_X509_CHAIN 構造体へのポインタ。

Example

```
WOLFSSL_SESSION* session;
WOLFSSL_X509_CHAIN* chain;
...
chain = wolfSSL_SESSION_get_peer_chain(session);
if(!chain){
    // チェーンがありませんでした。失敗ケース。
}
```

C.52.2.75 function wolfSSL_CTX_set_verify

```
void wolfSSL_CTX_set_verify(
    WOLFSSL_CTX * ctx,
    int mode,
    VerifyCallback verify_callback
)
```

この関数はリモートピアの検証方法を設定し、また検証コールバックを SSL コンテキストに登録できるようにします。検証コールバックは検証失敗が発生した場合にのみ呼び出されます。検証コールバックが不要な場合は、verify_callback に NULL ポインタを使用できます。ピア証明書の検証モードは論理 OR 演算されたフラグのリストです。可能なフラグ値は以下の通りです。SSL_VERIFY_NONE クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常通り続行されます。サーバーモード：サーバーはクライアントに証明書要求を送信しません。したがって、クライアント検証は有効になりません。SSL_VERIFY_PEER クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトで有効になっているため、このオプションを使用しても効果はありません。サーバーモード：サーバーはクライアントに証明書要求を送信し、受信したクライアント証明書を検証します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：クライアントが要求されたときに証明書の送信に失敗した場合（SSL サーバーで SSL_VERIFY_PEER を使用している場合）、サーバー側で検証が失敗します。SSL_VERIFY_FAIL_EXCEPT_PSK クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じですが、PSK 接続の場合を除きます。PSK 接続が行われている場合、接続はピア証明書なしで続行されます。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。
- **mode** ピアの証明書の検証モードを示すフラグ。
- **verify_callback** 検証が失敗したときに呼び出されるコールバック。コールバックが不要な場合は、verify_callback に NULL ポインタを使用できます。

See: wolfSSL_set_verify

Return: none 返り値はありません。

Example

```
WOLFSSL_CTX*   ctx      = 0;
...
wolfSSL_CTX_set_verify(ctx, (WOLFSSL_VERIFY_PEER |
                             WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT), NULL);
```

C.52.2.76 function wolfSSL_set_verify

```
void wolfSSL_set_verify(
    WOLFSSL * ssl,
    int mode,
    VerifyCallback verify_callback
)
```

この関数はリモートピアの検証方法を設定し、また検証コールバックを SSL セッションに登録できるようにします。検証コールバックは検証失敗が発生した場合にのみ呼び出されます。検証コールバックが不要な場合は、verify_callback に NULL ポインタを使用できます。ピア証明書の検証モードは論理 OR 演算されたフラグのリストです。可能なフラグ値は以下の通りです。SSL_VERIFY_NONE クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常通り続行されます。サーバーモード：サーバーはクライアントに証明書要求を送信しません。したがって、クライアント検証は有効になりません。SSL_VERIFY_PEER クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトで有効になっているため、このオプションを使用しても効果はありません。サーバーモード：サーバーはクライアントに証明書要求を送信し、受信したクライアント証明書を検証します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：クライアントが要求されたときに証明書の送信に失敗した場合（SSL サーバーで SSL_VERIFY_PEER を使用している場合）、サーバー側で検証が失敗します。SSL_VERIFY_FAIL_EXCEPT_PSK クライアントモード：クライアント側で使用しても効果はありません。サーバーモード：検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じですが、PSK 接続の場合を除きます。PSK 接続が行われている場合、接続はピア証明書なしで続行されます。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL セッションへのポインタ。
- **mode** ピアの証明書の検証モードを示すフラグ。
- **verify_callback** 検証が失敗したときに呼び出されるコールバック。コールバックが不要な場合は、`verify_callback` に NULL ポインタを使用できます。

See: `wolfSSL_CTX_set_verify`**Return:** none 返り値はありません。*Example*

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_verify(ssl, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, 0);
```

C.52.2.77 function wolfSSL_SetCertCbCtx

```
void wolfSSL_SetCertCbCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

この関数は検証コールバック用のユーザー CTX オブジェクト情報を格納します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **ctx** WOLFSSL 構造体の `verifyCbCtx` メンバの値に設定される void ポインタ。

See:

- `wolfSSL_CTX_save_cert_cache`
- `wolfSSL_CTX_restore_cert_cache`
- `wolfSSL_CTX_set_verify`

Return: none 返り値はありません。*Example*

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
(void*)ctx;
...
if(ssl != NULL){
    wolfSSL_SetCertCbCtx(ssl, ctx);
} else {
    // エラーケース、SSL が適切に初期化されていません。
}
```

C.52.2.78 function wolfSSL_CTX_SetCertCbCtx

```
void wolfSSL_CTX_SetCertCbCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

この関数は検証コールバック用のユーザー CTX オブジェクト情報を格納します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ。
- **userCtx** WOLFSSL_CTX 構造体の verifyCbCtx メンバの値を設定するために使用される void ポインタ。

See:

- [wolfSSL_CTX_save_cert_cache](#)
- [wolfSSL_CTX_restore_cert_cache](#)
- [wolfSSL_CTX_set_verify](#)

Return: none 返り値はありません。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
void* userCtx = NULL; // ユーザー定義のコンテキストを割り当て
...
if(ctx != NULL){
    wolfSSL_SetCertCbCtx(ctx, userCtx);
} else {
    // エラーケース、SSL が適切に初期化されていません。
}
```

C.52.2.79 function wolfSSL_pending

```
int wolfSSL_pending(
    WOLFSSL * ssl
)
```

この関数は SSL オブジェクト内でバッファリングされ、wolfSSL_read() によって読み取り可能な利用可能なバイト数を返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)で作成された SSL セッションへのポインタ。

See:

- [wolfSSL_recv](#)
- [wolfSSL_read](#)
- [wolfSSL_peek](#)

Return: int この関数は保留中のバイト数を返します。

Example

```
int pending = 0;
WOLFSSL* ssl = 0;
...

pending = wolfSSL_pending(ssl);
printf(" バッファリングされ読み取り可能な%dバイトがあります", pending);
```

C.52.2.80 function wolfSSL_load_error_strings

```
void wolfSSL_load_error_strings(
    void
)
```

この関数は OpenSSL 互換性 (SSL_load_error_string) のためのものであり、何も動作を行いません。

Parameters:

- **none** パラメータはありません。

See:

- [wolfSSL_get_error](#)
- [wolfSSL_ERR_error_string](#)
- [wolfSSL_ERR_error_string_n](#)
- [wolfSSL_ERR_print_errors_fp](#)
- [wolfSSL_load_error_strings](#)

Return: none 返り値はありません。

Example

```
wolfSSL_load_error_strings();
```

C.52.2.81 function wolfSSL_library_init

```
int wolfSSL_library_init(  
    void  
)
```

この関数は wolfSSL_CTX_new() 内で内部的に呼び出されます。この関数は wolfSSL_Init() のラッパーであり、wolfSSL が OpenSSL 互換性レイヤーでコンパイルされた場合の OpenSSL 互換性 (SSL_library_init) のために存在します。wolfSSL_Init() は、より一般的に使用される wolfSSL 初期化関数です。

Parameters:

- **none** パラメータはありません。

See:

- [wolfSSL_Init](#)
- [wolfSSL_Cleanup](#)

Return:

- SSL_SUCCESS 呼び出しが成功した場合に返されます。
- SSL_FATAL_ERROR 失敗時に返されます。

Example

```
int ret = 0;  
ret = wolfSSL_library_init();  
if (ret != SSL_SUCCESS) {  
    // wolfSSL の初期化に失敗しました  
}  
...
```

C.52.2.82 function wolfSSL_SetDevId

```
int wolfSSL_SetDevId(  
    WOLFSSL * ssl,  
    int devId  
)
```

この関数は WOLFSSL セッションレベルで Device Id を設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) で作成された SSL オブジェクトへのポインタ。
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定します。

See:

- [wolfSSL_CTX_SetDevId](#)
- [wolfSSL_CTX_GetDevId](#)

Return:

- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG ssl が NULL の場合。

Example

```
WOLFSSL* ssl;  
int DevId = -2;  
  
wolfSSL_SetDevId(ssl, devId);
```

C.52.2.83 function wolfSSL_CTX_SetDevId

```
int wolfSSL_CTX_SetDevId(  
    WOLFSSL_CTX * ctx,  
    int devId  
)
```

この関数は WOLFSSL_CTX コンテキストレベルで Device Id を設定します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された SSL コンテキストへのポインタ。
- **devId** 暗号コールバックまたは非同期ハードウェアで使用する ID。使用しない場合は INVALID_DEVID (-2) に設定します。

See:

- [wolfSSL_SetDevId](#)
- [wolfSSL_CTX_GetDevId](#)

Return:

- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG ssl が NULL の場合。

Example

```
WOLFSSL_CTX* ctx;  
int DevId = -2;  
  
wolfSSL_CTX_SetDevId(ctx, devId);
```

C.52.2.84 function wolfSSL_CTX_GetDevId

```
int wolfSSL_CTX_GetDevId(  
    WOLFSSL_CTX * ctx,  
    WOLFSSL * ssl  
)
```

この関数は Device Id を取得します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された SSL コンテキストへのポインタ。
- **ssl** [wolfSSL_new\(\)](#)で作成された SSL オブジェクトへのポインタ。

See:

- [wolfSSL_SetDevId](#)
- [wolfSSL_CTX_SetDevId](#)

Return:

- devId 成功時。
- INVALID_DEVID ssl と ctx の両方が NULL の場合。

Example

```
WOLFSSL_CTX* ctx;

wolfSSL_CTX_GetDevId(ctx, ssl);
```

C.52.2.85 function wolfSSL_CTX_set_session_cache_mode

```
long wolfSSL_CTX_set_session_cache_mode(
    WOLFSSL_CTX * ctx,
    long mode
)
```

この関数は SSL セッションキャッシングを有効または無効にします。動作は mode に使用される値に依存します。mode に使用できる値は以下の通りです。SSL_SESS_CACHE_OFF - セッションキャッシングを無効にします。セッションキャッシングはデフォルトで有効になっています。SSL_SESS_CACHE_NO_AUTO_CLEAR - セッションキャッシュの自動フラッシュを無効にします。自動フラッシュはデフォルトで有効になっています。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された SSL コンテキストへのポインタ。
- **mode** セッションキャッシュの動作を変更するために使用される修飾子。

See:

- [wolfSSL_flush_sessions](#)
- [wolfSSL_get1_session](#)
- [wolfSSL_set_session](#)
- [wolfSSL_get_sessionID](#)
- [wolfSSL_CTX_set_timeout](#)

Return: SSL_SUCCESS 成功時に返されます。

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_session_cache_mode(ctx, SSL_SESS_CACHE_OFF);
if (ret != SSL_SUCCESS) {
    // SSL セッションキャッシングをオフにすることができませんでした。
}
```

C.52.2.86 function wolfSSL_set_session_secret_cb

```
int wolfSSL_set_session_secret_cb(
    WOLFSSL * ssl,
    SessionSecretCb cb,
    void * ctx
)
```


この関数は、セッションシークレットコールバック関数を設定します。SessionSecretCb 型は次のシグネチャを持ちます: `int (SessionSecretCb)(WOLFSSL ssl, void* secret, int* secretSz, void* ctx)`。WOLFSSL 構造体の sessionSecretCb メンバが、パラメータ cb に設定されます。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** 上記のシグネチャを持つ関数ポインタである SessionSecretCb 型。
- **ctx** 保存されるユーザコンテキストへのポインタ。

See: SessionSecretCb

Return:

- SSL_SUCCESS 関数の実行がエラーを返さなかった場合に返されます。
- SSL_FATAL_ERROR WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
// SessionSecretCb のシグネチャ
int SessionSecretCb (WOLFSSL* ssl, void* secret, int* secretSz,
void* ctx) = SessionSecretCb;
...
int wolfSSL_set_session_secret_cb(ssl, SessionSecretCb, (void*)ssl->ctx){
    // 関数本体。
}
```

C.52.2.87 function wolfSSL_save_session_cache

```
int wolfSSL_save_session_cache(
    const char * fname
)
```

この関数は、セッションキャッシュをファイルに永続化します。追加のメモリ使用量のため、memsave は使用しません。

Parameters:

- **fname** 書き込み用ファイルを指す定数 char 型ポインタ。

See:

- XFWRITE
- `wolfSSL_restore_session_cache`
- `wolfSSL_memrestore_session_cache`

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。セッションキャッシュがファイルに書き込まれました。
- SSL_BAD_FILE fname を開くことができないか、またはそれ以外の理由で破損している場合に返されます。
- FWRITE_ERROR XFWRITE がファイルへの書き込みに失敗した場合に返されます。
- BAD_MUTEX_E mutex ロックの失敗があった場合に返されます。

Example

```
const char* fname;
...
if(wolfSSL_save_session_cache(fname) != SSL_SUCCESS){
```

```
// ファイルへの書き込みに失敗しました。
}
```

C.52.2.88 function wolfSSL_restore_session_cache

```
int wolfSSL_restore_session_cache(
    const char * fname
)
```

この関数は、永続的なセッションキャッシュをファイルから復元します。追加のメモリ使用量のため、memstore は使用しません。

Parameters:

- **fname** 読み取られる定数 char 型ポインタファイル入力。

See:

- XFREAD
- XFOPEN

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- SSL_BAD_FILE 関数に渡されたファイルが破損しており、XFOPEN で開くことができなかった場合に返されます。
- FREAD_ERROR ファイルが XFREAD からの読み取りエラーを持っていた場合に返されます。
- CACHE_MATCH_ERROR セッションキャッシュヘッダのマッチングに失敗した場合に返されます。
- BAD_MUTEX_E mutex ロックの失敗があった場合に返されます。

Example

```
const char *fname;
...
if(wolfSSL_restore_session_cache(fname) != SSL_SUCCESS){
    // 失敗ケースです。関数は SSL_SUCCESS を返しませんでした。
}
```

C.52.2.89 function wolfSSL_memsave_session_cache

```
int wolfSSL_memsave_session_cache(
    void * mem,
    int sz
)
```

この関数は、セッションキャッシュをメモリに永続化します。

Parameters:

- **mem** メモリコピー XMEMCPY() の宛先を表す void 型ポインタ。
- **sz** mem のサイズを表す int 型。

See:

- XMEMCPY
- [wolfSSL_get_session_cache_memsize](#)

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。セッションキャッシュがメモリに正常に永続化されました。
- BAD_MUTEX_E mutex ロックエラーがあった場合に返されます。

- BUFFER_E バッファサイズが小さすぎた場合に返されます。

Example

```
void* mem;
int sz; // メモリバッファの最大サイズ。
...
if(wolfSSL_memsave_session_cache(mem, sz) != SSL_SUCCESS){
    // 失敗ケースです。セッションキャッシュをメモリに永続化できませんでした。
}
```

C.52.2.90 function wolfSSL_memrestore_session_cache

```
int wolfSSL_memrestore_session_cache(
    const void * mem,
    int sz
)
```

この関数は、永続的なセッションキャッシュをメモリから復元します。

Parameters:

- **mem** 復元のソースを含む定数 void 型ポインタ。
- **sz** メモリバッファのサイズを表す整数。

See: [wolfSSL_save_session_cache](#)

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- BUFFER_E メモリバッファが小さすぎる場合に返されます。
- BAD_MUTEX_E セッションキャッシュの mutex ロックに失敗した場合に返されます。
- CACHE_MATCH_ERROR セッションキャッシュヘッダのマッチングに失敗した場合に返されます。

Example

```
const void* memoryFile;
int szMf;
...
if(wolfSSL_memrestore_session_cache(memoryFile, szMf) != SSL_SUCCESS){
    // 失敗ケースです。SSL_SUCCESS が返されませんでした。
}
```

C.52.2.91 function wolfSSL_get_session_cache_memsize

```
int wolfSSL_get_session_cache_memsize(
    void
)
```

この関数は、セッションキャッシュの保存バッファがどのくらい大きくあるべきかを返します。

Parameters:

- **none** パラメータなし。

See: [wolfSSL_memrestore_session_cache](#)

Return: int この関数は、セッションキャッシュの保存バッファのサイズを表す整数を返します。

Example

```
int sz = // エラーチェックのための最小サイズ;
...
if(sz < wolfSSL_get_session_cache_memsize()){
    // メモリバッファが小さすぎます。
}
```

C.52.2.92 function wolfSSL_CTX_save_cert_cache

```
int wolfSSL_CTX_save_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

この関数は、証明書キャッシュをメモリからファイルに書き込みます。

Parameters:

- **ctx** 証明書情報を保持する WOLFSSL_CTX 構造体へのポインタ。
- **fname** 書き込み用ファイルを指す定数 char 型ポインタ。

See:

- CM_SaveCertCache
- DoMemSaveCertCache

Return:

- SSL_SUCCESS CM_SaveCertCache が正常に終了した場合に返されます。
- BAD_FUNC_ARG いずれかの引数が NULL の場合に返されます。
- SSL_BAD_FILE 証明書キャッシュ保存ファイルを開くことができなかった場合に返されます。
- BAD_MUTEX_E ロック mutex が失敗した場合に返されます。
- MEMORY_E メモリの割り当てに失敗した場合に返されます。
- FWRITE_ERROR 証明書キャッシュファイルの書き込みに失敗しました。

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol def );
const char* fname;
...
if(wolfSSL_CTX_save_cert_cache(ctx, fname)){
    // ファイルが書き込まれました。
}
```

C.52.2.93 function wolfSSL_CTX_restore_cert_cache

```
int wolfSSL_CTX_restore_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

この関数は、証明書キャッシュをファイルから永続化します。

Parameters:

- **ctx** 証明書情報を保持する WOLFSSL_CTX 構造体へのポインタ。
- **fname** 読み取り用ファイルを指す定数 char 型ポインタ。

See:

- CM_RestoreCertCache
- XFOPEN

Return:

- SSL_SUCCESS 関数 CM_RestoreCertCache が正常に実行された場合に返されます。
- SSL_BAD_FILE XFOpen が XBADFILE を返した場合に返されます。ファイルが破損しています。
- MEMORY_E 一時バッファに割り当てられたメモリが失敗した場合に返されます。
- BAD_FUNC_ARG fname または ctx が NULL 値を持つ場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* fname = " ファイルへのパス";
...
if(wolfSSL_CTX_restore_cert_cache(ctx, fname)){
    // 実行が成功したかどうかを確認します。
}
```

C.52.2.94 function wolfSSL_CTX_memsave_cert_cache

```
int wolfSSL_CTX_memsave_cert_cache(
    WOLFSSL_CTX * ctx,
    void * mem,
    int sz,
    int * used
)
```

この関数は、証明書キャッシュをメモリに永続化します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **mem** 宛先 (出力バッファ) への void 型ポインタ。
- **sz** 出力バッファのサイズ。
- **used** 証明書キャッシュヘッダのサイズへのポインタ。

See:

- DoMemSaveCertCache
- GetCertCacheMemSize
- CM_MemRestoreCertCache
- CM_GetCertCacheMemSize

Return:

- SSL_SUCCESS 関数の実行が成功し、エラーが発生しなかった場合に返されます。
- BAD_MUTEX_E WOLFSSL_CERT_MANAGER メンバ caLock が 0(ゼロ) でなかった mutex エラー。
- BAD_FUNC_ARG ctx、mem、または used が NULL の場合、または sz が 0(ゼロ) 以下の場合に返されます。
- BUFFER_E 出力バッファ mem が小さすぎました。

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){
    // 関数がエラーを返しました。
}
```

C.52.2.95 function wolfSSL_CTX_memrestore_cert_cache

```
int wolfSSL_CTX_memrestore_cert_cache(
    WOLFSSL_CTX * ctx,
    const void * mem,
    int sz
)
```

この関数は、証明書キャッシュをメモリから復元します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **mem** 証明書キャッシュに復元される値を持つ void 型ポインタ。
- **sz** mem パラメータのサイズを表す int 型。

See: CM_MemRestoreCertCache

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなく実行された場合に返されます。
- BAD_FUNC_ARG ctx または mem パラメータが NULL の場合、または sz パラメータが 0 以下の場合に返されます。
- BUFFER_E 証明書キャッシュのメモリバッファが小さすぎる場合に返されます。
- CACHE_MATCH_ERROR 証明書キャッシュヘッダの不一致があった場合に返されます。
- BAD_MUTEX_E ロック mutex が失敗した場合に返されます。

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)){
    // 成功ケース
}
```

C.52.2.96 function wolfSSL_CTX_get_cert_cache_memsize

```
int wolfSSL_CTX_get_cert_cache_memsize(
    WOLFSSL_CTX * ctx
)
```

証明書キャッシュの保存バッファに必要なサイズを返します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された wolfSSL_CTX 構造体へのポインタ。

See: CM_GetCertCacheMemSize

Return:

- int 成功時にメモリサイズを表す整数値が返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合に返されます。
- BAD_MUTEX_E mutex ロックエラーがあった場合に返されます。

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){
// メモリサイズの取得に成功しました。
}
```

C.52.2.97 function wolfSSL_CTX_set_cipher_list

```
int wolfSSL_CTX_set_cipher_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

この関数は、指定された WOLFSSL_CTX に対する暗号スイートリストを設定します。設定したリストは、本コンテキストから新規に作成されるすべての WOLFSSL セッションのデフォルトリストとなり、文字列内の順序がそのまま優先順位（高い順から低い順）として用いられます。本関数を呼び出すたびに前回のリストは置き換えられます。リストは null 終端のテキスト文字列で、暗号スイート名および／または OpenSSL 互換のキーワードをコロンで区切って指定します（例: "TLS13-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-SHA256"）。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **list** 指定された SSL コンテキストで使用する暗号スイートおよび／またはキーワードの null 終端テキスト文字列（コロン区切り）。

See:

- `wolfSSL_set_cipher_list`
- `wolfSSL_get_ciphers`
- `wolfSSL_get_cipher_list`
- `wolfSSL_CTX_new`

Return:

- WOLFSSL_SUCCESS 関数が正常に完了した場合に返されます。
- WOLFSSL_FAILURE 失敗した場合に返されます。

リスト中の各トークンには次のいずれかを指定できます。

1. 個別の暗号スイート名。wolfSSL は独自の短縮名と IANA 名 (WOLFSSL_NO_ERROR_STRINGS が定義されていない場合) の両方を受理します。たとえば "ECDHE-RSA-AES128-GCM-SHA256" と "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256" は等価です。受理される名前の完全なリストは src/internal.c の cipher_names[] 配列にあります。実行時に現在のビルドに含まれるスイートを列挙するには `wolfSSL_get_ciphers()` を使用してください。
2. OpenSSL 互換のキーワード（下表参照）。スイート群を一括で選択したり、特定クラスを on/off するために使用します。一部のキーワードには OPENSSL_EXTRA または OPENSSL_ALL の定義が必要です。
3. 否定キーワード "!keyword" (OPENSSL_EXTRA または OPENSSL_ALL が必要)。例: "HIGH:!aNULL"。

"+" 演算子（例: "ECDHE+AESGCM"）は、先頭の公開鍵ファミリトークン ("ECDHE", "RSA", "DHE") の抽出にのみ使用されます。"+" の後の部分は wolfSSL では無視されます。特定のスイートを指定したい場合は完全なスイート名を直接指定してください。

OpenSSL 互換のグループキーワード:

キーワード	効果	必要なビルドオプション
DEFAULT / ALL	ビルドされている全スイートを含める。"ALL" は匿名 (aNULL) も有効化	OPENSSL_EXTRA / OPENSSL_ALL (他トークンを伴わない単独指定も可)

キーワード	効果	必要なビルドオプション
HIGH	静的・匿名・NULL を除く全スイート	OPENSSL_EXTRA / OPENSSL_ALL (単独指定も可)
LOW / MEDIUM	受理されるが鍵長による制限はかからず "RSA" 相当として扱われる	OPENSSL_EXTRA / OPENSSL_ALL
aNULL	匿名 (認証なし) スイート	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体は HAVE_ANON が必要)
eNULL / NULL	NULL 暗号化スイート	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体は HAVE_NULL_CIPHER が必要)
kDH	静的 DH 鍵交換	OPENSSL_EXTRA / OPENSSL_ALL
DHE / EDH	エフェメラル DH 鍵交換	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体は !NO_DH が必要)
ECDHE / ECDH	エフェメラル ECDH 鍵交換	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体は HAVE_ECC が必要)
kRSA / RSA	静的 RSA 鍵交換	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体は !NO_RSA が必要)
PSK	PSK スイート	OPENSSL_EXTRA / OPENSSL_ALL (スイート自体は !NO_PSK が必要)
DSS	黙って無視される (wolfSSL は DSA スイート非対応)	OPENSSL_EXTRA / OPENSSL_ALL
EXP / EXPORT	黙って無視される (export 等級は非対応)	OPENSSL_EXTRA / OPENSSL_ALL
AES128 / SHA1 / RC4	否定指定 ("!AES128" 等) でそのクラスを無効化	WOLFSSL_SYS_CRYPTOPOLICY
@SECLEVEL=n	OpenSSL 互換のセキュリティレベル設定	WOLFSSL_SYS_CRYPTOPOLICY

代表的な TLS 1.3 暗号スイート名 (各々個別の BUILD_* マクロでガードされる。多くは -enable-tls13 で自動有効化) :

名前 (wolfSSL)	IANA 名	必要なビルドオプション
TLS13-AES128-GCM-SHA256	TLS_AES_128_GCM_SHA256	BUILD_TLS_AES_128_GCM_SHA256 (TLS 1.3 既定)
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_AECCCM)
TLS13-AES128-CCM-8-SHA256	TLS_AES_128_CCM_8_SHA256	BUILD_TLS_AES_128_CCM_8_SHA256 (HAVE_AECCCM)
TLS13-SM4-GCM-SM3	TLS_SM4_GCM_SM3	BUILD_TLS_SM4_GCM_SM3 (WOLFSSL_SM4_GCM + WOLFSSL_SM3)

名前 (wolfSSL)	IANA 名	必要なビルドオプション
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

代表的な TLS 1.2 暗号スイート名のファミリ（各スイートは IANA 名から派生する個別の BUILD_* マクロでガードされる）：

スイート名のファミリ／例	一般的な必要オプション
ECDHE-ECDSA-AES128-GCM-SHA256, ECDHE-ECDSA-AES256-GCM-SHA384, ECDHE-ECDSA-CHACHA20-POLY1305	HAVE_ECC, HAVE_AESGCM（または HAVE_CHACHA + HAVE_POLY1305）
ECDHE-RSA-AES128-GCM-SHA256, ECDHE-RSA-AES256-GCM-SHA384, ECDHE-RSA-CHACHA20-POLY1305	HAVE_ECC, !NO_RSA, HAVE_AESGCM（または HAVE_CHACHA + HAVE_POLY1305）
DHE-RSA-AES128-GCM-SHA256, DHE-RSA-AES256-GCM-SHA384, DHE-RSA-CHACHA20-POLY1305	!NO_DH, !NO_RSA, HAVE_AESGCM（または HAVE_CHACHA + HAVE_POLY1305）
AES128-SHA, AES256-SHA, AES128-SHA256, AES256-SHA256, AES128-GCM-SHA256, AES256-GCM-SHA384（静的 RSA）	!NO_RSA, !NO_AES_CBC および／または HAVE_AESGCM
DES-CBC3-SHA, RC4-SHA, RC4-MD5 NULL-SHA, NULL-SHA256, NULL-MD5	レガシー: !NO_DES3 / !NO_RC4, !NO_OLD_TLS HAVE_NULL_CIPHER
PSK-AES128-CBC-SHA256, PSK-AES256-GCM-SHA384, ECDHE-PSK-AES128-CBC-SHA256, DHE-PSK-AES256-GCM-SHA384	!NO_PSK（ECDHE-PSK は HAVE_ECC、DHE-PSK は!NO_DH も必要）
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

注意事項:

- TLS 1.3 と TLS 1.2 のスイート名は同一リスト内で混在指定できます。wolfSSL は内部でバージョンごとに振り分けます。
- DTLS では、リスト中の RC4 ベースのストリーム暗号は黙って除外されます。
- リスト全体が "DEFAULT"、"ALL"、"HIGH"、または空文字列の場合、wolfSSL は組み込みのデフォルトスイートリストを設定して成功を返し、それ以上のトークン解析は行いません。

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) {
    // 暗号スイートリストの設定に失敗しました。
}
```

C.52.2.98 function wolfSSL_set_cipher_list

```
int wolfSSL_set_cipher_list(
    WOLFSSL * ssl,
    const char * list
)
```

この関数は、指定された WOLFSSL セッション（SSL セッション）に対する暗号スイートリストを設定します。リストの形式、認識される暗号スイート名およびキーワードは `wolfSSL_CTX_set_cipher_list()` の説明を参照してください。本関数を呼び出すたびに、当該セッションのリストは置き換えられます。

Parameters:

- **ssl** `wolfSSL_new()` で作成された SSL セッションへのポインタ。
- **list** 指定された SSL セッションで使用する暗号スイートおよび／またはキーワードの null 終端テキスト文字列（コロン区切り）。

See:

- `wolfSSL_CTX_set_cipher_list`
- `wolfSSL_get_ciphers`
- `wolfSSL_get_cipher_list`
- `wolfSSL_new`

Return:

- WOLFSSL_SUCCESS 関数が正常に完了すると返されます。
- WOLFSSL_FAILURE 失敗時に返されます。

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) {
    // 暗号スイートリストの設定に失敗
}
```

C.52.2.99 function wolfSSL_dtls_set_using_nonblock

```
void wolfSSL_dtls_set_using_nonblock(
    WOLFSSL * ssl,
    int nonblock
)
```

この関数は、WOLFSSL DTLS オブジェクトに対して、下層の UDP I/O がノンブロッキングであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキング UDP ソケットと共に使用する場合は、`wolfSSL_dtls_set_using_nonblock()` を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、recvfrom 呼び出しがブロックすることを意味すると認識できます。

Parameters:

- **ssl** `wolfSSL_new()` で作成された DTLS セッションへのポインタ。
- **nonblock** WOLFSSL オブジェクトのノンブロッキングフラグを設定するための値。ノンブロッキングを指定する場合は 1 を、そうでない場合は 0 を使用してください。

See:

- wolfSSL_dtls_get_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_get_current_timeout

Return: none 戻り値はありません。

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_dtls_set_using_nonblock(ssl, 1);
```

C.52.2.100 function wolfSSL_dtls_get_using_nonblock

```
int wolfSSL_dtls_get_using_nonblock(
    WOLFSSL * ssl
)
```

この関数により、アプリケーションは wolfSSL が UDP でノンブロッキング I/O を使用しているかどうかを判定できます。wolfSSL がノンブロッキング I/O を使用している場合、この関数は 1 を返し、それ以外の場合は 0 を返します。アプリケーションが WOLFSSL オブジェクトを作成した後、それをノンブロッキング UDP ソケットと共に使用する場合は、wolfSSL_dtls_set_using_nonblock() を呼び出してください。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受け取ることがタイムアウトではなく、recvfrom 呼び出しがブロックすることを意味すると認識できます。この関数は DTLS セッションにのみ意味があります。

Parameters:

- **ssl** wolfSSL_new() で作成された DTLS セッションへのポインタ。

See:

- wolfSSL_dtls_set_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_using_nonblock

Return:

- 0 下層の I/O がブロッキングです。
- 1 下層の I/O がノンブロッキングです。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_dtls_get_using_nonblock(ssl);
if (ret == 1) {
    // 下層の I/O はノンブロッキング
}
...
```

C.52.2.101 function wolfSSL_dtls_get_current_timeout

```
int wolfSSL_dtls_get_current_timeout(
    WOLFSSL * ssl
)
```

この関数は、WOLFSSL オブジェクトの現在のタイムアウト値を秒単位で返します。ノンブロッキングソケットを使用する場合、ユーザコード内の何かが、利用可能な受信データをいつチェックするか、およびどの

くらい待機しているかを決定する必要があります。この関数が返す値は、アプリケーションがどのくらい待機すべきかを示します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_dtls`
- `wolfSSL_dtls_get_peer`
- `wolfSSL_dtls_got_timeout`
- `wolfSSL_dtls_set_peer`

Return:

- seconds 現在の DTLS タイムアウト値 (秒単位)。
- NOT_COMPILED_IN `wolfSSL` が DTLS サポート付きでビルドされていない場合。

Example

```
int timeout = 0;
WOLFSSL* ssl;
...
timeout = wolfSSL_get_dtls_current_timeout(ssl);
printf("DTLS timeout (sec) = %d\n", timeout);
```

C.52.2.102 function `wolfSSL_dtls13_use_quick_timeout`

```
int wolfSSL_dtls13_use_quick_timeout(
    WOLFSSL * ssl
)
```

この関数は、アプリケーションがより短いタイムアウトを設定すべき場合に true を返します。ノンブロッキングソケットを使用する場合、ユーザコード内の何かが、利用可能なデータをいつチェックするか、およびどのくらい待機する必要があるかを決定する必要があります。この関数が true を返す場合、ライブラリはすでに通信の中断を検出していますが、他のピアからのメッセージがまだ転送中である可能性に備えて、もう少し待機したいことを意味します。このタイマーの値を微調整するのはアプリケーション次第であり、適切な値は `dtls_get_current_timeout() / 4` かもしれません。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_dtls`
- `wolfSSL_dtls_get_peer`
- `wolfSSL_dtls_got_timeout`
- `wolfSSL_dtls_set_peer`
- `wolfSSL_dtls13_set_send_more_acks`

Return: true アプリケーションコードがより短いタイムアウトを設定すべき場合。

C.52.2.103 function `wolfSSL_dtls13_set_send_more_acks`

```
void wolfSSL_dtls13_set_send_more_acks(
    WOLFSSL * ssl,
    int value
)
```

この関数は、中断を検出したときにライブラリが ACK を他のピアに即座に送信すべきかどうかを設定します。ACK を即座に送信すると最小のレイテンシが保証されますが、必要以上に帯域幅を消費する可能性があります。アプリケーションが自分でタイマーを管理し、このオプションが 0 に設定されている場合、アプリケーションコードは wolfSSL_dtls13_use_quick_timeout() を使用して、これらの遅延 ACK を送信するためにより短いタイムアウトを設定すべきかどうかを判定できます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **value** オプションを設定する場合は 1、オプションを無効にする場合は 0。

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls13_use_quick_timeout

C.52.2.104 function wolfSSL_dtls_set_timeout_init

```
int wolfSSL_dtls_set_timeout_init(
    WOLFSSL * ssl,
    int timeout
)
```

この関数は dtls タイムアウトを設定します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **timeout** WOLFSSL 構造体の dtls_timeout_init メンバに設定される int 型の値。

See:

- wolfSSL_dtls_set_timeout_max
- wolfSSL_dtls_got_timeout

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。SSL の dtls_timeout_init および dtls_timeout メンバが設定されています。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL であるか、またはタイムアウトが 0 より大きくない場合に返されます。また、timeout 引数が許容される最大値を超える場合にも返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int timeout = TIMEOUT;
...
if(wolfSSL_dtls_set_timeout_init(ssl, timeout)){
    // dtls タイムアウトが設定された
} else {
    // DTLS タイムアウトの設定に失敗
}
```

C.52.2.105 function wolfSSL_dtls_set_timeout_max

```
int wolfSSL_dtls_set_timeout_max(
    WOLFSSL * ssl,
```

```
    int timeout
)
```

この関数は最大 dtls タイムアウトを設定します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **timeout** dtls 最大タイムアウトを表す int 型の値。

See:

- `wolfSSL_dtls_set_timeout_init`
- `wolfSSL_dtls_got_timeout`

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL であるか、timeout 引数が 0 より大きくないか、または WOLFSSL 構造体の dtls_timeout_init メンバより小さい場合に返されます。

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){
    // 最大タイムアウトの設定に失敗
}
```

C.52.2.106 function `wolfSSL_dtls_got_timeout`

```
int wolfSSL_dtls_got_timeout(
    WOLFSSL * ssl
)
```

DTLS でノンブロッキングソケットを使用する場合、制御コードが送信がタイムアウトしたと判断したときに、この関数を WOLFSSL オブジェクトに対して呼び出す必要があります。この関数は、タイムアウト値の調整を含め、最後の送信を再試行するために必要なアクションを実行します。時間が経過しすぎた場合、失敗が返されます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_dtls_get_current_timeout`
- `wolfSSL_dtls_get_peer`
- `wolfSSL_dtls_set_peer`
- `wolfSSL_dtls`

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FATAL_ERROR ピアからの応答なしで再送信またはタイムアウトが多すぎた場合に返されます。
- NOT_COMPILED_IN `wolfSSL` が DTLS サポート付きでコンパイルされていない場合に返されます。

Example

使用例については以下のファイルを参照してください:

```
<wolfssl_root>/examples/client/client.c  
<wolfssl_root>/examples/server/server.c
```

C.52.2.107 function wolfSSL_dtls_retransmit

```
int wolfSSL_dtls_retransmit(  
    WOLFSSL * ssl  
)
```

DTLS でノンブロッキングソケットを使用する場合、この関数は予想されるタイムアウト値と再送信カウントを無視して、最後のハンドシェイクフライトを再送信します。これは、DTLS を使用していて、タイムアウトと再試行回数さえも管理する必要があるアプリケーションに役立ちます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FATAL_ERROR ピアからの応答なしで再送信またはタイムアウトが多すぎた場合に返されます。

Example

```
int ret = 0;  
WOLFSSL* ssl;  
...  
ret = wolfSSL_dtls_retransmit(ssl);
```

C.52.2.108 function wolfSSL_dtls

```
int wolfSSL_dtls(  
    WOLFSSL * ssl  
)
```

この関数は、SSL セッションが DTLS を使用するように設定されているかどうかを判定するために使用されます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer

Return:

- 1 SSL セッション (ssl) が DTLS を使用するように設定されている場合、この関数は 1 を返します。
- 0 それ以外の場合。

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_dtls(ssl);
if (ret) {
    // SSL セッションは DTLS を使用するように設定されている
}
```

C.52.2.109 function wolfSSL_dtls_set_peer

```
int wolfSSL_dtls_set_peer(
    WOLFSSL * ssl,
    void * peer,
    unsigned int peerSz
)
```

この関数は、DTLS ピア peer(sockaddr_in) をサイズ peerSz で設定します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **peer** ピアの sockaddr_in 構造体へのポインタ。NULL の場合、ssl 内のピア情報がクリアされます。
- **peerSz** peer が指す sockaddr_in 構造体のサイズ。0 の場合、ssl 内のピア情報がクリアされます。

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_set_pending_peer
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 失敗時に返されます。
- SSL_NOT_IMPLEMENTED wolfSSL が DTLS サポート付きでコンパイルされていない場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in addr;
...
ret = wolfSSL_dtls_set_peer(ssl, &addr, sizeof(addr));
if (ret != SSL_SUCCESS) {
    // DTLS ピアの設定に失敗
}
```

C.52.2.110 function wolfSSL_dtls_set_pending_peer

```
int wolfSSL_dtls_set_pending_peer(
    WOLFSSL * ssl,
    void * peer,
    unsigned int peerSz
)
```


この関数は、保留中の DTLS ピア peer(sockaddr_in) をサイズ peerSz で設定します。これにより、次のレコードの保護を正常に解除したときに通常のピアにアップグレードされる保留中のピアが設定されます。これは、ピアのアドレスが変更される可能性があるシナリオで、オフパス攻撃者がピアアドレスを変更するのを防ぐのに役立ちます。これは、新しいピアアドレスへのシームレスで安全な移行を可能にするために、Connection ID と共に使用する必要があります。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **peer** ピアの sockaddr_in 構造体へのポインタ。NULL の場合、ssl 内のピア情報がクリアされます。
- **peerSz** peer が指す sockaddr_in 構造体のサイズ。0 の場合、ssl 内のピア情報がクリアされます。

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 失敗時に返されます。
- SSL_NOT_IMPLEMENTED wolfSSL が DTLS サポート付きでコンパイルされていない場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in addr;
...
ret = wolfSSL_dtls_set_pending_peer(ssl, &addr, sizeof(addr));
if (ret != SSL_SUCCESS) {
    // DTLS ピアの設定に失敗
}
```

C.52.2.111 function wolfSSL_dtls_get_peer

```
int wolfSSL_dtls_get_peer(
    WOLFSSL * ssl,
    void * peer,
    unsigned int * peerSz
)
```

この関数は、現在の DTLS ピアの sockaddr_in(サイズ peerSz) を取得します。この関数は peerSz を、SSL セッションに格納されている実際の DTLS ピアサイズと比較します。ピアが peer に収まる場合、ピアの sockaddr_in が peer にコピーされ、peerSz が peer のサイズに設定されます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **peer** ピアの sockaddr_in 構造体を格納するメモリ位置へのポインタ。
- **peerSz** 入出力サイズ。入力として、peer が指す割り当てられたメモリのサイズ。出力として、peer が指す実際の sockaddr_in 構造体のサイズ。

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_got_timeout

- `wolfSSL_dtls_set_peer`
- `wolfSSL_dtls`

Return:

- `SSL_SUCCESS` 成功時に返されます。
- `SSL_FAILURE` 失敗時に返されます。
- `SSL_NOT_IMPLEMENTED` wolfSSL が DTLS サポート付きでコンパイルされていない場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in addr;
... ret = wolfSSL_dtls_get_peer(ssl, &addr, sizeof(addr));
if (ret != SSL_SUCCESS) {
    // DTLS ピアの取得に失敗しました
}
```

C.52.2.112 function wolfSSL_dtls_get0_peer

```
int wolfSSL_dtls_get0_peer(
    WOLFSSL * ssl,
    const void ** peer,
    unsigned int * peerSz
)
```

この関数は現在の DTLS ピアの `sockaddr_in` (サイズ `peerSz`) を取得します。これは `wolfSSL_dtls_get_peer()` のゼロコピー代替です。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **peer** ピアアドレスを保持する内部バッファを返すためのポインタ。
- **peerSz** `peer` が指す実際の `sockaddr_in` 構造体のサイズを出力します。

See:

- `wolfSSL_dtls_get_current_timeout`
- `wolfSSL_dtls_got_timeout`
- `wolfSSL_dtls_set_peer`
- `wolfSSL_dtls`

Return:

- `SSL_SUCCESS` 成功時に返されます。
- `SSL_FAILURE` 失敗時に返されます。
- `SSL_NOT_IMPLEMENTED` wolfSSL が DTLS サポートでコンパイルされていない場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in* addr;
unsigned int addrSz;
...
ret = wolfSSL_dtls_get_peer(ssl, &addr, &addrSz);
if (ret != SSL_SUCCESS) {
    // DTLS ピアの取得に失敗しました
}
```

C.52.2.113 function wolfSSL_ERR_error_string

```
char * wolfSSL_ERR_error_string(  
    unsigned long errNumber,  
    char * data  
)
```

この関数は wolfSSL_get_error() によって返されたエラーコードを、より人間が読みやすいエラー文字列に変換します。errNumber は wolfSSL_get_error() によって返されたエラーコードであり、data はエラー文字列が配置される格納バッファです。data の最大長はデフォルトで 80 文字であり、wolfssl/wolfcrypt/error.h の MAX_ERROR_SZ で定義されています。

Parameters:

- **errNumber** wolfSSL_get_error() によって返されたエラーコード。
- **data** errNumber に一致する人間が読みやすいエラー文字列を含む出力バッファ。

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- success 正常に完了した場合、この関数は data で返されるのと同じ文字列を返します。
- failure 失敗時には、この関数は適切な失敗理由 msg を含む文字列を返します。

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.52.2.114 function wolfSSL_ERR_error_string_n

```
void wolfSSL_ERR_error_string_n(  
    unsigned long e,  
    char * buf,  
    unsigned long len  
)
```

この関数は wolfSSL_ERR_error_string() のバージョンであり、len は buf に書き込まれる最大文字数を指定します。wolfSSL_ERR_error_string() と同様に、この関数は wolfSSL_get_error() から返されたエラーコードをより人間が読みやすいエラー文字列に変換します。人間が読みやすい文字列は buf に配置されます。

Parameters:

- **e** wolfSSL_get_error() によって返されたエラーコード。
- **buf** e に一致する人間が読みやすいエラー文字列を含む出力バッファ。
- **len** buf に書き込まれる最大文字数。

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_print_errors_fp

- `wolfSSL_load_error_strings`

Return: none 返り値はありません。

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.52.2.115 function `wolfSSL_get_shutdown`

```
int wolfSSL_get_shutdown(
    const WOLFSSL * ssl
)
```

この関数は Options 構造体の `closeNotify`、`connReset`、または `sentNotify` メンバのシャットダウン条件をチェックします。Options 構造体は WOLFSSL 構造体内にあります。

Parameters:

- `ssl` `wolfSSL_new()`を使用して作成された WOLFSSL 構造体への定数ポインタ。

See: `wolfSSL_SESSION_free`

Return:

- 1 `SSL_SENT_SHUTDOWN` が返されます。
- 2 `SSL_RECEIVED_SHUTDOWN` が返されます。

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 {
    // 致命的エラー。
}
```

C.52.2.116 function `wolfSSL_session_reused`

```
int wolfSSL_session_reused(
    WOLFSSL * ssl
)
```

この関数は options 構造体の `resuming` メンバを返します。このフラグはセッションを再利用するかどうかを示します。再利用しない場合、新しいセッションを確立する必要があります。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_SESSION_free`
- `wolfSSL_GetSessionIndex`
- `wolfSSL_memsave_session_cache`

Return: この関数はセッション再利用のフラグを表す Options 構造体に保持された int 型を返します。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(!wolfSSL_session_reused(sslResume)){
    // セッション再利用は許可されていません。
}
```

C.52.2.117 function wolfSSL_is_init_finished

```
int wolfSSL_is_init_finished(
    const WOLFSSL * ssl
)
```

この関数は接続が確立されているかどうかをチェックします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_set_accept_state`
- `wolfSSL_get_keys`
- `wolfSSL_set_shutdown`

Return:

- 0 接続が確立されていない場合に返されます、つまり WOLFSSL 構造体が NULL またはハンドシェイクが完了していない場合。
- 1 接続が確立されている場合に返されます、つまり WOLFSSL ハンドシェイクが完了している場合。

EXAMPLE

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_is_init_finished(ssl)){
    // ハンドシェイクが完了し、接続が確立されています
}
```

C.52.2.118 function wolfSSL_get_version

```
const char * wolfSSL_get_version(
    WOLFSSL * ssl
)
```

使用されている SSL バージョンを文字列として返します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_lib_version`

Return:

- "SSLv3" SSLv3 を使用しています
- "TLSv1" TLSv1 を使用しています
- "TLSv1.1" TLSv1.1 を使用しています
- "TLSv1.2" TLSv1.2 を使用しています
- "TLSv1.3" TLSv1.3 を使用しています
- "DTLS" DTLS を使用しています
- "DTLSv1.2" DTLSv1.2 を使用しています
- "unknown" 使用されている TLS のバージョンを判定する際に問題が発生しました。

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // 何らかの wolfSSL メソッド
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);
printf(wolfSSL_get_version("Using version: %s", ssl));
```

C.52.2.119 function `wolfSSL_get_current_cipher_suite`

```
int wolfSSL_get_current_cipher_suite(
    WOLFSSL * ssl
)
```

ssl セッションが使用している現在の暗号スイートを返します。

Parameters:

- **ssl** チェックする SSL セッション。

See:

- `wolfSSL_CIPHER_get_name`
- `wolfSSL_get_current_cipher`
- `wolfSSL_get_cipher_list`

Return:

- `ssl->options.cipherSuite` 現在の暗号スイートを表す整数。
- 0 提供された ssl セッションが null です。

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // 何らかの wolfSSL メソッド
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_get_current_cipher_suite(ssl) == 0)
```

```
{
    // 暗号スイートの取得エラー
}
```

C.52.2.120 function wolfSSL_get_current_cipher

```
WOLFSSL_CIPHER * wolfSSL_get_current_cipher(
    WOLFSSL * ssl
)
```

この関数は ssl セッション内の現在の暗号へのポインタを返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_get_cipher](#)
- [wolfSSL_get_cipher_name_internal](#)
- [wolfSSL_get_cipher_name](#)

Return:

- この関数は WOLFSSL 構造体の cipher メンバのアドレスを返します。これは WOLFSSL_CIPHER 構造体へのポインタです。
- NULL WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
WOLFSSL_CIPHER* cipherCurr = wolfSSL_get_current_cipher;

if(!cipherCurr){
    // 失敗ケース。
} else {
    // 暗号が cipherCurr に返されました
}
```

C.52.2.121 function wolfSSL_CIPHER_get_name

```
const char * wolfSSL_CIPHER_get_name(
    const WOLFSSL_CIPHER * cipher
)
```

この関数は SSL オブジェクト内の暗号スイートを利用可能なスイートと照合し、文字列表現を返します。

Parameters:

- **cipher** WOLFSSL_CIPHER 構造体への定数ポインタ。

See:

- [wolfSSL_get_cipher](#)
- [wolfSSL_get_current_cipher](#)
- [wolfSSL_get_cipher_name_internal](#)
- [wolfSSL_get_cipher_name](#)

Return:

- string この関数は一致した暗号スイートの文字列表現を返します。
- none 一致するスイートがない場合は "None" を返します。

Example

```
// DHE_RSA ... の形式で暗号名を取得します
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){
    // 返された暗号の健全性チェック
}
```

C.52.2.122 function wolfSSL_get_cipher

```
const char * wolfSSL_get_cipher(
WOLFSSL *
```

この関数は SSL オブジェクト内の暗号スイートを利用可能なスイートと照合します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return: この関数は一致したスイートの文字列値を返します。一致するスイートがない場合は "None" を返します。

Example

```
#ifdef WOLFSSL_DTLS
...
// 有効なスイートが使用されていることを確認
if(wolfSSL_get_cipher(ssl) == NULL){
    WOLFSSL_MSG(" インポートされた暗号スイートと一致しません");
    return MATCH_SUITE_ERROR;
}
...
#endif // WOLFSSL_DTLS
```

C.52.2.123 function wolfSSL_get1_session

```
WOLFSSL_SESSION * wolfSSL_get1_session(
WOLFSSL * ssl
)
```

この関数は WOLFSSL 構造体から WOLFSSL_SESSION を参照型として返します。これには wolfSSL_SESSION_free を呼び出してセッション参照を解放する必要があります。指される WOLFSSL_SESSION には、セッション再開を実行し、新しいハンドシェイクなしで接続を再確立するために必要なすべての情報が含まれています。セッション再開のために、セッションオブジェクトで wolfSSL_shutdown() を呼び出す前に、アプリケーションは wolfSSL_get1_session() の呼び出しでオブジェクトからセッション ID を保存す

る必要があります、これはセッションへのポインタを返します。後で、アプリケーションは新しい WOLFSSL オブジェクトを作成し、wolfSSL_set_session() で保存されたセッションを割り当てる必要があります。この時点で、アプリケーションは wolfSSL_connect() を呼び出すことができ、wolfSSL はセッションの再開を試みます。wolfSSL サーバーコードはデフォルトでセッション再開を許可します。wolfSSL_get1_session() によって返されたオブジェクトは、アプリケーションが使用を終えた後、wolfSSL_SESSION_free() を呼び出すことによって解放する必要があります。

Parameters:

- **ssl** セッションを取得する WOLFSSL 構造体。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)
- [wolfSSL_SESSION_free](#)

Return:

- WOLFSSL_SESSION 成功時にセッションポインタを返します。
- NULL ssl が NULL、SSL セッションキャッシュが無効、wolfSSL がセッション ID を利用できない、または mutex 関数が失敗した場合に返されます。

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* ses;    // ハンドシェイクを試行/完了
wolfSSL_connect(ssl);
ses = wolfSSL_get1_session(ssl);
// ses 情報を確認
// 切断/新しい SSL インスタンスをセットアップ
wolfSSL_set_session(ssl, ses);
// ハンドシェイクを試行/再開
wolfSSL_SESSION_free(ses);
```

C.52.2.124 function wolfSSLv23_client_method

```
WOLFSSL_METHOD * wolfSSLv23_client_method(
    void
)
```

[wolfSSLv23_client_method\(\)](#)関数は、アプリケーションがクライアントであり、SSL 3.0 から TLS 1.3 までの間でサーバがサポートする最も高いプロトコルバージョンをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい WOLFSSL_METHOD 構造体のためのメモリを割り当て、初期化します。wolfSSL のクライアントとサーバの両方は、堅牢なバージョンダウングレード機能を持っています。どちらか一方で特定のプロトコルバージョンメソッドが使用された場合、そのバージョンのみがネゴシエートされるか、エラーが返されます。例えば、TLSv1 を使用するクライアントが SSLv3 のみのサーバに接続しようとするとう失敗し、同様に TLSv1.1 への接続も失敗します。この問題を解決するために、wolfSSLv23_client_method() 関数を使用するクライアントは、サーバがサポートする最も高いプロトコルバージョンを使用し、必要に応じて SSLv3 までダウングレードします。この場合、クライアントは SSLv3 から TLSv1.3 までを実行しているサーバに接続できます。

Parameters:

- **none** パラメータなし。

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 成功時、WOLFSSL_METHOD へのポインタ。
- Failure XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
method = wolfSSLv23_client_method();
if (method == NULL) {
    // メソッドを取得できませんでした
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.125 function wolfSSL_BIO_get_mem_data

```
int wolfSSL_BIO_get_mem_data(
    WOLFSSL_BIO * bio,
    void * p
)
```

これは、内部メモリバッファの先頭にバイトポインタを設定するために使用されます。

Parameters:

- **bio** メモリバッファを取得する WOLFSSL_BIO 構造体。
- **p** メモリバッファに設定するバイトポインタ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- size 成功時、バッファのサイズが返されます。
- SSL_FATAL_ERROR エラーケースが発生した場合。

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);
// ret 値を確認
```

C.52.2.126 function wolfSSL_BIO_set_fd

```
long wolfSSL_BIO_set_fd(
    WOLFSSL_BIO * b,
    int fd,
    int flag
)
```

bio が使用するファイルディスクリプタを設定します。

Parameters:

- **bio** fd を設定する WOLFSSL_BIO 構造体。
- **fd** 使用するファイルディスクリプタ。
- **closeF** fd をクローズする際の動作フラグ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) 成功時。

Example

```
WOLFSSL_BIO* bio;
int fd;
// bio をセットアップ
wolfSSL_BIO_set_fd(bio, fd, BIO_NOCLOSE);
```

C.52.2.127 function wolfSSL_BIO_set_close

```
int wolfSSL_BIO_set_close(
    WOLFSSL_BIO * b,
    long flag
)
```

クローズフラグを設定します。これは、BIO が解放される際に I/O ストリームをクローズすべきかを示すために使用されます。

Parameters:

- **bio** WOLFSSL_BIO 構造体。
- **flag** I/O ストリームをクローズする際の動作フラグ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) 成功時。

Example

```
WOLFSSL_BIO* bio;
// bio をセットアップ
wolfSSL_BIO_set_close(bio, BIO_NOCLOSE);
```

C.52.2.128 function wolfSSL_BIO_s_socket

```
WOLFSSL_BIO_METHOD * wolfSSL_BIO_s_socket(
    void
)
```

これは、BIO_SOCKET タイプの WOLFSSL_BIO_METHOD を取得するために使用されます。

Parameters:

- **none** パラメータなし。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return: WOLFSSL_BIO_METHOD ソケットタイプである WOLFSSL_BIO_METHOD 構造体へのポインタ。

Example

```
WOLFSSL_BIO* bio;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_socket);
```

C.52.2.129 function wolfSSL_BIO_set_write_buf_size

```
int wolfSSL_BIO_set_write_buf_size(
    WOLFSSL_BIO * b,
    long size
)
```

これは、WOLFSSL_BIO の書き込みバッファのサイズを設定するために使用されます。書き込みバッファが以前に設定されていた場合、この関数はサイズをリセットする際にそれを解放します。これは、読み取りと書き込みのインデックスを 0 にリセットする点で wolfSSL_BIO_reset に似ています。

Parameters:

- **bio** fd を設定する WOLFSSL_BIO 構造体。
- **size** 割り当てるバッファのサイズ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS 書き込みバッファの設定に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。

Example

```
WOLFSSL_BIO* bio;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret = wolfSSL_BIO_set_write_buf_size(bio, 15000);
// 戻り値を確認
```

C.52.2.130 function wolfSSL_BIO_make_bio_pair

```
int wolfSSL_BIO_make_bio_pair(
    WOLFSSL_BIO * b1,
    WOLFSSL_BIO * b2
)
```

これは、2 つの bio をペアにするために使用されます。ペアになった bio は双方向パイプのように動作し、一方への書き込みはもう一方から読み取ることができ、その逆も同様です。両方の bio が同じスレッドにあることが期待されます。この関数はスレッドセーフではありません。2 つの bio のうちの 1 つを解放する

と、両方のペアが解除されます。いずれかの bio に書き込みバッファサイズが以前に設定されていなかった場合、ペアになる前にデフォルトサイズの 17000(WOLFSSL_BIO_SIZE) に設定されます。

Parameters:

- **b1** ペアを設定する WOLFSSL_BIO 構造体。
- **b2** ペアを完成させる 2 番目の WOLFSSL_BIO 構造体。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS 2 つの bio のペア化に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。

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);
// ret 値を確認
```

C.52.2.131 function wolfSSL_BIO_ctrl_reset_read_request

```
int wolfSSL_BIO_ctrl_reset_read_request(
    WOLFSSL_BIO * b
)
```

これは、読み取り要求フラグを 0 に戻すために使用されます。

Parameters:

- **b** 読み取り要求フラグを設定する WOLFSSL_BIO 構造体。

See:

- wolfSSL_BIO_new, wolfSSL_BIO_s_mem
- wolfSSL_BIO_new, wolfSSL_BIO_free

Return:

- SSL_SUCCESS 値の設定に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。

Example

```
WOLFSSL_BIO* bio;
int ret;
...
ret = wolfSSL_BIO_ctrl_reset_read_request(bio);
// ret 値を確認
```

C.52.2.132 function wolfSSL_BIO_nread0

```
int wolfSSL_BIO_nread0(
    WOLFSSL_BIO * bio,
    char ** buf
)
```

これは、読み取り用のバッファポインタを取得するために使用されます。wolfSSL_BIO_nread とは異なり、内部読み取りインデックスは関数呼び出しから返される数だけ進められません。返される値を超えて読み取ると、配列の境界外を読み取る結果になる可能性があります。

Parameters:

- **bio** 読み取る WOLFSSL_BIO 構造体。
- **buf** 読み取り配列の先頭に設定するポインタ。

See: wolfSSL_BIO_new

Return: > =0 成功時、読み取るバイト数を返します。

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;
// bio をセットアップ
ret = wolfSSL_BIO_nread0(bio, &bufPt); // 可能な限り多くのバイトを読み取り
// 負の ret 値を確認
// bufPt から ret バイトを読み取り
```

C.52.2.133 function wolfSSL_BIO_nread

```
int wolfSSL_BIO_nread(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

これは、読み取り用のバッファポインタを取得するために使用されます。内部読み取りインデックスは、関数呼び出しから返される数だけ進められ、buf は読み取るバッファの先頭を指します。読み取りバッファ内のバイト数が num で要求された値より少ない場合、より小さい値が返されます。返される値を超えて読み取ると、配列の境界外を読み取る結果になる可能性があります。

Parameters:

- **bio** 読み取る WOLFSSL_BIO 構造体。
- **buf** 読み取り配列の先頭に設定するポインタ。
- **num** 読み取りを試みるバイト数。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_nwrite

Return:

- > =0 成功時、読み取るバイト数を返します。
- WOLFSSL_BIO_ERROR(-1) 読み取るものがない場合のエラーケースで-1 を返します。

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;

// bio をセットアップ
ret = wolfSSL_BIO_nread(bio, &bufPt, 10); // 10 バイトの読み取りを試行
// 負の ret 値を確認
// bufPt から ret バイトを読み取り
```

C.52.2.134 function wolfSSL_BIO_nwrite

```
int wolfSSL_BIO_nwrite(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

関数が返す数だけのバイトを書き込むためのバッファへのポインタを取得します。返される値よりも多くのバイトを返されたポインタに書き込むと、境界外への書き込みになる可能性があります。

Parameters:

- **bio** 書き込む WOLFSSL_BIO 構造体。
- **buf** 書き込むバッファへのポインタ。
- **num** 書き込みたいバイト数。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free
- **wolfSSL_BIO_nread**

Return:

- int バッファポインタに書き込むことができるバイト数を返します。
- WOLFSSL_BIO_UNSET(-2) bio ペアの一部ではない場合。
- WOLFSSL_BIO_ERROR(-1) 書き込むスペースがこれ以上ない場合。

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;
// bio をセットアップ
ret = wolfSSL_BIO_nwrite(bio, &bufPt, 10); // 10 バイトの書き込みを試行
// 負の ret 値を確認
// bufPt に ret バイトを書き込み
```

C.52.2.135 function wolfSSL_BIO_reset

```
int wolfSSL_BIO_reset(
    WOLFSSL_BIO * bio
)
```

bio を初期状態にリセットします。例えば、BIO_BIO タイプの場合、これは読み取りと書き込みのインデックスをリセットします。

Parameters:

- **bio** リセットする WOLFSSL_BIO 構造体。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return:

- 0 bio のリセットに成功した場合。
- WOLFSSL_BIO_ERROR(-1) 不正な入力またはリセットに失敗した場合に返されます。

Example

```
WOLFSSL_BIO* bio;  
// bio をセットアップ  
wolfSSL_BIO_reset(bio);  
// pt を使用
```

C.52.2.136 function wolfSSL_BIO_seek

```
int wolfSSL_BIO_seek(  
    WOLFSSL_BIO * bio,  
    int ofs  
)
```

この関数は、ファイルポインタを指定されたオフセットに調整します。これはファイルの先頭からのオフセットです。

Parameters:

- **bio** 設定する WOLFSSL_BIO 構造体。
- **ofs** ファイルへのオフセット。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- **wolfSSL_BIO_set_fp**
- wolfSSL_BIO_free

Return:

- 0 シークに成功した場合。
- -1 エラーケースが発生した場合。

Example

```
WOLFSSL_BIO* bio;  
XFILE fp;  
int ret;  
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_file());  
ret = wolfSSL_BIO_set_fp(bio, &fp);  
// ret 値を確認  
ret = wolfSSL_BIO_seek(bio, 3);  
// ret 値を確認
```

C.52.2.137 function wolfSSL_BIO_write_filename

```
int wolfSSL_BIO_write_filename(  
    WOLFSSL_BIO * bio,  
    char * name  
)
```


ファイルを設定し、書き込むために使用されます。ファイル内の既存データを上書きし、bio が解放される際にファイルを閉じるよう設定されます。

Parameters:

- **bio** ファイルを設定する WOLFSSL_BIO 構造体。
- **name** 書き込み先のファイル名。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_file
- **wolfSSL_BIO_set_fp**
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS ファイルのオープンと設定が成功した場合。
- SSL_FAILURE エラーが発生した場合。

Example

```
WOLFSSL_BIO* bio;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret = wolfSSL_BIO_write_filename(bio, "test.txt");
// ret 値を確認
```

C.52.2.138 function wolfSSL_BIO_set_mem_eof_return

```
long wolfSSL_BIO_set_mem_eof_return(
    WOLFSSL_BIO * bio,
    int v
)
```

ファイル終端値を設定するために使用されます。一般的な値は-1 で、期待される正の値と混同しないようにします。

Parameters:

- **bio** ファイル終端値を設定する WOLFSSL_BIO 構造体。
- **v** bio に設定する値。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- **wolfSSL_BIO_set_fp**
- wolfSSL_BIO_free

Return: 0 完了時に返されます。

Example

```
WOLFSSL_BIO* bio;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret = wolfSSL_BIO_set_mem_eof_return(bio, -1);
// ret 値を確認
```

C.52.2.139 function wolfSSL_BIO_get_mem_ptr

```
long wolfSSL_BIO_get_mem_ptr(  
    WOLFSSL_BIO * bio,  
    WOLFSSL_BUF_MEM ** m  
)
```

WOLFSSL_BIO メモリポインタの getter 関数です。

Parameters:

- **bio** メモリポインタを取得するための WOLFSSL_BIO 構造体へのポインタ。
- **ptr** 現在 char* である構造体。bio のメモリを指すように設定されます。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return:

- SSL_SUCCESS ポインタの取得に成功した場合、SSL_SUCCESS が返されます (現在の値は 1)。
- SSL_FAILURE NULL 引数が渡された場合に返されます (現在の値は 0)。

Example

```
WOLFSSL_BIO* bio;  
WOLFSSL_BUF_MEM* pt;  
// bio をセットアップ  
wolfSSL_BIO_get_mem_ptr(bio, &pt);  
// pt を使用
```

C.52.2.140 function wolfSSL_X509_NAME_online

```
char * wolfSSL_X509_NAME_online(  
    WOLFSSL_X509_NAME * name,  
    char * in,  
    int sz  
)
```

この関数は x509 の名前をバッファにコピーします。

Parameters:

- **name** WOLFSSL_X509 構造体へのポインタ。
- **in** WOLFSSL_X509_NAME 構造体からコピーされた名前を保持するバッファ。
- **sz** バッファの最大サイズ。

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: 関数が正常に実行された場合、WOLFSSL_X509_NAME 構造体の name メンバーのデータを持つバッファへの char ポインタが返されます。

Example

```
WOLFSSL_X509 x509;  
char* name;
```

```
...
name = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(name <= 0){
    // バッファに何もありません
}
```

C.52.2.141 function wolfSSL_X509_get_issuer_name

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_issuer_name(
    WOLFSSL_X509 * cert
)
```

この関数は証明書の発行者名を返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ。

See:

- [wolfSSL_X509_get_subject_name](#)
- [wolfSSL_X509_get_isCA](#)
- [wolfSSL_get_peer_certificate](#)
- [wolfSSL_X509_NAME_oneline](#)

Return:

- point 渡された cert が NULL でない場合、WOLFSSL_X509 構造体の issuer メンバーへのポインタが返されます。
- NULL 渡された cert が NULL の場合。

Example

```
WOLFSSL_X509* x509;
WOLFSSL_X509_NAME issuer;
...
issuer = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(!issuer){
    // NULL が返されました
} else {
    // issuer は証明書の発行者名を保持しています
}
```

C.52.2.142 function wolfSSL_X509_get_subject_name

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_subject_name(
    WOLFSSL_X509 * cert
)
```

この関数は WOLFSSL_X509 構造体の subject メンバーを返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ。

See:

- [wolfSSL_X509_get_issuer_name](#)
- [wolfSSL_X509_get_isCA](#)

- `wolfSSL_get_peer_certificate`

Return: pointer WOLFSSL_X509_NAME 構造体へのポインタ。WOLFSSL_X509 構造体が NULL の場合、または構造体の subject メンバーが NULL の場合、ポインタは NULL になる可能性があります。

Example

```
WOLFSSL_X509* cert;
WOLFSSL_X509_NAME name;
...
name = wolfSSL_X509_get_subject_name(cert);
if(name == NULL){
    // NULL ケースを処理
}
```

C.52.2.143 function `wolfSSL_X509_get_isCA`

```
int wolfSSL_X509_get_isCA(
    WOLFSSL_X509 * x509
)
```

WOLFSSL_X509 構造体の isCa メンバーをチェックし、その値を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

See:

- `wolfSSL_X509_get_issuer_name`
- `wolfSSL_X509_get_isCA`

Return:

- isCA WOLFSSL_X509 構造体の isCa メンバーの値が返されます。
- 0 有効な x509 構造体が渡されなかった場合に返されます。

Example

```
WOLFSSL* ssl;
...
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_X509_get_isCA(ssl)){
    // これは CA です
}else {
    // 失敗ケース
}
```

C.52.2.144 function `wolfSSL_X509_NAME_get_text_by_NID`

```
int wolfSSL_X509_NAME_get_text_by_NID(
    WOLFSSL_X509_NAME * name,
    int nid,
    char * buf,
    int len
)
```

この関数は渡された NID 値に関連するテキストを取得します。

Parameters:

- **name** テキストを検索する WOLFSSL_X509_NAME。
- **nid** 検索する NID。
- **buf** 見つかったテキストを保持するバッファ。
- **len** バッファの長さ。

See: none**Return:** int テキストバッファのサイズを返します。*Example*

```
WOLFSSL_X509_NAME* name;
char buffer[100];
int bufferSz;
int ret;
// WOLFSSL_X509_NAME を取得
ret = wolfSSL_X509_NAME_get_text_by_NID(name, NID_commonName,
buffer, bufferSz);

// ret 値を確認
```

C.52.2.145 function wolfSSL_X509_get_signature_type

```
int wolfSSL_X509_get_signature_type(
    WOLFSSL_X509 * x509
)
```

この関数は WOLFSSL_X509 構造体の sigOID メンバーに格納されている値を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- int x509 オブジェクトから取得された整数値が返されます。

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){
    // 予期しない値を処理
}
```

C.52.2.146 function wolfSSL_X509_free

```
void wolfSSL_X509_free(
    WOLFSSL_X509 * x509
)
```

この関数は WOLFSSL_X509 構造体を解放します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

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.52.2.147 function wolfSSL_X509_get_signature

```
int wolfSSL_X509_get_signature(
    WOLFSSL_X509 * x509,
    unsigned char * buf,
    int * bufSz
)
```

X509 署名を取得し、バッファに格納します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。
- **buf** バッファへの char ポインタ。
- **bufSz** バッファのサイズへの整数ポインタ。

See:

- [wolfSSL_X509_get_serial_number](#)
- [wolfSSL_X509_get_signature_type](#)
- [wolfSSL_X509_get_device_type](#)

Return:

- SSL_SUCCESS 関数が正常に実行された場合に返されます。署名はバッファに読み込まれます。
- SSL_FATAL_ERROR x509 構造体または bufSz メンバーが NULL の場合に返されます。sig 構造体の length メンバー (sig は x509 のメンバー) のチェックもあります。

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
unsigned char* buf; // 初期化
int* bufSz = sizeof(buf)/sizeof(unsigned char);
...
```

```
if(wolfSSL_X509_get_signature(x509, buf, bufSz) != SSL_SUCCESS){  
    // 関数は正常に実行されませんでした  
} else{  
    // バッファは正しく書き込まれました  
}
```

C.52.2.148 function wolfSSL_X509_STORE_add_cert

```
int wolfSSL_X509_STORE_add_cert(  
    WOLFSSL_X509_STORE * store,  
    WOLFSSL_X509 * x509  
)
```

この関数は WOLFSSL_X509_STORE 構造体に証明書を追加します。

Parameters:

- **str** 証明書を追加する証明書ストア。
- **x509** 追加する証明書。

See: [wolfSSL_X509_free](#)

Return:

- SSL_SUCCESS 証明書が正常に追加された場合。
- SSL_FATAL_ERROR 証明書が正常に追加されなかった場合。

Example

```
WOLFSSL_X509_STORE* str;  
WOLFSSL_X509* x509;  
int ret;  
ret = wolfSSL_X509_STORE_add_cert(str, x509);  
// ret 値を確認
```

C.52.2.149 function wolfSSL_X509_STORE_CTX_get_chain

```
WOLFSSL_STACK * wolfSSL_X509_STORE_CTX_get_chain(  
    WOLFSSL_X509_STORE_CTX * ctx  
)
```

この関数は WOLFSSL_X509_STORE_CTX 構造体の chain 変数の getter 関数です。現在、chain は入力されていません。

Parameters:

- **ctx** 解析チェーンを取得する証明書ストアコンテキスト。

See: [wolfSSL_sk_X509_free](#)

Return:

- pointer 成功した場合、WOLFSSL_STACK(STACK_OF(WOLFSSL_X509) と同じ) ポインタを返します。
- Null 失敗時。

Example

```
WOLFSSL_STACK* sk;  
WOLFSSL_X509_STORE_CTX* ctx;  
sk = wolfSSL_X509_STORE_CTX_get_chain(ctx);  
// sk が NULL でないか確認してから使用。使用後は sk を解放する必要があります
```

C.52.2.150 function wolfSSL_X509_STORE_set_flags

```
int wolfSSL_X509_STORE_set_flags(  
    WOLFSSL_X509_STORE * store,  
    unsigned long flag  
)
```

この関数は渡された WOLFSSL_X509_STORE 構造体の動作を変更するためのフラグを受け取ります。使用されるフラグの例として WOLFSSL_CRL_CHECK があります。

Parameters:

- **str** フラグを設定する証明書ストア。
- **flag** 動作のためのフラグ。

See:

- wolfSSL_X509_STORE_new
- wolfSSL_X509_STORE_free

Return:

- SSL_SUCCESS フラグの設定時にエラーが発生しなかった場合。
- <0 失敗時に負の値が返されます。

Example

```
WOLFSSL_X509_STORE* str;  
int ret;  
// str を作成して設定  
ret = wolfSSL_X509_STORE_set_flags(str, WOLFSSL_CRL_CHECKALL);  
if (ret != SSL_SUCCESS) {  
    //ret 値を確認してエラーケースを処理する  
}
```

C.52.2.151 function wolfSSL_X509_notBefore

```
const byte * wolfSSL_X509_notBefore(  
    WOLFSSL_X509 * x509  
)
```

この関数は、バイト配列としてエンコードされた証明書の「not before」有効期限を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- byte notBeforeData を含むバイトが返されます。

Example


```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                             DYNAMIC_TYPE_X509);
...
byte* notBeforeData = wolfSSL_X509_notBefore(x509);
```

C.52.2.152 function wolfSSL_X509_notAfter

```
const byte * wolfSSL_X509_notAfter(
    WOLFSSL_X509 * x509
)
```

この関数は、バイト配列としてエンコードされた証明書の「not after」有効期限を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- byte notAfterData を含むバイトが返されます。

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                             DYNAMIC_TYPE_X509);
...
byte* notAfterData = wolfSSL_X509_notAfter(x509);
```

C.52.2.153 function wolfSSL_ASN1_INTEGER_to_BN

```
WOLFSSL_BIGNUM * wolfSSL_ASN1_INTEGER_to_BN(
    const WOLFSSL_ASN1_INTEGER * ai,
    WOLFSSL_BIGNUM * bn
)
```

この関数は、WOLFSSL_ASN1_INTEGER 値を WOLFSSL_BIGNUM 構造体にコピーするために使用されます。

Parameters:

- **ai** コピー元の WOLFSSL_ASN1_INTEGER 構造体。
- **bn** 既存の WOLFSSL_BIGNUM 構造体にコピーしたい場合は、そのポインタを渡します。オプションとして、これを NULL にすることで新しい WOLFSSL_BIGNUM 構造体を作成されます。

See: none

Return:

- pointer WOLFSSL_ASN1_INTEGER 値のコピーに成功すると、WOLFSSL_BIGNUM ポインタが返されます。
- Null 失敗時には Null が返されます。

Example

```
WOLFSSL_ASN1_INTEGER* ai;
WOLFSSL_BIGNUM* bn;
// ai を作成
bn = wolfSSL_ASN1_INTEGER_to_BN(ai, NULL);

// または既に bn を作成済みで構造体を再利用したい場合
// wolfSSL_ASN1_INTEGER_to_BN(ai, bn);
// bn または戻り値が NULL でないことを確認
```

C.52.2.154 function wolfSSL_CTX_add_extra_chain_cert

```
long wolfSSL_CTX_add_extra_chain_cert(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509 * x509
)
```

この関数は、WOLFSSL_CTX 構造体内で構築中の内部チェーンに証明書を追加します。

Parameters:

- **ctx** 証明書を追加する WOLFSSL_CTX 構造体。
- **x509** チェーンに追加する証明書。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)

Return:

- SSL_SUCCESS 証明書の追加に成功した後に返されます。
- SSL_FAILURE 証明書のチェーンへの追加に失敗した場合に返されます。

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL_X509* x509;
int ret;
// ctx を作成
ret = wolfSSL_CTX_add_extra_chain_cert(ctx, x509);
// ret 値を確認
```

C.52.2.155 function wolfSSL_CTX_get_read_ahead

```
int wolfSSL_CTX_get_read_ahead(
    WOLFSSL_CTX * ctx
)
```

この関数は、WOLFSSL_CTX 構造体からリードアヘッドフラグを取得して返します。

Parameters:

- **ctx** リードアヘッドフラグを取得する WOLFSSL_CTX 構造体。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)
- [wolfSSL_CTX_set_read_ahead](#)

Return:

- flag 成功時にはリードアヘッドフラグが返されます。
- SSL_FAILURE ctx が NULL の場合、SSL_FAILURE が返されます。

Example

```
WOLFSSL_CTX* ctx;
int flag;
// ctx をセットアップ
flag = wolfSSL_CTX_get_read_ahead(ctx);
// flag を確認
```

C.52.2.156 function wolfSSL_CTX_set_read_ahead

```
int wolfSSL_CTX_set_read_ahead(
    WOLFSSL_CTX * ctx,
    int v
)
```

この関数は、WOLFSSL_CTX 構造体内のリードアヘッドフラグを設定します。

Parameters:

- **ctx** リードアヘッドフラグを設定する WOLFSSL_CTX 構造体。
- **v** リードアヘッドフラグ。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)
- [wolfSSL_CTX_get_read_ahead](#)

Return:

- SSL_SUCCESS ctx のリードアヘッドフラグが設定された場合。
- SSL_FAILURE ctx が NULL の場合、SSL_FAILURE が返されます。

Example

```
WOLFSSL_CTX* ctx;
int flag;
int ret;
// ctx をセットアップ
ret = wolfSSL_CTX_set_read_ahead(ctx, flag);
// 戻り値を確認
```

C.52.2.157 function wolfSSL_CTX_set_tlsext_status_arg

```
long wolfSSL_CTX_set_tlsext_status_arg(
    WOLFSSL_CTX * ctx,
    void * arg
)
```

この関数は、OCSP で使用するオプション引数を設定します。

Parameters:

- **ctx** ユーザー引数を設定する WOLFSSL_CTX 構造体。
- **arg** ユーザー引数。
- **ctx** WOLFSSL_CTX オブジェクト。
- **arg** コールバックに渡すユーザー引数。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)
- [wolfSSL_CTX_set_tlsext_status_cb](#)

Return:

- SSL_FAILURE ctx またはその証明書マネージャが NULL の場合。
- SSL_SUCCESS 正常に設定された場合。
- SSL_SUCCESS 成功時、それ以外の場合は SSL_FAILURE。

OCSP ステータスコールバックに渡される引数を設定します。

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// ctx をセットアップ
ret = wolfSSL_CTX_set_tlsext_status_arg(ctx, data);

//ret 値を確認
```

C.52.2.158 function wolfSSL_CTX_set_client_cert_cb

```
void wolfSSL_CTX_set_client_cert_cb(
    WOLFSSL_CTX * ctx,
    client_cert_cb cb
)
```

クライアント証明書と秘密鍵を選択するコールバックを設定します。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクト。
- **cb** クライアント証明書と鍵を選択するコールバック関数。

See: [wolfSSL_CTX_set_cert_cb](#)

Return: void

この関数は、ハンドシェイク中にクライアント証明書が要求されたときに呼び出されるコールバックをアプリケーションが登録できるようにします。コールバックは、使用する証明書と鍵を選択して提供できます。

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.52.2.159 function wolfSSL_CTX_set_cert_cb

```
void wolfSSL_CTX_set_cert_cb(
    WOLFSSL_CTX * ctx,
    CertSetupCallback cb,
    void * arg
)
```

汎用証明書セットアップコールバックを設定します。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクト。
- **cb** 証明書セットアップ用のコールバック関数。
- **arg** コールバックに渡すユーザー引数。

See: [wolfSSL_CTX_set_client_cert_cb](#)

Return: void

この関数は、証明書セットアップ中に呼び出されるコールバックをアプリケーションが登録できるようにします。コールバックは、カスタム証明書選択またはロードロジックを実行できます。

Example

```
int my_cert_setup_cb(WOLFSSL* ssl, void* arg) { ... }
wolfSSL_CTX_set_cert_cb(ctx, my_cert_setup_cb, NULL);
```

C.52.2.160 function wolfSSL_CTX_set_tlsext_status_cb

```
int wolfSSL_CTX_set_tlsext_status_cb(
    WOLFSSL_CTX * ctx,
    tlsextStatusCb cb
)
```

OCSP ステータス要求 (OCSP ステープリング) を処理するために使用されるコールバックを設定します。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクト。
- **cb** OCSP ステータス要求を処理するコールバック関数。

See:

- [wolfSSL_CTX_get_tlsext_status_cb](#)
- [wolfSSL_CTX_set_tlsext_status_arg](#)

Return: SSL_SUCCESS 成功時、それ以外の場合は SSL_FAILURE。

この関数は、TLS ハンドシェイク中に OCSP ステータス要求を受信したときに呼び出されるコールバックをアプリケーションが登録できるようにします。コールバックは、ハンドシェイクにステープルされる OCSP レスポンスを提供できます。この API はサーバー側でのみ有効です。

Example

```
int my_ocsp_status_cb(WOLFSSL* ssl, void* arg) { ... }
wolfSSL_CTX_set_tlsext_status_cb(ctx, my_ocsp_status_cb);
```

C.52.2.161 function wolfSSL_CTX_get_tlsext_status_cb

```
int wolfSSL_CTX_get_tlsext_status_cb(
    WOLFSSL_CTX * ctx,
    tlsextStatusCb * cb
)
```

コンテキストに現在設定されている OCSP ステータスコールバックを取得します。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクト。
- **cb** コールバック関数を受け取るポインタ。

See: [wolfSSL_CTX_set_tlsext_status_cb](#)

Return: SSL_SUCCESS 成功時、それ以外の場合は SSL_FAILURE。

C.52.2.162 function wolfSSL_get_tlsexp_status_ocsp_resp

```
long wolfSSL_get_tlsexp_status_ocsp_resp(  
    WOLFSSL * ssl,  
    unsigned char ** resp  
)
```

ピアに送信 (ステープル) される OCSP レスポンスを取得します。

Parameters:

- **ssl** WOLFSSL セッション。
- **resp** レスポンスバッファを受け取るポインタ。

See: [wolfSSL_set_tlsexp_status_ocsp_resp](#)

Return: Length レスポンスの長さ、またはエラー時は負の値。

C.52.2.163 function wolfSSL_set_tlsexp_status_ocsp_resp

```
long wolfSSL_set_tlsexp_status_ocsp_resp(  
    WOLFSSL * ssl,  
    unsigned char * resp,  
    int len  
)
```

ピアに送信 (ステープル) される OCSP レスポンスを設定します。

Parameters:

- **ssl** WOLFSSL セッション。
- **resp** レスポンスバッファへのポインタ。
- **len** レスポンスバッファの長さ。

See: [wolfSSL_get_tlsexp_status_ocsp_resp](#)

Return: SSL_SUCCESS 成功時、それ以外の場合は SSL_FAILURE。

resp のバッファは wolfSSL によって所有され、wolfSSL によって解放されます。アプリケーションは、この関数を呼び出した後、バッファを解放してはいけません。

C.52.2.164 function wolfSSL_set_tlsexp_status_ocsp_resp_multi

```
int wolfSSL_set_tlsexp_status_ocsp_resp_multi(  
    WOLFSSL * ssl,  
    unsigned char * resp,  
    int len,  
    word32 idx  
)
```

TLS マルチ証明書チェーン用の複数の OCSP レスポンスを設定します。

Parameters:

- **ssl** WOLFSSL セッション。
- **resp** レスポンスバッファへのポインタ。
- **len** レスポンスバッファの長さ。
- **idx** 証明書チェーンのインデックス。

Return: SSL_SUCCESS 成功時、それ以外の場合は SSL_FAILURE。

resp のバッファは wolfSSL によって所有され、wolfSSL によって解放されます。アプリケーションは、この関数を呼び出した後、バッファを解放してはいけません。

C.52.2.165 function wolfSSL_CTX_set_ocsp_status_verify_cb

```
void wolfSSL_CTX_set_ocsp_status_verify_cb(
    WOLFSSL_CTX * ctx,
    ocspVerifyStatusCb cb,
    void * cbArg
)
```

OCSP ステータスレスポンスを検証するコールバックを設定します。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクト。
- **cb** コールバック関数。
- **cbArg** コールバックに渡すユーザー引数。

Return: void

OCSP 検証中にピアの証明書チェーンにアクセスできるようにするため、SESSION_CERTS を有効にすることを推奨します。

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;
    }
    // err != 0 の場合、手動 OCSP ステープル検証
    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)
            goto cleanup;

        for (i = idx + 1; i < wolfSSL_get_chain_count(peerCerts); i++) {
            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.52.2.166 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
)

```

この関数は、PRF コールバックに渡されるオプション引数を設定します。

Parameters:

- **ctx** ユーザ引数を設定する WOLFSSL_CTX 構造体。
- **arg** ユーザ引数。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)

Return:

- SSL_FAILURE ctx が NULL の場合。
- SSL_SUCCESS 正常に設定された場合。

Example

```

WOLFSSL_CTX* ctx;
void* data;
int ret;
// ctx をセットアップ

```



```
ret = wolfSSL_CTX_set_tlsext_opaques_prf_input_callback_arg(ctx, data);
//ret 値を確認
```

C.52.2.167 function wolfSSL_CTX_set_options

```
long wolfSSL_CTX_set_options(
    WOLFSSL_CTX * ctx,
    long opt
)
```

この関数は、指定された WOLFSSL_CTX のオプションマスクに opt のビットを OR で追加します。設定したオプションマスクは、本コンテキストから後に作成されるすべての WOLFSSL セッションに継承されます。ビットは累積されるため、特定のオプションを解除するには `wolfSSL_CTX_clear_options()` を使用してください。OpenSSL 形式の "SSL_OP_" マクロは対応する "WOLFSSL_OP_" 値の別名であり、どちらの表記でも使用できます。

Parameters:

- **ctx** オプションマスクを設定する WOLFSSL_CTX 構造体。
- **opt** 現在のマスクに OR で追加する SSL_OP_* / WOLFSSL_OP_* フラグのビットマスク。

See:

- `wolfSSL_CTX_clear_options`
- `wolfSSL_CTX_get_options`
- `wolfSSL_set_options`
- `wolfSSL_get_options`

Return:

- BAD_FUNC_ARG ctx が NULL の場合。
- 成功時、ctx に格納された更新後のオプションマスク値。

実効を持つオプション:

マクロ	効果
SSL_OP_NO_SSLv2	SSLv2 を無効化 (wolfSSL は SSLv2 を一切サポートしない。OpenSSL 互換のため受け付けるのみ)
SSL_OP_NO_SSLv3	SSLv3 を無効化
SSL_OP_NO_TLSv1	TLS 1.0 を無効化
SSL_OP_NO_TLSv1_1	TLS 1.1 を無効化
SSL_OP_NO_TLSv1_2	TLS 1.2 を無効化
SSL_OP_NO_TLSv1_3	TLS 1.3 を無効化 (WOLFSSL_TLS13 が必要)
SSL_OP_NO_COMPRESSION	レコード層圧縮を無効化 (HAVE_LIBZ 未定義時は何もしない)
SSL_OP_NO_TICKET	RFC 5077 セッションチケットを無効化 (TLS 1.2 のみ。TLS 1.3 では本フラグは無視される)。HAVE_SESSION_TICKET と OPENSSL_EXTRA / HAVE_WEBSERVER / WOLFSSL_WPAS_SMALL のいずれかが必要
SSL_OP_NO_RENEGOTIATION	相手主導の再ネゴシエーションを拒否
SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION	再ネゴシエーション時のセッション再利用を無効化
SSL_OP_COOKIE_EXCHANGE	HelloVerifyRequest クッキー交換を有効化 (DTLS では既定で有効)
SSL_OP_NO_QUERY_MTU	DTLS: 経路 MTU を問い合わせない
SSL_OP_CIPHER_SERVER_PREFERENCE	サーバ側の暗号スイート優先順位をクライアントの順序より優先

マクロ	効果
SSL_OP_SINGLE_DH_USE	ハンドシェイクごとに新しい DH 鍵を生成
SSL_OP_SINGLE_ECDH_USE	ハンドシェイクごとに新しい ECDH 鍵を生成
SSL_OP_EPHEMERAL_RSA	エフェメラル RSA を使用（レガシー。OpenSSL 互換のため受け付けるのみ）
SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS	レコード前の空フラグメント挿入を抑制（CBC BEAST 回避策）
SSL_OP_PKCS1_CHECK_1 / _2	OpenSSL 互換のため受け付けるのみ
SSL_OP_LEGACY_SERVER_CONNECT	安全でない初回接続を常に許可。定義値は 0 のため実質無効化（openssl/ssl.h 互換ヘッダが必要）

便宜マクロ・バグ回避フラグ（すべて SSL_OP_ALL の構成要素。OpenSSL 互換のため受け付けるが wolfSSL では実質 no-op）：

- SSL_OP_ALL（下記バグ回避フラグのビット OR）
- 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

複合マクロ：

- 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 には ctx に蓄積されたオプション値が反映される
```

C.52.2.168 function wolfSSL_set_options

```
long wolfSSL_set_options(
    WOLFSSL * s,
    long op
)
```

この関数は、指定された WOLFSSL セッションのオプションマスクに op のビットを OR で追加します。認識される "SSL_OP_" / "WOLFSSL_OP_" フラグの集合は `wolfSSL_CTX_set_options()` の説明を参照してください。親 WOLFSSL_CTX から継承したフラグはそのまま保持されます。本関数で SSL_OP_NO_TLSv1_3 を設定した場合、当該セッションのネゴシエーション可能なバージョンは TLS 1.2 まで引き下げられます。

Parameters:

- **s** オプションマスクを設定する WOLFSSL セッション。

- **op** 現在のマスクに OR で追加する SSL_OP_* / WOLFSSL_OP_* フラグのビットマスク。

See:

- `wolfSSL_CTX_set_options`
- `wolfSSL_clear_options`
- `wolfSSL_get_options`
- `wolfSSL_new`
- `wolfSSL_free`

Return: 成功時、s に格納された更新後のオプションマスク値。s が NULL の場合は 0 を返します。

Example

```
WOLFSSL* ssl;
long mask;
mask = wolfSSL_set_options(ssl, SSL_OP_NO_TLSv1_3);
// mask には ssl に蓄積されたオプション値が反映される
```

C.52.2.169 function `wolfSSL_get_options`

```
long wolfSSL_get_options(
    const WOLFSSL * s
)
```

この関数は、現在のオプションマスクを返します。

Parameters:

- **s** オプションマスクを取得する WOLFSSL 構造体。

See:

- `wolfSSL_new`
- `wolfSSL_free`
- `wolfSSL_set_options`

Return: val ssl に格納されているマスク値を返します。

Example

```
WOLFSSL* ssl;
unsigned long mask;
mask = wolfSSL_get_options(ssl);
// mask を確認
```

C.52.2.170 function `wolfSSL_set_tlsext_debug_arg`

```
long wolfSSL_set_tlsext_debug_arg(
    WOLFSSL * s,
    void * arg
)
```

これは、渡されるデバッグ引数を設定するために使用されます。

Parameters:

- **s** 引数を設定する WOLFSSL 構造体。
- **arg** 使用する引数。

See:

- `wolfSSL_new`

- `wolfSSL_free`

Return:

- SSL_SUCCESS 引数の設定に成功した場合。
- SSL_FAILURE NULL の ssl が渡された場合。

Example

```
WOLFSSL* ssl;
void* args;
int ret;
// ssl オブジェクトを作成
ret = wolfSSL_set_tlsext_debug_arg(ssl, args);
// ret 値を確認
```

C.52.2.171 function wolfSSL_set_tlsext_status_type

```
long wolfSSL_set_tlsext_status_type(
    WOLFSSL * s,
    int type
)
```

この関数は、クライアントアプリケーションがサーバに OCSP ステータスレスポンス (OCSP ステータスレスポンスとも呼ばれます) を返送するよう要求する際に呼び出されます。現在、サポートされている唯一のタイプは TLSEXT_STATUSTYPE_ocsp です。

Parameters:

- **s** SSL_new() 関数によって作成された WOLFSSL 構造体へのポインタ。
- **type** TLSEXT_STATUSTYPE_ocsp のみがサポートされている SSL 拡張タイプ。

See:

- `wolfSSL_new`
- `wolfSSL_CTX_new`
- `wolfSSL_free`
- `wolfSSL_CTX_free`

Return:

- 1 成功時。
- 0 エラー時。

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.52.2.172 function wolfSSL_get_verify_result

```
long wolfSSL_get_verify_result(
    const WOLFSSL * ssl
)
```

これは、ピアの証明書の検証を試みた後の結果を取得するために使用されます。

Parameters:

- **ssl** 検証結果を取得する WOLFSSL 構造体。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- X509_V_OK 検証が成功した場合。
- SSL_FAILURE_NULL の ssl が渡された場合。

Example

```
WOLFSSL* ssl;
long ret;
// ハンドシェイクを試行/完了
ret = wolfSSL_get_verify_result(ssl);
// ret 値を確認
```

C.52.2.173 function wolfSSL_ERR_print_errors_fp

```
void wolfSSL_ERR_print_errors_fp(
    XFILE fp,
    int err
)
```

この関数は、wolfSSL_get_error() によって返されたエラーコードを、より人間が読みやすいエラー文字列に変換し、その文字列を出力ファイル fp に出力します。err は wolfSSL_get_error() によって返されたエラーコードであり、fp はエラー文字列が配置されるファイルです。

Parameters:

- **fp** 人間が読みやすいエラー文字列が書き込まれる出力ファイル。
- **err** [wolfSSL_get_error\(\)](#)によって返されたエラーコード。

See:

- [wolfSSL_get_error](#)
- [wolfSSL_ERR_error_string](#)
- [wolfSSL_ERR_error_string_n](#)
- [wolfSSL_load_error_strings](#)

Return: none 戻り値なし。

Example

```
int err = 0;
WOLFSSL* ssl;
FILE* fp = ...
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_print_errors_fp(fp, err);
```

C.52.2.174 function wolfSSL_ERR_print_errors_cb

```
void wolfSSL_ERR_print_errors_cb(
    int (*)(const char *str, size_t len, void *u) cb,
```

```
void * u
)
```

この関数は、提供されたコールバックを使用してエラーレポートを処理します。コールバック関数は、各エラー行に対して実行されます。文字列、長さ、およびユーザデータがコールバックパラメータに渡されます。

Parameters:

- **cb** コールバック関数。
- **u** コールバック関数に渡すユーザデータ。

See:

- [wolfSSL_get_error](#)
- [wolfSSL_ERR_error_string](#)
- [wolfSSL_ERR_error_string_n](#)
- [wolfSSL_load_error_strings](#)

Return: none 戻り値なし。

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.52.2.175 function wolfSSL_CTX_set_psk_client_callback

```
void wolfSSL_CTX_set_psk_client_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_client_callback cb
)
```

この関数は、WOLFSSL_CTX 構造体の client_psk_cb メンバを設定します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **cb** WOLFSSL_CTX 構造体に格納される関数ポインタである wc_psk_client_callback。戻り値は、成功時には鍵の長さ、エラー時にはゼロです。unsigned int (wc_psk_client_callback) PSK クライアントコールバックのパラメータ: WOLFSSL ssl - wolfSSL 構造体へのポインタ const char* hint - ユーザへのヒントを表示するために使用できる格納された文字列。char* identity - ID がここに格納されます。unsigned int id_max_len - ID バッファのサイズ。unsigned char* key - 鍵がここに格納されます。unsigned int key_max_len - 鍵の最大サイズ。

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 戻り値なし。

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.52.2.176 function wolfSSL_set_psk_client_callback

```

void wolfSSL_set_psk_client_callback(
    WOLFSSL * ssl,
    wc_psk_client_callback cb
)

```

PSK クライアント側コールバックを設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** `wc_psk_client_callback` 型への関数ポインタ。戻り値は、成功時には鍵の長さ、エラー時にはゼロです。unsigned int (wc_psk_client_callback) PSK クライアントコールバックのパラメータ: WOLFSSL ssl - wolfSSL 構造体へのポインタ const char* hint - ユーザへのヒントを表示するために使用できる格納された文字列。char* identity - ID がここに格納されます。unsigned int id_max_len - ID バッファのサイズ。unsigned char* key - 鍵がここに格納されます。unsigned int key_max_len - 鍵の最大サイズ。

See:

- [wolfSSL_CTX_set_psk_client_callback](#)
- [wolfSSL_CTX_set_psk_server_callback](#)
- [wolfSSL_set_psk_server_callback](#)

Return: none 戻り値なし。

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 {
        // コールバックを設定できませんでした。
    }
}

```

C.52.2.177 function wolfSSL_get_psk_identity_hint

```

const char * wolfSSL_get_psk_identity_hint(
    const WOLFSSL *
)

```

この関数は、PSK アイデンティティヒントを返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See: [wolfSSL_get_psk_identity](#)

Return:

- pointer WOLFSSL 構造体の arrays メンバに格納されていた値への const char 型ポインタが返されます。
- NULL WOLFSSL または Arrays 構造体が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char* idHint;
...
idHint = wolfSSL_get_psk_identity_hint(ssl);
if(idHint){
    // ヒントが取得されました。
    return idHint;
} else {
    // ヒントの取得に成功しませんでした。
}
```

C.52.2.178 function wolfSSL_get_psk_identity

```
const char * wolfSSL_get_psk_identity(
    const WOLFSSL *
```

この関数は、Arrays 構造体の client_identity メンバへの定数ポインタを返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_get_psk_identity_hint](#)
- [wolfSSL_use_psk_identity_hint](#)

Return:

- string Arrays 構造体の client_identity メンバの文字列値。
- NULL WOLFSSL 構造体が NULL、または WOLFSSL 構造体の Arrays メンバが NULL の場合。

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){
    // pskID に値がありません
}
```

C.52.2.179 function wolfSSL_CTX_use_psk_identity_hint

```
int wolfSSL_CTX_use_psk_identity_hint(
    WOLFSSL_CTX * ctx,
    const char * hint
)
```

この関数は、hint 引数を WOLFSSL_CTX 構造体の server_hint メンバに格納します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **hint** WOLFSSL_CTX 構造体にコピーされる定数 char ポインタ。

See: `wolfSSL_use_psk_identity_hint`

Return: SSL_SUCCESS 関数の実行が成功した場合に返されます。

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){
    // 関数は成功しました
return ret;
} else {
    // 失敗ケース
}
```

C.52.2.180 function `wolfSSL_use_psk_identity_hint`

```
int wolfSSL_use_psk_identity_hint(
    WOLFSSL * ssl,
    const char * hint
)
```

この関数は、hint 引数を WOLFSSL 構造体内の Arrays 構造体の server_hint メンバに格納します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **hint** メモリに保存されるヒントを保持する定数文字ポインタ。

See: `wolfSSL_CTX_use_psk_identity_hint`

Return:

- SSL_SUCCESS hint が WOLFSSL 構造体に正常に格納された場合に返されます。
- SSL_FAILURE WOLFSSL または Arrays 構造体が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* hint;
...
if(wolfSSL_use_psk_identity_hint(ssl, hint) != SSL_SUCCESS){
    // 失敗ケースを処理
}
```

C.52.2.181 function `wolfSSL_CTX_set_psk_server_callback`

```
void wolfSSL_CTX_set_psk_server_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_server_callback cb
)
```

この関数は、WOLFSSL_CTX 構造体にサーバ側の psk コールバックを設定します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** コールバック用の関数ポインタで、WOLFSSL_CTX 構造体に格納されます。戻り値は成功時は鍵の長さ、エラー時は 0 です。unsigned int (wc_psk_server_callback) PSK サーバコールバックのパラメータ WOLFSSL ssl - wolfSSL 構造体へのポインタ char* identity - ID がここに格納されます unsigned char* key - 鍵がここに格納されます unsigned int key_max_len - 鍵の最大サイズ

See:

- wc_psk_server_callback
- `wolfSSL_set_psk_client_callback`
- `wolfSSL_set_psk_server_callback`
- `wolfSSL_CTX_set_psk_client_callback`

Return: none 戻り値なし。

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)
{
    // 関数本体
}
...
if(ctx != NULL){
    wolfSSL_CTX_set_psk_server_callback(ctx, my_psk_server_cb);
} else {
    // CTX オブジェクトが適切に初期化されませんでした
}
```

C.52.2.182 function `wolfSSL_set_psk_server_callback`

```
void wolfSSL_set_psk_server_callback(
    WOLFSSL * ssl,
    wc_psk_server_callback cb
)
```

WOLFSSL 構造体の options メンバを設定することにより、サーバ側の psk コールバックを設定します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** コールバック用の関数ポインタで、WOLFSSL 構造体に格納されます。戻り値は成功時は鍵の長さ、エラー時は 0 です。unsigned int (wc_psk_server_callback) PSK サーバコールバックのパラメータ WOLFSSL ssl - wolfSSL 構造体へのポインタ char* identity - ID がここに格納されます unsigned char* key - 鍵がここに格納されます unsigned int key_max_len - 鍵の最大サイズ

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 戻り値なし。

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)
{
    // 関数本体
}
...
if(ssl != NULL && cb != NULL){
    wolfSSL_set_psk_server_callback(ssl, my_psk_server_cb);
}
```

C.52.2.183 function wolfSSL_set_psk_callback_ctx

```
int wolfSSL_set_psk_callback_ctx(
    WOLFSSL * ssl,
    void * psk_ctx
)
```

WOLFSSL 構造体の options メンバに PSK ユーザコンテキストを設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **psk_ctx** ユーザ PSK コンテキストへの void ポインタ。

See:

- [wolfSSL_get_psk_callback_ctx](#)
- [wolfSSL_CTX_set_psk_callback_ctx](#)
- [wolfSSL_CTX_get_psk_callback_ctx](#)

Return: WOLFSSL_SUCCESS または WOLFSSL_FAILURE

C.52.2.184 function wolfSSL_CTX_set_psk_callback_ctx

```
int wolfSSL_CTX_set_psk_callback_ctx(
    WOLFSSL_CTX * ctx,
    void * psk_ctx
)
```

WOLFSSL_CTX 構造体に PSK ユーザコンテキストを設定します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **psk_ctx** ユーザ PSK コンテキストへの void ポインタ。

See:

- [wolfSSL_set_psk_callback_ctx](#)
- [wolfSSL_get_psk_callback_ctx](#)
- [wolfSSL_CTX_get_psk_callback_ctx](#)

Return: WOLFSSL_SUCCESS または WOLFSSL_FAILURE

C.52.2.185 function wolfSSL_get_psk_callback_ctx

```
void * wolfSSL_get_psk_callback_ctx(  
    WOLFSSL * ssl  
)
```

WOLFSSL 構造体の options メンバから PSK ユーザコンテキストを取得します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_set_psk_callback_ctx](#)
- [wolfSSL_CTX_set_psk_callback_ctx](#)
- [wolfSSL_CTX_get_psk_callback_ctx](#)

Return: ユーザ PSK コンテキストへの void ポインタ。

C.52.2.186 function wolfSSL_CTX_get_psk_callback_ctx

```
void * wolfSSL_CTX_get_psk_callback_ctx(  
    WOLFSSL_CTX * ctx  
)
```

WOLFSSL_CTX 構造体から PSK ユーザコンテキストを取得します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- [wolfSSL_CTX_set_psk_callback_ctx](#)
- [wolfSSL_set_psk_callback_ctx](#)
- [wolfSSL_get_psk_callback_ctx](#)

Return: ユーザ PSK コンテキストへの void ポインタ。

C.52.2.187 function wolfSSL_CTX_allow_anon_cipher

```
int wolfSSL_CTX_allow_anon_cipher(  
    WOLFSSL_CTX * ctx  
)
```

この関数は、コンパイル時に HAVE_ANON が定義されている場合、CTX 構造体の havAnon メンバを有効にします。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See: none

Return:

- SSL_SUCCESS 関数が正常に実行され、CTX の haveAnnon メンバが 1 に設定された場合に返されます。
- SSL_FAILURE CTX 構造体が NULL の場合に返されます。

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){
        // 失敗ケース
    }
}
#endif

```

C.52.2.188 function wolfSSLv23_server_method

```

WOLFSSL_METHOD * wolfSSLv23_server_method(
    void
)

```

wolfSSLv23_server_method()関数は、アプリケーションがサーバであり、SSL 3.0 から TLS 1.3 までのプロトコルバージョンで接続するクライアントをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい WOLFSSL_METHOD 構造体のためのメモリを割り当て、初期化します。

Parameters:

- **none** パラメータなし。

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 成功時、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- Failure XMALLOC を呼び出す際にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv23_server_method();
if (method == NULL) {
    // メソッドを取得できませんでした
}

ctx = wolfSSL_CTX_new(method);
...

```

C.52.2.189 function wolfSSL_state

```
int wolfSSL_state(
    WOLFSSL * ssl
)
```

これは、WOLFSSL 構造体の内部エラー状態を取得するために使用されます。

See:

- `wolfSSL_new`
- `wolfSSL_free`

Return:

- `wolfssl_error` ssl エラー状態を返します。通常は負の値です。
- `BAD_FUNC_ARG` ssl が NULL の場合。
- ssl 状態を取得する WOLFSSL 構造体。

Example

```
WOLFSSL* ssl;
int ret;
// ssl オブジェクトを作成
ret = wolfSSL_state(ssl);
// ret 値を確認
```

C.52.2.190 function `wolfSSL_get_peer_certificate`

```
WOLFSSL_X509 * wolfSSL_get_peer_certificate(
    WOLFSSL * ssl
)
```

この関数は、ピアの証明書を取得します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_X509_get_issuer_name`
- `wolfSSL_X509_get_subject_name`
- `wolfSSL_X509_get_isCA`

Return:

- pointer 存在する場合、WOLFSSL_X509 構造体の `peerCert` メンバへのポインタ。
- 0 ピア証明書発行者のサイズが定義されていない場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
WOLFSSL_X509* peerCert = wolfSSL_get_peer_certificate(ssl);

if(peerCert){
    // ピア証明書へのポインタ peerCert があります
}
```

C.52.2.191 function wolfSSL_want_read

```
int wolfSSL_want_read(
    WOLFSSL * ssl
)
```

この関数は、wolfSSL_get_error() を呼び出して SSL_ERROR_WANT_READ が返される場合と同様です。基礎となるエラー状態が SSL_ERROR_WANT_READ の場合、この関数は 1 を返し、それ以外の場合は 0 を返します。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。

See:

- wolfSSL_want_write
- wolfSSL_get_error

Return:

- 1 wolfSSL_get_error() が SSL_ERROR_WANT_READ を返す場合。基礎となる I/O に読み取り可能なデータがあります。
- 0 SSL_ERROR_WANT_READ エラー状態がない場合。

Example

```
int ret;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_want_read(ssl);
if (ret == 1) {
    // 基礎となる I/O に読み取り可能なデータがあります (SSL_ERROR_WANT_READ)
}
```

C.52.2.192 function wolfSSL_want_write

```
int wolfSSL_want_write(
    WOLFSSL * ssl
)
```

この関数は、wolfSSL_get_error() を呼び出して SSL_ERROR_WANT_WRITE が返される場合と同様です。基礎となるエラー状態が SSL_ERROR_WANT_WRITE の場合、この関数は 1 を返し、それ以外の場合は 0 を返します。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。

See:

- wolfSSL_want_read
- wolfSSL_get_error

Return:

- 1 wolfSSL_get_error() が SSL_ERROR_WANT_WRITE を返す場合。基礎となる SSL 接続で進行するために、基礎となる I/O にデータを書き込む必要があります。
- 0 SSL_ERROR_WANT_WRITE エラー状態がない場合。

Example

```

int ret;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_want_write(ssl);
if (ret == 1) {
    // 基礎となる I/O にデータを書き込む必要があります (SSL_ERROR_WANT_WRITE)
}

```

C.52.2.193 function wolfSSL_check_domain_name

```

int wolfSSL_check_domain_name(
    WOLFSSL * ssl,
    const char * dn
)

```

wolfSSL はデフォルトでピア証明書の有効な日付範囲と検証済み署名をチェックします。wolfSSL_connect() または wolfSSL_accept() の前にこの関数を呼び出すと、実行するチェックのリストにドメイン名チェックが追加されます。dn は、受信したピア証明書に対してチェックするドメイン名を保持します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **dn** 受信したピア証明書に対してチェックするドメイン名。

See: none

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE メモリエラーが発生した場合に返されます。

Example

```

int ret = 0;
WOLFSSL* ssl;
char* domain = (char*) "www.yassl.com";
...

ret = wolfSSL_check_domain_name(ssl, domain);
if (ret != SSL_SUCCESS) {
    // ドメイン名チェックの有効化に失敗しました
}

```

C.52.2.194 function wolfSSL_check_ip_address

```

int wolfSSL_check_ip_address(
    WOLFSSL * ssl,
    const char * ipaddr
)

```

wolfSSL_connect() の前にこの関数を呼ぶと、ピア証明書の SAN IPAddress エントリに対する IP アドレス検証を追加します。

Parameters:

- **ssl** wolfSSL_new() で作成された WOLFSSL 構造体へのポインタ。
- **ipaddr** 検証する NULL 終端 ASCII IP アドレス文字列。

See: wolfSSL_check_domain_name

Return:

- SSL_SUCCESS 成功。
- SSL_FAILURE パラメータ不正またはメモリ確保失敗。

C.52.2.195 function wolfSSL_Init

```
int wolfSSL_Init(  
    void  
)
```

使用するために wolfSSL ライブラリを初期化します。アプリケーションごとに 1 回、ライブラリへの他の呼び出しの前に呼び出す必要があります。

See: [wolfSSL_Cleanup](#)

Return:

- SSL_SUCCESS 成功した場合に返されます。
- BAD_MUTEX_E 返される可能性のあるエラーです。
- WC_INIT_E wolfCrypt 初期化エラーが返されます。

Example

```
int ret = 0;  
ret = wolfSSL_Init();  
if (ret != SSL_SUCCESS) {  
    wolfSSL ライブラリの初期化に失敗しました  
}
```

C.52.2.196 function wolfSSL_Cleanup

```
int wolfSSL_Cleanup(  
    void  
)
```

wolfSSL ライブラリをこれ以上使用しないように初期化を解除します。呼び出す必要はありませんが、ライブラリが使用したリソースを解放します。

See: [wolfSSL_Init](#)

Return:

- SSL_SUCCESS エラーなしで返されます。
- BAD_MUTEX_E mutex エラーが返されます。

Example

```
wolfSSL_Cleanup();
```

C.52.2.197 function wolfSSL_lib_version

```
const char * wolfSSL_lib_version(  
    void  
)
```

この関数は現在のライブラリバージョンを返します。

Parameters:

- **none** パラメータなし。

See: word32_wolfSSL_lib_version_hex

Return: LIBWOLFSSL_VERSION_STRING バージョンを定義する const char ポインタ。

Example

```
char version[MAXSIZE];
version = wolfSSL_KeepArrays();
...
if(version != ExpectedVersion){
    // 不一致ケースを処理
}
```

C.52.2.198 function wolfSSL_lib_version_hex

```
word32 wolfSSL_lib_version_hex(
    void
)
```

この関数は現在のライブラリバージョンを 16 進表記で返します。

Parameters:

- **none** パラメータなし。

See: wolfSSL_lib_version

Return: LIBWOLFSSL_VERSION_HEX wolfssl/version.h で定義された 16 進バージョンを返します。

Example

```
word32 libV;
libV = wolfSSL_lib_version_hex();

if(libV != EXPECTED_HEX){
    // 予期しない値の処理方法
} else {
    // libV の期待される結果
}
```

C.52.2.199 function wolfSSL_negotiate

```
int wolfSSL_negotiate(
    WOLFSSL * ssl
)
```

SSL メソッドの側面に基づいて実際の接続または受け入れを実行します。クライアント側から呼び出された場合は wolfSSL_connect() が実行され、サーバー側から呼び出された場合は wolfSSL_accept() が実行されます。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。

See:

- SSL_connect
- SSL_accept

Return:

- SSL_SUCCESS 成功した場合に返されます (注: 古いバージョンでは 0 が返されます)。

- SSL_FATAL_ERROR 基礎となる呼び出しがエラーになった場合に返されます。特定のエラーコードを取得するには wolfSSL_get_error() を使用してください。

Example

```
int ret = SSL_FATAL_ERROR;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_negotiate(ssl);
if (ret == SSL_FATAL_ERROR) {
    // SSL 確立に失敗しました
    int error_code = wolfSSL_get_error(ssl);
    ...
}
...
```

C.52.2.200 function wolfSSL_set_compression

```
int wolfSSL_set_compression(
    WOLFSSL * ssl
)
```

SSL 接続で圧縮を使用する機能をオンにします。両側で圧縮をオンにする必要があります。そうでない場合、圧縮は使用されません。zlib ライブラリが実際のデータ圧縮を実行します。ライブラリにコンパイルするには、configure システムで `-with-libz` を使用し、HAVE_LIBZ を定義してください。送信前にデータを圧縮すると送受信されるメッセージの実際のサイズは減少しますが、圧縮によって節約されるデータの量は、最も遅いネットワークを除いて、生データで送信するよりも解析に時間がかかることに注意してください。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。

See: none**Return:**

- SSL_SUCCESS 成功時。
- NOT_COMPILED_IN 圧縮サポートがライブラリにビルドされていない場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_compression(ssl);
if (ret == SSL_SUCCESS) {
    // SSL セッションの圧縮を正常に有効化しました
}
}
```

C.52.2.201 function wolfSSL_set_timeout

```
int wolfSSL_set_timeout(
    WOLFSSL * ssl,
    unsigned int to
)
```

この関数は SSL セッションのタイムアウト値を秒単位で設定します。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL オブジェクトへのポインタ。

- **to** SSL セッションタイムアウトの設定に使用される秒単位の値。

See:

- `wolfSSL_get1_session`
- `wolfSSL_set_session`

Return:

- SSL_SUCCESS セッションの設定に成功した場合に返されます。
- BAD_FUNC_ARG ssl が NULL の場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_timeout(ssl, 500);
if (ret != SSL_SUCCESS) {
    // セッションタイムアウト値の設定に失敗しました
}
...
```

C.52.2.202 function wolfSSL_CTX_set_timeout

```
int wolfSSL_CTX_set_timeout(
    WOLFSSL_CTX * ctx,
    unsigned int to
)
```

この関数は、指定された SSL コンテキストの SSL セッションのタイムアウト値を秒単位で設定します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された SSL コンテキストへのポインタ。
- **to** セッションタイムアウト値 (秒単位)。

See:

- `wolfSSL_flush_sessions`
- `wolfSSL_get1_session`
- `wolfSSL_set_session`
- `wolfSSL_get_sessionID`
- `wolfSSL_CTX_set_session_cache_mode`

Return:

- WOLFSSL_ERROR_CODE_OPENSSL が定義されている場合、成功時に以前のタイムアウト値が返されます。定義されていない場合、SSL_SUCCESS が返されます。
- BAD_FUNC_ARG 入力コンテキスト (ctx) が null の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_timeout(ctx, 500);
if (ret != SSL_SUCCESS) {
    // セッションタイムアウト値の設定に失敗しました
}
}
```

C.52.2.203 function wolfSSL_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_get_peer_chain(  
    WOLFSSL * ssl  
)
```

ピアの証明書チェーンを取得します。

Parameters:

- **ssl** 有効な WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_get_chain_count](#)
- [wolfSSL_get_chain_length](#)
- [wolfSSL_get_chain_cert](#)
- [wolfSSL_get_chain_cert_pem](#)

Return:

- chain 成功した場合、呼び出しはピアの証明書チェーンを返します。
- 0 無効な WOLFSSL ポインタが関数に渡された場合に返されます。

Example

none

C.52.2.204 function wolfSSL_get_chain_count

```
int wolfSSL_get_chain_count(  
    WOLFSSL_X509_CHAIN * chain  
)
```

ピアの証明書チェーン数を取得します。

Parameters:

- **chain** 有効な WOLFSSL_X509_CHAIN 構造体へのポインタ。

See:

- [wolfSSL_get_peer_chain](#)
- [wolfSSL_get_chain_length](#)
- [wolfSSL_get_chain_cert](#)
- [wolfSSL_get_chain_cert_pem](#)

Return:

- Success 成功した場合、呼び出しはピアの証明書チェーン数を返します。
- 0 無効な chain ポインタが関数に渡された場合に返されます。

Example

none

C.52.2.205 function wolfSSL_get_chain_length

```
int wolfSSL_get_chain_length(  
    WOLFSSL_X509_CHAIN * chain,  
    int idx  
)
```

インデックス (idx) におけるピアの ASN1.DER 証明書の長さをバイト単位で取得します。

Parameters:

- **chain** 有効な WOLFSSL_X509_CHAIN 構造体へのポインタ。
- **idx** チェーンの開始インデックス。

See:

- `wolfSSL_get_peer_chain`
- `wolfSSL_get_chain_count`
- `wolfSSL_get_chain_cert`
- `wolfSSL_get_chain_cert_pem`

Return:

- Success 成功した場合、呼び出しはインデックスによるピアの証明書の長さをバイト単位で返します。
- 0 無効な chain ポインタが関数に渡された場合に返されます。

Example

none

C.52.2.206 function `wolfSSL_get_chain_cert`

```
unsigned char * wolfSSL_get_chain_cert(  
    WOLFSSL_X509_CHAIN * chain,  
    int idx  
)
```

インデックス (idx) におけるピアの ASN1.DER 証明書を取得します。

Parameters:

- **chain** 有効な WOLFSSL_X509_CHAIN 構造体へのポインタ。
- **idx** チェーンの開始インデックス。

See:

- `wolfSSL_get_peer_chain`
- `wolfSSL_get_chain_count`
- `wolfSSL_get_chain_length`
- `wolfSSL_get_chain_cert_pem`

Return:

- Success 成功した場合、呼び出しはインデックスによるピアの証明書を返します。
- 0 無効な chain ポインタが関数に渡された場合に返されます。

Example

none

C.52.2.207 function `wolfSSL_get_chain_X509`

```
WOLFSSL_X509 * wolfSSL_get_chain_X509(  
    WOLFSSL_X509_CHAIN * chain,  
    int idx  
)
```

この関数は証明書チェーンからインデックス (idx) にあるピアの `wolfSSL_X509_certificate` を取得します。

Parameters:

- **chain** 動的メモリを使用しない SESSION_CACHE に使用される WOLFSSL_X509_CHAIN へのポインタ。
- **idx** WOLFSSL_X509 証明書のインデックス。

See:

- InitDecodedCert
- ParseCertRelative
- CopyDecodedToX509

Return: pointer WOLFSSL_X509 構造体へのポインタを返します。

返されたメモリを wolfSSL_FreeX509() を呼び出して解放することはユーザーの責任です。

Example

```
WOLFSSL_X509_CHAIN* chain = &session->chain;
int idx = 999; // idx を設定
...
WOLFSSL_X509_CHAIN ptr;
prt = wolfSSL_get_chain_X509(chain, idx);

if(ptr != NULL){
    // ptr は指定されたインデックスの証明書を含みます
    wolfSSL_FreeX509(ptr);
} else {
    // ptr は NULL です
}
```

C.52.2.208 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
)
```

インデックス (idx) におけるピアの PEM 証明書を取得します。

Parameters:

- **chain** 有効な WOLFSSL_X509_CHAIN 構造体へのポインタ。
- **idx** チェーンの開始インデックス。

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert

Return:

- Success 成功した場合、呼び出しはインデックスによるピアの証明書を返します。
- 0 無効な chain ポインタが関数に渡された場合に返されます。

Example

none

C.52.2.209 function wolfSSL_get_sessionID

```
const unsigned char * wolfSSL_get_sessionID(  
    const WOLFSSL_SESSION * s  
)
```

セッション ID を取得します。セッション ID は常に 32 バイト長です。

Parameters:

- **session** 有効な wolfssl セッションへのポインタ。

See: SSL_get_session

Return: id セッション ID。

Example

none

C.52.2.210 function wolfSSL_X509_get_serial_number

```
int wolfSSL_X509_get_serial_number(  
    WOLFSSL_X509 * x509,  
    unsigned char * in,  
    int * inOutSz  
)
```

ピアの証明書シリアル番号を取得します。シリアル番号バッファ (in) は少なくとも 32 バイト長であり、入力として inOutSz 引数として提供される必要があります。関数を呼び出した後、inOutSz には in バッファに書き込まれた実際の長さ (バイト単位) が保持されます。

Parameters:

- **in** シリアル番号バッファで、少なくとも 32 バイト長である必要があります。
- **inOutSz** in バッファに書き込まれた実際の長さ (バイト単位) を保持します。

See: SSL_get_peer_certificate

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG 不正な関数引数が検出された場合に返されます。

Example

none

C.52.2.211 function wolfSSL_X509_get_subjectCN

```
char * wolfSSL_X509_get_subjectCN(  
    WOLFSSL_X509 *  
)
```

証明書からサブジェクトのコモンネームを返します。

Parameters:

- **x509** 証明書情報を含む WOLFSSL_X509 構造体へのポインタ。

See:

- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name

- `wolfSSL_X509_get_subject_name`

Return:

- NULL x509 構造体が null の場合に返されます。
- string 成功時にはサブジェクトのコモンネームの文字列表現が返されます。

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                           DYNAMIC_TYPE_X509);

...
int x509Cn = wolfSSL_X509_get_subjectCN(x509);
if(x509Cn == NULL){
    // NULL ケースを処理
} else {
    // x509Cn にはコモンネームが含まれる
}
```

C.52.2.212 function wolfSSL_X509_get_der

```
const unsigned char * wolfSSL_X509_get_der(
    WOLFSSL_X509 * x509,
    int * outSz
)
```

この関数は、WOLFSSL_X509 構造体内の DER エンコードされた証明書を取得します。

Parameters:

- **x509** 証明書情報を含む WOLFSSL_X509 構造体へのポインタ。
- **outSz** WOLFSSL_X509 構造体の derBuffer メンバの長さ。

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 この関数は、DerBuffer 構造体の buffer メンバを返します。これは byte 型です。
- NULL x509 または outSz パラメータが NULL の場合に返されます。

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                           DYNAMIC_TYPE_X509);

int* outSz; // 初期化
...
byte* x509Der = wolfSSL_X509_get_der(x509, outSz);
if(x509Der == NULL){
    // 失敗ケース、パラメータの 1 つが NULL だった
}
```

C.52.2.213 function wolfSSL_X509_get_notAfter

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notAfter(
    WOLFSSL_X509 *
)
```

この関数は、x509 が NULL であるかどうかをチェックし、そうでない場合は x509 構造体の notAfter メンバを返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

See: [wolfSSL_X509_get_notBefore](#)

Return:

- pointer x509 構造体の notAfter メンバへの ASN1_TIME を持つ構造体へのポインタ。
- NULL x509 オブジェクトが NULL の場合に返されます。

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){
    // 失敗ケース、x509 オブジェクトが null
}
```

C.52.2.214 function wolfSSL_X509_version

```
int wolfSSL_X509_version(
    WOLFSSL_X509 * x509
)
```

この関数は、X509 証明書のバージョンを取得します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_X509_get_subject_name](#)
- [wolfSSL_X509_get_issuer_name](#)
- [wolfSSL_X509_get_isCA](#)
- [wolfSSL_get_peer_certificate](#)

Return:

- 0 x509 構造体が NULL の場合に返されます。
- version x509 構造体に格納されているバージョンが返されます。

Example

```
WOLFSSL_X509* x509;
int version;
...
version = wolfSSL_X509_version(x509);
if(!version){
    // 関数は 0 を返した、失敗ケース
}
```

C.52.2.215 function wolfSSL_X509_d2i_fp

```
WOLFSSL_X509 * wolfSSL_X509_d2i_fp(
    WOLFSSL_X509 ** x509,
    FILE * file
)
```

NO_STDIO_FILESYSTEM が定義されている場合、この関数はヒープメモリを割り当て、WOLFSSL_X509 構造体を初期化し、そのポインタを返します。

Parameters:

- **x509** WOLFSSL_X509 ポインタへのポインタ。
- **file** FILE へのポインタである定義された型。

See:

- wolfSSL_X509_d2i
- XFTELL
- XREWIND
- XFSEEK

Return:

- *WOLFSSL_X509 関数が正常に実行された場合、WOLFSSL_X509 構造体ポインタが返されます。
- NULL XFTELL マクロの呼び出しが負の値を返した場合。

Example

```
WOLFSSL_X509* x509a = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
WOLFSSL_X509** x509 = x509a;
XFILE file; // (struct fs_file* にマップされる)
...
WOLFSSL_X509* newX509 = wolfSSL_X509_d2i_fp(x509, file);
if(newX509 == NULL){
    // 関数は NULL を返した
}
```

C.52.2.216 function wolfSSL_X509_load_certificate_file

```
WOLFSSL_X509 * wolfSSL_X509_load_certificate_file(
    const char * fname,
    int format
)
```

この関数は、x509 証明書をメモリにロードします。

Parameters:

- **fname** ロードする証明書ファイル。
- **format** 証明書のフォーマット。

See:

- InitDecodedCert
- PemToDer
- wolfSSL_get_certificate
- AssertNotNull

Return:

- pointer 正常に実行されると、WOLFSSL_X509 構造体へのポインタが返されます。

- NULL 証明書を書き込むことができなかった場合に返されます。

Example

```
#define cliCert    "certs/client-cert.pem"
...
X509* x509;
...
x509 = wolfSSL_X509_load_certificate_file(cliCert, SSL_FILETYPE_PEM);
AssertNotNull(x509);
```

C.52.2.217 function wolfSSL_X509_get_device_type

```
unsigned char * wolfSSL_X509_get_device_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

この関数は、x509 構造体からデバイスタイプをバッファにコピーします。

Parameters:

- **x509** WOLFSSL_X509_new() で作成された WOLFSSL_X509 構造体へのポインタ。
- **in** デバイスタイプ (バッファ) を保持するバイト型へのポインタ。
- **inOutSz** パラメータ inOutSz または x509 構造体の deviceTypeSz メンバのいずれか小さい方。

See:

- wolfSSL_X509_get_hw_type
- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_d2i

Return:

- pointer x509 構造体からデバイスタイプを保持するバイトポインタが返されます。
- NULL バッファサイズが NULL の場合に返されます。

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){
    // 失敗ケース、NULL が返された
}
```

C.52.2.218 function wolfSSL_X509_get_hw_type

```
unsigned char * wolfSSL_X509_get_hw_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

この関数は、WOLFSSL_X509 構造体の hwType メンバをバッファにコピーします。

Parameters:

- **x509** 証明書情報を含む WOLFSSL_X509 構造体へのポインタ。
- **in** バッファを表す byte 型へのポインタ。
- **inOutSz** バッファのサイズを表す int 型へのポインタ。

See:

- [wolfSSL_X509_get_hw_serial_number](#)
- [wolfSSL_X509_get_device_type](#)

Return:

- byte 関数は、以前に WOLFSSL_X509 構造体の hwType メンバに保持されていたデータのバイト型を返します。
- NULL inOutSz が NULL の場合に返されます。

Example

```
WOLFSSL_X509* x509; // X509 証明書
byte* in; // バッファを初期化
int* inOutSz; // バッファのサイズを保持
...
byte* hwType = wolfSSL_X509_get_hw_type(x509, in, inOutSz);

if(hwType == NULL){
    // 失敗ケース、関数は NULL を返した
}
```

C.52.2.219 function wolfSSL_X509_get_hw_serial_number

```
unsigned char * wolfSSL_X509_get_hw_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

この関数は、x509 オブジェクトの hwSerialNum メンバを返します。

Parameters:

- **x509** 証明書情報を含む WOLFSSL_X509 構造体へのポインタ。
- **in** コピー先のバッファへのポインタ。
- **inOutSz** バッファのサイズへのポインタ。

See:

- [wolfSSL_X509_get_subject_name](#)
- [wolfSSL_X509_get_issuer_name](#)
- [wolfSSL_X509_get_isCA](#)
- [wolfSSL_get_peer_certificate](#)
- [wolfSSL_X509_version](#)

Return: pointer 関数は、x509 オブジェクトからロードされたシリアル番号を含む in バッファへのバイトポインタを返します。

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){
    // 失敗ケース
}

```

C.52.2.220 function wolfSSL_connect_cert

```

int wolfSSL_connect_cert(
    WOLFSSL * ssl
)

```

この関数はクライアント側で呼び出され、ピアの証明書チェーンを取得するのに十分な長さだけサーバーとの SSL/TLS ハンドシェイクを開始します。この関数が呼び出されるとき、基礎となる通信チャンネルはすでに設定されています。wolfSSL_connect_cert() は、ブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、wolfSSL_connect_cert() は、基礎となる I/O が wolfSSL_connect_cert() がハンドシェイクを続行するために必要な処理を満たせないときに戻ります。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。呼び出しプロセスは、基礎となる I/O が準備できたときに wolfSSL_connect_cert() の呼び出しを繰り返す必要があり、wolfSSL は中断した場所から再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件をチェックできます。基礎となる I/O がブロッキング I/O の場合、wolfSSL_connect_cert() はピアの証明書チェーンが受信されたときにのみ戻ります。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE SSL セッションパラメータが NULL の場合に返されます。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出してください。

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.52.2.221 function wolfSSL_d2i_PKCS12_bio

```
WC_PKCS12 * wolfSSL_d2i_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 ** pkcs12
)
```

wolfSSL_d2i_PKCS12_bio(d2i_PKCS12_bio) は、WOLFSSL_BIO から WC_PKCS12 構造体へ PKCS12 情報をコピーします。情報は、構造体内でコンテンツ情報のリストとして分割され、オプションの MAC 情報を保持する構造体も含まれます。情報が WC_PKCS12 構造体内でチャンク (ただし復号されていない) に分割された後、呼び出すことによって解析および復号できます。

Parameters:

- **bio** PKCS12 バッファを読み取る WOLFSSL_BIO 構造体。
- **pkcs12** 作成された新しい PKCS12 構造体用の WC_PKCS12 構造体ポインタ。NULL でも可。

See:

- [wolfSSL_PKCS12_parse](#)
- [wc_PKCS12_free](#)

Return:

- WC_PKCS12 WC_PKCS12 構造体へのポインタ。
- Failure 関数が失敗した場合、NULL を返します。

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio は PKCS12 ファイルを読み込みます。
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, "a password", &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//cert、pkey、およびオプションで certs スタックを使用します。
```

C.52.2.222 function wolfSSL_i2d_PKCS12_bio

```
WC_PKCS12 * wolfSSL_i2d_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 * pkcs12
)
```

wolfSSL_i2d_PKCS12_bio(i2d_PKCS12_bio) は、WC_PKCS12 構造体から WOLFSSL_BIO へ証明書情報をコピーします。

Parameters:

- **bio** PKCS12 バッファを書き込む WOLFSSL_BIO 構造体。
- **pkcs12** 入力としての PKCS12 構造体用の WC_PKCS12 構造体。

See:

- [wolfSSL_PKCS12_parse](#)
- [wc_PKCS12_free](#)

Return:

- 1 成功時。

- 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);
//DER ファイルを内部構造に変換します。
wc_d2i_PKCS12(buffer, bytes, pkcs12);
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
//PKCS12 構造体を bio に変換します。
wolfSSL_i2d_PKCS12_bio(bio, pkcs12);
wc_PKCS12_free(pkcs);
//bio を使用します。
```

C.52.2.223 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 は、configure コマンドに `-enable-opensslextra` を追加することで有効にできます。復号にトリプル DES と RC4 を使用できるため、opensslextra を有効にする際にこれらの機能も有効にすることを推奨します (`-enable-des3 -enable-arc4`)。wolfSSL は現在 RC2 をサポートしていないため、RC2 での復号は現在利用できません。これは、OpenSSL コマンドラインで p12 ファイルを作成する際に使用されるデフォルトの暗号化スキームで顕著になる可能性があります。wolfSSL_PKCS12_parse(PKCS12_parse)。この関数が最初に行うことは、存在する場合に MAC が正しいかどうかを確認することです。MAC が失敗した場合、関数は返され、保存されているコンテンツ情報の復号を試みません。この関数は、各コンテンツ情報を解析してバッグタイプを探し、バッグタイプが既知の場合、必要に応じて復号され、構築中の証明書リストまたは見つかった鍵として保存されます。すべてのバッグを解析した後、見つかった鍵は証明書リストと比較され、一致するペアが見つかります。この一致するペアは、鍵と証明書として返されます。オプションで、見つかった証明書リストは STACK_OF の証明書として返されます。現時点では、CRL、Secret、または SafeContents バッグはスキップされ、解析されません。これらまたは他の「不明な」バッグがスキップされているかどうかは、デバッグ出力を表示することで確認できます。フレンドリ名などの追加属性は、PKCS12 ファイルを解析する際にスキップされます。

Parameters:

- **pkcs12** 解析する WC_PKCS12 構造体。
- **passwd** PKCS12 を復号するためのパスワード。
- **pkey** PKCS12 から復号された秘密鍵を保持する構造体。
- **cert** PKCS12 から復号された証明書を保持する構造体。
- **stack** 追加の証明書のオプションスタック。

See:

- [wolfSSL_d2i_PKCS12_bio](#)

- `wc_PKCS12_free`

Return:

- `SSL_SUCCESS` PKCS12 の解析が成功した場合。
- `SSL_FAILURE` エラーケースに遭遇した場合。

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio は PKCS12 ファイルを読み込みます。
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, "a password", &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//cert、pkey、およびオプションで certs スタックを使用します。
```

C.52.2.224 function wolfSSL_SetTmpDH

```
int wolfSSL_SetTmpDH(
    WOLFSSL * ssl,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

サーバ Diffie-Hellman エフェメラルパラメータの設定。この関数は、サーバが DHE を使用する暗号スイートをネゴシエートする場合に使用されるグループパラメータを設定します。

Parameters:

- `ssl` `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- `p` Diffie-Hellman 素数パラメータ。
- `pSz` `p` のサイズ。
- `g` Diffie-Hellman 「生成元」パラメータ。
- `gSz` `g` のサイズ。

See: `SSL_accept`

Return:

- `SSL_SUCCESS` 成功時。
- `MEMORY_ERROR` メモリエラーが発生した場合に返されます。
- `SIDE_ERROR` この関数が SSL サーバではなく SSL クライアントで呼び出された場合に返されます。

Example

```
WOLFSSL* ssl;
static unsigned char p[] = {...};
static unsigned char g[] = {...};
...
wolfSSL_SetTmpDH(ssl, p, sizeof(p), g, sizeof(g));
```

C.52.2.225 function wolfSSL_SetTmpDH_buffer

```
int wolfSSL_SetTmpDH_buffer(
    WOLFSSL * ssl,
    const unsigned char * b,
    long sz,
    int format
)
```

この関数は、Diffie-Hellman パラメータのラッパーである wolfSSL_SetTmpDH_buffer_wrapper を呼び出します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** wolfSSL_SetTmpDH_file_wrapper から渡される割り当てられたバッファ。
- **sz** ファイル (wolfSSL_SetTmpDH_file_wrapper 内の fname) のサイズを保持する long int 型。
- **format** wolfSSL_SetTmpDH_file_wrapper() から渡される証明書形式の表現である整数型。

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wc_DhParamsLoad
- wolfSSL_SetTmpDH
- PemToDer
- wolfSSL_CTX_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- SSL_SUCCESS 実行が成功した場合。
- SSL_BAD_FILETYPE ファイルタイプが PEM でなく、ASN.1 でもない場合。また、wc_DhParamsLoad が正常に返されなかった場合にも返されます。
- SSL_NO_PEM_HEADER PEM ヘッダが存在しない場合、PemToDer から返されます。
- SSL_BAD_FILE PemToDer でファイルエラーがあった場合に返されます。
- SSL_FATAL_ERROR コピーエラーがあった場合、PemToDer から返されます。
- MEMORY_E メモリ割り当てエラーがあった場合。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合、またはサブルーチンに NULL 引数が渡された場合に返されます。
- DH_KEY_SIZE_E wolfSSL_SetTmpDH() または wolfSSL_CTX_SetTmpDH() でキーサイズエラーがある場合に返されます。
- SIDE_ERROR wolfSSL_SetTmpDH でサーバ側でない場合に返されます。

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.52.2.226 function wolfSSL_SetTmpDH_file

```
int wolfSSL_SetTmpDH_file(
    WOLFSSL * ssl,
    const char * f,
    int format
)
```

この関数は、wolfSSL_SetTmpDH_file_wrapper を呼び出してサーバ Diffie-Hellman パラメータを設定します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **fname** 証明書を保持する定数 char 型ポインタ。
- **format** 証明書の形式を保持する整数型。

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 この関数とそのサブルーチンが正常に完了した場合に返されます。
- MEMORY_E この関数またはサブルーチンでメモリ割り当てが失敗した場合に返されます。
- SIDE_ERROR WOLFSSL 構造体内の Options 構造体の side メンバがサーバ側でない場合。
- SSL_BAD_FILETYPE 証明書が一連のチェックに失敗した場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL 構造体の minDhKeySz メンバの値より小さい場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL 構造体の maxDhKeySz メンバの値より大きい場合に返されます。
- BAD_FUNC_ARG WOLFSSL 構造体など、許可されていない引数値が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* dhParam;
...
AssertIntNE(SSL_SUCCESS,
    wolfSSL_SetTmpDH_file(ssl, dhParam, SSL_FILETYPE_PEM));
```

C.52.2.227 function wolfSSL_CTX_SetTmpDH

```
int wolfSSL_CTX_SetTmpDH(
    WOLFSSL_CTX * ctx,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

サーバ CTX Diffie-Hellman のパラメータを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **p** serverDH_P 構造体の buffer メンバに読み込まれる定数 unsigned char 型ポインタ。
- **pSz** p のサイズを表す int 型で、MAX_DH_SIZE に初期化されます。
- **g** serverDH_G 構造体の buffer メンバに読み込まれる定数 unsigned char 型ポインタ。
- **gSz** g のサイズを表す int 型で、MAX_DH_SIZE に初期化されます。

See:

- `wolfSSL_SetTmpDH`
- `wc_DhParamsLoad`

Return:

- `SSL_SUCCESS` 関数とすべてのサブルーチンがエラーなく返された場合に返されます。
- `BAD_FUNC_ARG` `ctx`、`p` または `g` パラメータが `NULL` の場合に返されます。
- `DH_KEY_SIZE_E` `DH` パラメータの鍵サイズが `WOLFSSL_CTX` 構造体の `minDhKeySz` メンバの値より小さい場合に返されます。
- `DH_KEY_SIZE_E` `DH` パラメータの鍵サイズが `WOLFSSL_CTX` 構造体の `maxDhKeySz` メンバの値より大きい場合に返されます。
- `MEMORY_E` この関数またはサブルーチンでメモリ割り当てが失敗した場合に返されます。

Example

```
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){
    // 失敗ケース
}
```

C.52.2.228 function wolfSSL_CTX_SetTmpDH_buffer

```
int wolfSSL_CTX_SetTmpDH_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * b,
    long sz,
    int format
)
```

`wolfSSL_SetTmpDH_buffer_wrapper` を呼び出すラッパー関数です。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された `WOLFSSL` 構造体へのポインタ。
- **buf** バッファとして割り当てられ、`wolfSSL_SetTmpDH_buffer_wrapper` に渡される定数 `unsigned char` 型へのポインタ。
- **sz** `wolfSSL_SetTmpDH_file_wrapper()` の `fname` パラメータから導出される `long` 整数型。
- **format** `wolfSSL_SetTmpDH_file_wrapper()` から渡される整数型。

See:

- `wolfSSL_SetTmpDH_buffer_wrapper`
- `wolfSSL_SetTmpDH_buffer`
- `wolfSSL_SetTmpDH_file_wrapper`
- `wolfSSL_CTX_SetTmpDH_file`

Return:

- 0 実行が成功した場合に返されます。
- `BAD_FUNC_ARG` `ctx` または `buf` パラメータが `NULL` の場合に返されます。
- `MEMORY_E` メモリ割り当てエラーがある場合。
- `SSL_BAD_FILETYPE` `format` が正しくない場合に返されます。

Example

```
static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
    Const char* fname, int format);
#ifdef WOLFSSL_SMALL_STACK
byte staticBuffer[1]; // ヒープ使用を強制
#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.52.2.229 function wolfSSL_CTX_SetTmpDH_file

```
int wolfSSL_CTX_SetTmpDH_file(
    WOLFSSL_CTX * ctx,
    const char * f,
    int format
)
```

この関数は、wolfSSL_SetTmpDH_file_wrapper を呼び出してサーバ Diffie-Hellman パラメータを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **fname** 証明書ファイルへの定数文字ポインタ。
- **format** 証明書形式の表現である、wolfSSL_SetTmpDH_file_wrapper() から渡される整数型。

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 wolfSSL_SetTmpDH_file_wrapper またはそのサブルーチンのいずれかが正常に返された場合に返されます。
- MEMORY_E サブルーチンで動的メモリの割り当てが失敗した場合に返されます。
- BAD_FUNC_ARG ctx または fname パラメータが NULL の場合、またはサブルーチンに NULL 引数が渡された場合に返されます。
- SSL_BAD_FILE 証明書ファイルを開けない場合、または wolfSSL_SetTmpDH_file_wrapper からのファイルに対する一連のチェックが失敗した場合に返されます。
- SSL_BAD_FILETYPE wolfSSL_SetTmpDH_buffer_wrapper() から、形式が PEM でも ASN.1 でもない場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の minDhKeySz メンバの値より小さい場合に返されます。

- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の maxDhKeySz メンバの値より大きい場合に返されます。
- SIDE_ERROR サーバ側でない場合、wolfSSL_SetTmpDH() で返されます。
- SSL_NO_PEM_HEADER PEM ヘッダがない場合、PemToDer から返されます。
- SSL_FATAL_ERROR メモリコピーの失敗がある場合、PemToDer から返されます。

Example

```
#define dhParam      "certs/dh2048.pem"
#define aASSERTiNTne(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.52.2.230 function wolfSSL_CTX_SetMinDhKey_Sz

```
int wolfSSL_CTX_SetMinDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)
```

この関数は、WOLFSSL_CTX 構造体の minDhKeySz メンバにアクセスして、Diffie-Hellman 鍵サイズの最小サイズ (ビット単位) を設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz_bits** 最小 DH 鍵サイズをビット単位で設定するために使用される word16 型。WOLFSSL_CTX 構造体はこの情報を minDhKeySz メンバに保持します。

See:

- [wolfSSL_SetMinDhKey_Sz](#)
- [wolfSSL_CTX_SetMaxDhKey_Sz](#)
- [wolfSSL_SetMaxDhKey_Sz](#)
- [wolfSSL_GetDhKey_Sz](#)
- [wolfSSL_CTX_SetTMpDH_file](#)

Return:

- SSL_SUCCESS 関数が正常に完了した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合、または keySz_bits が 16,000 より大きいか 8 で割り切れない場合に返されます。

Example

```
public static int CTX_SetMinDhKey_Sz(IntPtr ctx, short minDhKey){
    ...
    return wolfSSL_CTX_SetMinDhKey_Sz(local_ctx, minDhKeyBits);
}
```

C.52.2.231 function wolfSSL_SetMinDhKey_Sz

```
int wolfSSL_SetMinDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

WOLFSSL 構造体内の Diffie-Hellman 鍵の最小サイズ (ビット単位) を設定します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz_bits** 最小 DH 鍵サイズをビット単位で設定するために使用される word16 型。WOLFSSL_CTX 構造体はこの情報を minDhKeySz メンバに保持します。

See:

- `wolfSSL_CTX_SetMinDhKey_Sz`
- `wolfSSL_GetDhKey_Sz`

Return:

- SSL_SUCCESS 最小サイズが正常に設定されました。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合、または keySz_bits が 16,000 より大きいか 8 で割り切れない場合。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz_bits;
...
if(wolfSSL_SetMinDhKey_Sz(ssl, keySz_bits) != SSL_SUCCESS){
    // 設定に失敗しました
}
```

C.52.2.232 function wolfSSL_CTX_SetMaxDhKey_Sz

```
int wolfSSL_CTX_SetMaxDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)
```

この関数は、WOLFSSL_CTX 構造体の maxDhKeySz メンバにアクセスして、Diffie-Hellman 鍵サイズの最大サイズ (ビット単位) を設定します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz_bits** 最大 DH 鍵サイズをビット単位で設定するために使用される word16 型。WOLFSSL_CTX 構造体はこの情報を maxDhKeySz メンバに保持します。

See:

- `wolfSSL_SetMinDhKey_Sz`
- `wolfSSL_CTX_SetMinDhKey_Sz`
- `wolfSSL_SetMaxDhKey_Sz`
- `wolfSSL_GetDhKey_Sz`
- `wolfSSL_CTX_SetTMpDH_file`

Return:

- SSL_SUCCESS 関数が正常に完了した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合、または keySz_bits が 16,000 より大きいか 8 で割り切れない場合に返されます。

Example

```
public static int CTX_SetMaxDhKey_Sz(IntPtr ctx, short maxDhKey){
    ...
    return wolfSSL_CTX_SetMaxDhKey_Sz(local_ctx, keySz_bits);
}
```

C.52.2.233 function wolfSSL_SetMaxDhKey_Sz

```
int wolfSSL_SetMaxDhKey_Sz(  
    WOLFSSL * ssl,  
    word16 keySz_bits  
)
```

WOLFSSL 構造体内の Diffie-Hellman 鍵の最大サイズ (ビット単位) を設定します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz** 最大 DH 鍵のビットサイズを表す word16 型。

See:

- `wolfSSL_CTX_SetMaxDhKey_Sz`
- `wolfSSL_GetDhKey_Sz`

Return:

- SSL_SUCCESS 最大サイズが正常に設定されました。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合、または keySz パラメータが許容サイズより大きい 8 で割り切れない場合。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);  
word16 keySz;  
...  
if(wolfSSL_SetMaxDhKey(ssl, keySz) != SSL_SUCCESS){  
    // 設定に失敗しました  
}
```

C.52.2.234 function wolfSSL_GetDhKey_Sz

```
int wolfSSL_GetDhKey_Sz(  
    WOLFSSL * ssl  
)
```

options 構造体のメンバである dhKeySz の値 (ビット単位) を返します。この値は、Diffie-Hellman 鍵サイズをバイト単位で表します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_SetMinDhKey_sz`
- `wolfSSL_CTX_SetMinDhKey_Sz`
- `wolfSSL_CTX_SetTmpDH`
- `wolfSSL_SetTmpDH`
- `wolfSSL_CTX_SetTmpDH_file`

Return:

- dhKeySz ssl->options.dhKeySz に保持されている値を返します。これはビット単位のサイズを表す整数値です。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合に返されます。

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){
    // 失敗ケース
} else {
    // dhKeySz は鍵のサイズを保持しています
}

```

C.52.2.235 function wolfSSL_CTX_SetMinRsaKey_Sz

```

int wolfSSL_CTX_SetMinRsaKey_Sz(
    WOLFSSL_CTX * ctx,
    short keySz
)

```

WOLFSSL_CTX 構造体と WOLFSSL_CERT_MANAGER 構造体の両方で最小 RSA 鍵サイズを設定します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **keySz** ctx 構造体と cm 構造体の minRsaKeySz に格納され、バイトに変換される short 整数型。

See: [wolfSSL_SetMinRsaKey_Sz](#)

Return:

- SSL_SUCCESS 関数の実行が成功した場合に返されます。
- BAD_FUNC_ARG ctx 構造体が NULL の場合、または keySz がゼロ未満か 8 で割り切れない場合に返されます。

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.52.2.236 function wolfSSL_SetMinRsaKey_Sz

```

int wolfSSL_SetMinRsaKey_Sz(
    WOLFSSL * ssl,
    short keySz
)

```

WOLFSSL 構造体にある RSA の最小許容鍵サイズをビット単位で設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz** 最小鍵をビット単位で表す short 整数値。

See: [wolfSSL_CTX_SetMinRsaKey_Sz](#)

Return:

- SSL_SUCCESS 最小値が正常に設定されました。
- BAD_FUNC_ARG ssl 構造体が NULL の場合、または keySz がゼロ未満か 8 で割り切れない場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
short keySz;
...

int isSet = wolfSSL_SetMinRsaKey_Sz(ssl, keySz);
if(isSet != SSL_SUCCESS){
    設定に失敗しました
}
```

C.52.2.237 function wolfSSL_CTX_SetMinEccKey_Sz

```
int wolfSSL_CTX_SetMinEccKey_Sz(
    WOLFSSL_CTX * ctx,
    short keySz
)
```

WOLF_CTX 構造体と WOLFSSL_CERT_MANAGER 構造体で ECC 鍵の最小サイズをビット単位で設定します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **keySz** 最小 ECC 鍵サイズをビット単位で表す short 整数型。

See: [wolfSSL_SetMinEccKey_Sz](#)

Return:

- SSL_SUCCESS 実行が成功し、minEccKeySz メンバが設定された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合、または keySz が負の値か 8 で割り切れない場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
short keySz; // 最小鍵サイズ
...
if(wolfSSL_CTX_SetMinEccKey(ctx, keySz) != SSL_SUCCESS){
    // 最小鍵サイズの設定に失敗しました
}
```

C.52.2.238 function wolfSSL_SetMinEccKey_Sz

```
int wolfSSL_SetMinEccKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

options 構造体の minEccKeySz メンバの値を設定します。options 構造体は WOLFSSL 構造体のメンバであり、ssl パラメータを通じてアクセスされます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **keySz** 最小 ECC 鍵サイズを設定するために使用される値。options 構造体に値を設定します。

See:

- `wolfSSL_CTX_SetMinEccKey_Sz`
- `wolfSSL_CTX_SetMinRsaKey_Sz`
- `wolfSSL_SetMinRsaKey_Sz`

Return:

- SSL_SUCCESS 関数が options 構造体の minEccKeySz メンバを正常に設定した場合。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合、または鍵サイズ (keySz) が 0 未満か 8 で割り切れない場合。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx); // 新しいセッション
short keySz = 999; // 許容される最小鍵サイズに設定すべきです
...
if(wolfSSL_SetMinEccKey_Sz(ssl, keySz) != SSL_SUCCESS){
    // 失敗ケース
}
```

C.52.2.239 function wolfSSL_make_eap_keys

```
int wolfSSL_make_eap_keys(
    WOLFSSL * ssl,
    void * key,
    unsigned int len,
    const char * label
)
```

この関数は、マスターシークレットから鍵材料を導出するために EAP-TLS と EAP-TTLS によって使用されます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **key** p_hash 関数の結果を保持する void ポインタ変数。
- **len** key 変数の長さを表す unsigned 整数。
- **label** wc_PRF() でコピーされる定数 char ポインタ。

See:

- `wc_PRF`
- `wc_HmacFinal`
- `wc_HmacUpdate`

Return:

- BUFFER_E バッファの実際のサイズが許容される最大サイズを超えた場合に返されます。
- MEMORY_E メモリ割り当てでエラーが発生した場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
void* key;
unsigned int len;
const char* label;
```

...

```
return wolfSSL_make_eap_keys(ssl, key, len, label);
```

C.52.2.240 function wolfSSL_writev

```
int wolfSSL_writev(
    WOLFSSL * ssl,
    const struct iovec * iov,
    int iovcnt
)
```

writev のセマンティクスをシミュレートしますが、SSL_write() の動作とフロント追加が小さい場合があるため、実際には一度にブロック単位では行いません。writev を使用するソフトウェアへの移植を容易にします。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **iov** 書き込む I/O ベクトルの配列。
- **iovcnt** iov 配列内のベクトル数。

See: [wolfSSL_write](#)

Return:

- > 0 成功時、書き込まれたバイト数。
- 0 失敗時に返されます。wolfSSL_get_error() を呼び出して特定のエラーコードを取得してください。
- MEMORY_ERROR メモリエラーが発生した場合に返されます。
- SSL_FATAL_ERROR エラーが発生した場合、またはノンブロッキングソケットを使用しているときに SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーを受信し、アプリケーションが wolfSSL_write() を再度呼び出す必要がある場合に返されます。wolfSSL_get_error() を使用して特定のエラーコードを取得してください。

Example

```
WOLFSSL* ssl = 0;
char *bufA = "hello\n";
char *bufB = "hello world\n";
int iovcnt;
struct iovec iov[2];

iov[0].iov_base = bufA;
iov[0].iov_len = strlen(bufA);
iov[1].iov_base = bufB;
iov[1].iov_len = strlen(bufB);
iovcnt = 2;

...
ret = wolfSSL_writev(ssl, iov, iovcnt);
// "ret" バイトを書き込みました。または <=0 の場合はエラー
```

C.52.2.241 function wolfSSL_CTX_UnloadCAs

```
int wolfSSL_CTX_UnloadCAs(
    WOLFSSL_CTX * ctx
)
```

この関数は、CA 署名者リストをアンロードし、署名者テーブル全体を解放します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- `wolfSSL_CertManagerUnloadCAs`
- `LockMutex`
- `UnlockMutex`

Return:

- `SSL_SUCCESS` 関数の実行が成功した場合に返されます。
- `BAD_FUNC_ARG` WOLFSSL_CTX 構造体が NULL の場合、またはサブルーチンで許可されない引数値が渡された場合に返されます。
- `BAD_MUTEX_E` mutex エラーが発生した場合に返されます。 `LockMutex()` が 0 を返しませんでした。

Example

```
WOLFSSL_METHOD method = wolfTLSEv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
...
if(wolfSSL_CTX_UnloadCAs(ctx) != SSL_SUCCESS){
    // 関数は CA をアンロードしませんでした
}
```

C.52.2.242 function `wolfSSL_CTX_UnloadIntermediateCerts`

```
int wolfSSL_CTX_UnloadIntermediateCerts(
    WOLFSSL_CTX * ctx
)
```

この関数は、CA 署名者リストに追加された中間証明書をアンロードし、解放します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- `wolfSSL_CTX_UnloadCAs`
- `wolfSSL_CertManagerUnloadIntermediateCerts`

Return:

- `SSL_SUCCESS` 関数の実行が成功した場合に返されます。
- `BAD_FUNC_ARG` WOLFSSL_CTX 構造体が NULL の場合、またはサブルーチンで許可されない引数値が渡された場合に返されます。
- `BAD_STATE_E` WOLFSSL_CTX の参照カウントが 1 より大きい場合に返されます。
- `BAD_MUTEX_E` mutex エラーが発生した場合に返されます。 `LockMutex()` が 0 を返しませんでした。

Example

```
WOLFSSL_METHOD method = wolfTLSEv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
...
if(wolfSSL_CTX_UnloadIntermediateCerts(ctx) != NULL){
    // 関数は CA をアンロードしませんでした
}
```

C.52.2.243 function `wolfSSL_CTX_Unload_trust_peers`

```
int wolfSSL_CTX_Unload_trust_peers(
    WOLFSSL_CTX * ctx
)
```

この関数は以前にロードされたすべての信頼されたピア証明書をアンロードするために使用されます。この機能は、マクロ WOLFSSL_TRUST_PEER_CERT を定義することで有効になります。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。

See:

- wolfSSL_CTX_trust_peer_buffer
- wolfSSL_CTX_trust_peer_cert

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG ctx が NULL の場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_Unload_trust_peers(ctx);
if (ret != SSL_SUCCESS) {
    // 信頼されたピア証明書のアンロードエラー
}
...
```

C.52.2.244 function wolfSSL_CTX_trust_peer_buffer

```
int wolfSSL_CTX_trust_peer_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、TLS/SSL ハンドシェイクを実行する際にピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されるピア証明書は、利用可能な場合は SKID と署名を使用して比較されます。これら 2 つが一致しない場合は、ロードされた CA が使用されます。ファイルではなくバッファからの入力である点を除いて、wolfSSL_CTX_trust_peer_cert と同じ機能です。この機能は、マクロ WOLFSSL_TRUST_PEER_CERT を定義することで有効になります。適切な使用方法については例を参照してください。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。
- **buffer** 証明書を含むバッファへのポインタ。
- **sz** 入力バッファの長さ。
- **type** ロードされる証明書のタイプ、すなわち SSL_FILETYPE_ASN1 または 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 成功時。
- SSL_FAILURE ctx が NULL、または file と type の両方が無効な場合に返されます。
- SSL_BAD_FILETYPE ファイルの形式が間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...

ret = wolfSSL_CTX_trust_peer_buffer(ctx, bufferPtr, bufferSz,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // 信頼されたピア証明書のロードエラー
}
...
```

C.52.2.245 function wolfSSL_CTX_load_verify_buffer

```
int wolfSSL_CTX_load_verify_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は CA 証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。適切な使用方法については例を参照してください。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファ (in) のサイズ。
- **format** バッファ証明書の形式、SSL_FILETYPE_ASN1 または 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` 成功時。
- `SSL_BAD_FILETYPE` ファイルの形式が間違っている場合に返されます。
- `SSL_BAD_FILE` ファイルが存在しない、読み取れない、または破損している場合に返されます。
- `MEMORY_E` メモリ不足の状態が発生した場合に返されます。
- `ASN_INPUT_E` ファイルの Base16 デコードが失敗した場合に返されます。
- `BUFFER_E` チェーンバッファが受信バッファより大きい場合に返されます。

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) {
    // バッファからの CA 証明書のロードエラー
}
...
```

C.52.2.246 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
)
```

この関数は CA 証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ `sz` の `in` 引数によって提供されます。`format` はバッファの形式タイプを指定します。`SSL_FILETYPE_ASN1` または `SSL_FILETYPE_PEM` です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。`_ex` 版は PR 2413 で追加され、`userChain` と `flags` の追加引数をサポートします。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファ (`in`) のサイズ。
- **format** バッファ証明書の形式、`SSL_FILETYPE_ASN1` または `SSL_FILETYPE_PEM`。
- **userChain** 形式 `WOLFSSL_FILETYPE_ASN1` を使用している場合、これを非ゼロに設定すると、DER のチェーンが提示されていることを示します。
- **flags** `ssl.h` の `WOLFSSL_LOAD_VERIFY_DEFAULT_FLAGS` 付近を参照してください。

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 成功時。
- SSL_BAD_FILETYPE ファイルの形式が間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファより大きい場合に返されます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

// 期限切れ証明書を強制的にロードする例
ret = wolfSSL_CTX_load_verify_buffer_ex(ctx, certBuff, sz, SSL_FILETYPE_PEM,
    0, (WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY));
if (ret != SSL_SUCCESS) {
    // バッファからの CA 証明書のロードエラー
}
...
```

C.52.2.247 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
)
```

この関数は CA 証明書チェーンバッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。形式が PEM であれば、1 つのバッファに複数の CA 証明書をロードできます。適切な使用方法については例を参照してください。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファ (in) のサイズ。
- **format** バッファ証明書の形式、SSL_FILETYPE_ASN1 または 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` 成功時。
- `SSL_BAD_FILETYPE` ファイルの形式が間違っている場合に返されます。
- `SSL_BAD_FILE` ファイルが存在しない、読み取れない、または破損している場合に返されます。
- `MEMORY_E` メモリ不足の状態が発生した場合に返されます。
- `ASN_INPUT_E` ファイルの Base16 デコードが失敗した場合に返されます。
- `BUFFER_E` チェーンバッファが受信バッファより大きい場合に返されます。

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) {
    // バッファからの CA 証明書のロードエラー
}
...
```

C.52.2.248 function `wolfSSL_CTX_use_certificate_buffer`

```
int wolfSSL_CTX_use_certificate_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は証明書バッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ `sz` の `in` 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **in** ロードする証明書を含む入力バッファ。
- **sz** 入力バッファのサイズ。
- **format** 入力バッファ (in) に格納されている証明書の形式。指定可能な値は SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。

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 成功時。
- SSL_BAD_FILETYPE ファイルの形式が間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。

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) {
    // バッファからの証明書のロードエラー
}
...
```

C.52.2.249 function wolfSSL_CTX_use_PrivateKey_buffer

```
int wolfSSL_CTX_use_PrivateKey_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は秘密鍵バッファを SSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファの形式タイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。
- **in** ロードする秘密鍵を含む入力バッファ。
- **sz** 入力バッファのサイズ。
- **format** 入力バッファ (in) に格納されている秘密鍵の形式。指定可能な値は SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。

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 成功時。
- SSL_BAD_FILETYPE ファイルの形式が間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。
- NO_PASSWORD 鍵ファイルが暗号化されているがパスワードが提供されていない場合に返されます。

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) {
    // バッファからの秘密鍵のロードエラー
}
...

```

C.52.2.250 function wolfSSL_CTX_use_certificate_chain_buffer

```

int wolfSSL_CTX_use_certificate_chain_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz
)

```

この関数は証明書チェーンバッファを WOLFSSL コンテキストにロードします。バッファ非対応版と同様に動作しますが、ファイルの代わりにバッファを入力として呼び出せる点が異なります。バッファはサイズ `sz` の `in` 引数によって提供されます。バッファは PEM 形式で、サブジェクトの証明書から始まり、ルート証明書で終わる必要があります。適切な使用方法については例を参照してください。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された SSL コンテキストへのポインタ。
- **in** ロードする PEM 形式の証明書チェーンを含む入力バッファ。
- **sz** 入力バッファのサイズ。

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` 成功時。
- `SSL_BAD_FILETYPE` ファイルの形式が間違っている場合に返されます。
- `SSL_BAD_FILE` ファイルが存在しない、読み取れない、または破損している場合に返されます。
- `MEMORY_E` メモリ不足の状態が発生した場合に返されます。
- `ASN_INPUT_E` ファイルの Base16 デコードが失敗した場合に返されます。
- `BUFFER_E` チェーンバッファが受信バッファより大きい場合に返されます。

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) {

```

```

    // バッファから証明書チェーンの読み込みに失敗
}
...

```

C.52.2.251 function wolfSSL_use_certificate_buffer

```

int wolfSSL_use_certificate_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)

```

この関数は、証明書バッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **in** ロードする証明書を含むバッファ。
- **sz** バッファ内の証明書のサイズ。
- **format** ロードする証明書のフォーマット。指定可能な値は SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。

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 成功時。
- SSL_BAD_FILETYPE ファイルのフォーマットが間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードに失敗した場合に返されます。

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) {
    // バッファから証明書の読み込みに失敗
}

```

C.52.2.252 function wolfSSL_use_PrivateKey_buffer

```
int wolfSSL_use_PrivateKey_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、秘密鍵バッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができる点が異なります。バッファはサイズ sz の in 引数によって提供されます。format はバッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。適切な使用方法については例を参照してください。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **in** ロードする秘密鍵を含むバッファ。
- **sz** バッファ内の秘密鍵のサイズ。
- **format** ロードする秘密鍵のフォーマット。指定可能な値は SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM です。

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 成功時。
- SSL_BAD_FILETYPE ファイルのフォーマットが間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードに失敗した場合に返されます。
- NO_PASSWORD 鍵ファイルが暗号化されているがパスワードが提供されていない場合に返されます。

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) {
    // バッファから秘密鍵の読み込みに失敗
}
```

C.52.2.253 function wolfSSL_use_certificate_chain_buffer

```
int wolfSSL_use_certificate_chain_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz
)
```

この関数は、証明書チェーンバッファを WOLFSSL オブジェクトにロードします。バッファなしバージョンと同様に動作しますが、ファイルの代わりにバッファを入力として呼び出すことができます。バッファはサイズ `sz` の `in` 引数によって提供されます。バッファは PEM フォーマットである必要があり、サブジェクトの証明書から始まり、ルート証明書で終わる必要があります。適切な使用方法については例を参照してください。

Parameters:

- **ssl** `wolfSSL_new()` で作成された SSL セッションへのポインタ。
- **in** ロードする証明書を含むバッファ。
- **sz** バッファ内の証明書のサイズ。

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_SUCCESS` 成功時。
- `SSL_BAD_FILETYPE` ファイルのフォーマットが間違っている場合に返されます。
- `SSL_BAD_FILE` ファイルが存在しない、読み取れない、または破損している場合に返されます。
- `MEMORY_E` メモリ不足状態が発生した場合に返されます。
- `ASN_INPUT_E` ファイルの Base16 デコードに失敗した場合に返されます。
- `BUFFER_E` チェーンバッファが受信バッファより大きい場合に返されます。

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) {
    // バッファから証明書チェーンの読み込みに失敗
}
```

C.52.2.254 function wolfSSL_UnloadCertsKeys

```
int wolfSSL_UnloadCertsKeys(
    WOLFSSL * ssl
)
```

この関数は、SSL が所有する証明書または鍵をアンロードします。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_CTX_UnloadCAs`

Return:

- `SSL_SUCCESS` 関数が正常に実行された場合に返されます。
- `BAD_FUNC_ARG` WOLFSSL オブジェクトが `NULL` の場合に返されます。

Example


```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
int unloadKeys = wolfSSL_UnloadCertsKeys(ssl);
if(unloadKeys != SSL_SUCCESS){
    // 失敗ケース
}
```

C.52.2.255 function wolfSSL_CTX_set_group_messages

```
int wolfSSL_CTX_set_group_messages(
    WOLFSSL_CTX * ctx
)
```

この関数は、可能な場合にハンドシェイクメッセージのグループ化をオンにします。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された SSL コンテキストへのポインタ。
- **ctx** WOLFSSL_CTX 構造体へのポインタ。

See:

- [wolfSSL_set_group_messages](#)
- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_clear_group_messages](#)
- [wolfSSL_set_group_messages](#)
- [wolfSSL_clear_group_messages](#)

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 入力コンテキストが null の場合に返されます。
- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG ctx が NULL の場合。

指定された WOLFSSL_CTX コンテキストに対してハンドシェイクメッセージグループ化を有効にします。

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_group_messages(ctx);
if (ret != SSL_SUCCESS) {
    // ハンドシェイクメッセージのグループ化の設定に失敗
}
```

この関数は、指定されたコンテキストから作成されたすべての SSL オブジェクトに対してハンドシェイクメッセージグループ化をオンにします。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
wolfSSL_CTX_set_group_messages(ctx);
```

C.52.2.256 function wolfSSL_set_group_messages

```
int wolfSSL_set_group_messages(
    WOLFSSL * ssl
)
```


この関数は、可能な場合にハンドシェイクメッセージのグループ化をオンにします。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL セッションへのポインタ。
- **ssl** WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_CTX_set_group_messages`
- `wolfSSL_new`
- `wolfSSL_clear_group_messages`
- `wolfSSL_CTX_set_group_messages`
- `wolfSSL_CTX_clear_group_messages`

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 入力コンテキストが null の場合に返されます。
- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG ssl が NULL の場合。

指定された WOLFSSL オブジェクトに対してハンドシェイクメッセージグループ化を有効にします。

Example

```
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_group_messages(ssl);
if (ret != SSL_SUCCESS) {
    // ハンドシェイクメッセージのグループ化の設定に失敗
}
```

この関数は、指定された SSL オブジェクトに対してハンドシェイクメッセージグループ化をオンにします。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_set_group_messages(ssl);
```

C.52.2.257 function wolfSSL_SetFuzzerCb

```
void wolfSSL_SetFuzzerCb(
    WOLFSSL * ssl,
    CallbackFuzzer cbf,
    void * fCtx
)
```

この関数はファザーコールバックを設定します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **cbf** 次の形式の関数ポインタである CallbackFuzzer 型: `int (CallbackFuzzer)(WOLFSSL ssl, const unsigned char* buf, int sz, int type, void* fuzzCtx);`
- **fCtx** WOLFSSL 構造体の `fuzzerCtx` メンバに設定される void ポインタ型。

See: CallbackFuzzer

Return: none 戻り値はありません。

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){
    // 関数定義
}
...
wolfSSL_SetFuzzerCb(ssl, callbackFuzzerCB, fCtx);

```

C.52.2.258 function wolfSSL_DTLS_SetCookieSecret

```

int wolfSSL_DTLS_SetCookieSecret(
    WOLFSSL * ssl,
    const byte * secret,
    word32 secretSz
)

```

この関数は新しい dtls cookie シークレットを設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **secret** シークレットバッファを表す定数バイトポインタ。
- **secretSz** バッファのサイズ。

See:

- [ForceZero](#)
- [wc_RNG_GenerateBlock](#)

Return:

- 0 関数がエラーなく実行された場合に返されます。
- BAD_FUNC_ARG 関数に許容できない値を持つ引数が渡された場合に返されます。
- COOKIE_SECRET_SZ シークレットサイズが 0 の場合に返されます。
- MEMORY_ERROR 新しい cookie シークレット用のメモリ割り当てに問題があった場合に返されます。

Example

```

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const* byte secret;
word32 secretSz; // secret のサイズ
...
if(!wolfSSL_DTLS_SetCookieSecret(ssl, secret, secretSz)){
    // DTLS クッキーシークレットの設定失敗のコードブロック
} else {
    // 成功! クッキーシークレットが設定されました
}

```

C.52.2.259 function wolfSSL_GetRNG

```

WC_RNG * wolfSSL_GetRNG(
    WOLFSSL * ssl
)

```

この関数は乱数を取得します。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL オブジェクトへのポインタ。

See: `wolfSSL_CTX_new_rng`

Return:

- rng 成功時。
- NULL ssl が NULL の場合。

Example

```
WOLFSSL* ssl;

wolfSSL_GetRNG(ssl);
```

C.52.2.260 function `wolfSSL_CTX_SetMinVersion`

```
int wolfSSL_CTX_SetMinVersion(
    WOLFSSL_CTX * ctx,
    int version
)
```

この関数は、許可される最小ダウングレードバージョンを設定します。ダウングレードを許可する接続 (`wolfSSLv23_client_method` または `wolfSSLv23_server_method`) を使用する場合にのみ適用されます。

Parameters:

- **ctx** `wolfSSL_CTX_new()`を使用して作成された `WOLFSSL_CTX` 構造体へのポインタ。
- **version** 最小値として設定されるバージョンの整数表現: `WOLFSSL_SSLV3` = 0、`WOLFSSL_TLSV1` = 1、`WOLFSSL_TLSV1_1` = 2、または `WOLFSSL_TLSV1_2` = 3。

See: `SetMinVersionHelper`

Return:

- `SSL_SUCCESS` 関数がエラーなく戻り、最小バージョンが設定された場合に返されます。
- `BAD_FUNC_ARG` `WOLFSSL_CTX` 構造体が NULL の場合、または最小バージョンがサポートされていない場合に返されます。

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
int version; // マクロ表現
...
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
    // 最小バージョンの設定に失敗
}
```

C.52.2.261 function `wolfSSL_SetMinVersion`

```
int wolfSSL_SetMinVersion(
    WOLFSSL * ssl,
    int version
)
```

この関数は、許可される最小ダウングレードバージョンを設定します。ダウングレードを許可する接続 (`wolfSSLv23_client_method` または `wolfSSLv23_server_method`) を使用する場合にのみ適用されます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **version** 最小値として設定されるバージョンの整数表現: WOLFSSL_SSLV3 = 0、WOLFSSL_TLSV1 = 1、WOLFSSL_TLSV1_1 = 2、または WOLFSSL_TLSV1_2 = 3。

See: SetMinVersionHelper

Return:

- SSL_SUCCESS この関数とそのサブルーチンがエラーなく実行された場合に返されます。
- BAD_FUNC_ARG SSL オブジェクトが NULL の場合に返されます。サブルーチンでは、バージョンマッチが良好でない場合にこのエラーがスローされます。

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(protocol method);
WOLFSSL* ssl = WOLFSSL_new(ctx);
int version; // マクロ表現
...
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
    // 最小バージョンの設定に失敗
}
```

C.52.2.262 function wolfSSL_GetObjectSize

```
int wolfSSL_GetObjectSize(
    void
)
```

この関数は WOLFSSL オブジェクトのサイズを返し、ビルドオプションと設定に依存します。wolfSSL をビルドする際に SHOW_SIZES が定義されている場合、この関数は WOLFSSL オブジェクト内の個々のオブジェクト (Suites、Ciphers など) のサイズも stdout に出力します。

Parameters:

- **none** パラメータはありません。

See: `wolfSSL_new`

Return: size この関数は WOLFSSL オブジェクトのサイズを返します。

Example

```
int size = 0;
size = wolfSSL_GetObjectSize();
printf("sizeof(WOLFSSL) = %d\n", size);
```

C.52.2.263 function wolfSSL_GetOutputSize

```
int wolfSSL_GetOutputSize(
    WOLFSSL * ssl,
    int inSz
)
```

平文入力のレコード層サイズを返します。これは、アプリケーションが指定された平文入力サイズに対して、トランスポート層を介して送信されるバイト数を知りたい場合に役立ちます。この関数は、SSL/TLS ハンドシェイクが完了した後に呼び出す必要があります。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。

- **inSz** 平文データのサイズ。

See: [wolfSSL_GetMaxOutputSize](#)

Return:

- size 成功時には、要求されたサイズが返されます。
- INPUT_SIZE_E 入力サイズが最大 TLS フラグメントサイズより大きい場合に返されます ([wolfSSL_GetMaxOutputSize\(\)](#)を参照)。
- BAD_FUNC_ARG 無効な関数引数が渡された場合、または SSL/TLS ハンドシェイクがまだ完了していない場合に返されます。

Example

none

C.52.2.264 function wolfSSL_GetMaxOutputSize

```
int wolfSSL_GetMaxOutputSize(
    WOLFSSL * ssl
)
```

平文データの最大レコード層サイズを返します。これは、プロトコル標準で指定されている最大 SSL/TLS レコードサイズ、または TLS 最大フラグメント長拡張によって設定された最大 TLS フラグメントサイズのいずれかに対応します。この関数は、アプリケーションが [wolfSSL_GetOutputSize\(\)](#) を呼び出して INPUT_SIZE_E エラーを受け取った場合に役立ちます。この関数は、SSL/TLS ハンドシェイクが完了した後に呼び出す必要があります。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL オブジェクトへのポインタ。

See: [wolfSSL_GetOutputSize](#)

Return:

- size 成功時には、最大出力サイズが返されます。
- BAD_FUNC_ARG 無効な関数引数が渡された場合、または SSL/TLS ハンドシェイクがまだ完了していない場合に返されます。

Example

none

C.52.2.265 function wolfSSL_SetVersion

```
int wolfSSL_SetVersion(
    WOLFSSL * ssl,
    int version
)
```

この関数は、version で指定されたバージョンを使用して、指定された SSL セッション (WOLFSSL オブジェクト) の SSL/TLS プロトコルバージョンを設定します。これにより、SSL セッション (ssl) のプロトコル設定が上書きされます。これは元々 SSL コンテキスト ([wolfSSL_CTX_new\(\)](#)) のメソッドタイプによって定義および設定されたものです。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **version** SSL/TLS プロトコルバージョン。指定可能な値には、WOLFSSL_SSLV3、WOLFSSL_TL SV1、WOLFSSL_TL SV1_1、WOLFSSL_TL SV1_2 があります。

See: [wolfSSL_CTX_new](#)

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG 入力された SSL オブジェクトが NULL の場合、または version に不正なプロトコルバージョンが指定された場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_SetVersion(ssl, WOLFSSL_TLSV1);
if (ret != SSL_SUCCESS) {
    // SSL セッションのプロトコルバージョンの設定に失敗しました。
}
```

C.52.2.266 function wolfSSL_CTX_SetMacEncryptCb

```
void wolfSSL_CTX_SetMacEncryptCb(
    WOLFSSL_CTX * ctx,
    CallbackMacEncrypt cb
)
```

呼び出し元がアトミックユーザレコード処理 Mac/暗号化コールバックを設定できるようにします。コールバックは、成功時には 0、エラー時には <0 を返す必要があります。ssl と ctx ポインタは、ユーザの利便性のために利用可能です。macOut は、mac の結果を格納する出力バッファです。macIn は mac 入力バッファであり、macInSz はバッファのサイズを示します。macContent と macVerify は、wolfSSL_SetTlsHmacInner() に必要であり、そのまま渡す必要があります。encOut は、暗号化の結果を格納する出力バッファです。encIn は暗号化する入力バッファであり、encSz は入力のサイズです。コールバックの例は、wolfssl/test.h の myMacEncryptCb() にあります。

Parameters:

- none パラメータなし。

See:

- [wolfSSL_SetMacEncryptCtx](#)
- [wolfSSL_GetMacEncryptCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.267 function wolfSSL_SetMacEncryptCtx

```
void wolfSSL_SetMacEncryptCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

呼び出し元がアトミックユーザレコード処理 Mac/暗号化コールバックコンテキストを ctx に設定できるようにします。

Parameters:

- none パラメータなし。

See:

- [wolfSSL_CTX_SetMacEncryptCb](#)
- [wolfSSL_GetMacEncryptCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.268 function wolfSSL_GetMacEncryptCtx

```
void * wolfSSL_GetMacEncryptCtx(  
    WOLFSSL * ssl  
)
```

呼び出し元が、wolfSSL_SetMacEncryptCtx() で以前に保存されたアトミックユーザレコード処理 Mac/暗号化コールバックコンテキストを取得できるようにします。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_CTX_SetMacEncryptCb](#)
- [wolfSSL_SetMacEncryptCtx](#)

Return:

- pointer 成功した場合、呼び出しはコンテキストへの有効なポインタを返します。
- NULL 空のコンテキストの場合に返されます。

Example

none

C.52.2.269 function wolfSSL_CTX_SetDecryptVerifyCb

```
void wolfSSL_CTX_SetDecryptVerifyCb(  
    WOLFSSL_CTX * ctx,  
    CallbackDecryptVerify cb  
)
```

呼び出し元がアトミックユーザレコード処理復号/検証コールバックを設定できるようにします。コールバックは、成功時には0、エラー時には<0を返す必要があります。sslとctxポインタは、ユーザの利便性のために利用可能です。decOutは、復号の結果を格納する出力バッファです。decInは暗号化された入力バッファであり、decInSzはバッファのサイズを示します。contentとverifyは、wolfSSL_SetTlsHmacInner()に必要であり、そのまま渡す必要があります。padSzは、パディングの合計値を設定する必要がある出力変数です。つまり、macサイズに加えて、パディングとパディングバイトです。コールバックの例は、wolfssl/test.hのmyDecryptVerifyCb()にあります。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_SetMacEncryptCtx](#)
- [wolfSSL_GetMacEncryptCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.270 function wolfSSL_SetDecryptVerifyCtx

```
void wolfSSL_SetDecryptVerifyCtx(  
    WOLFSSL * ssl,  
    void * ctx  
)
```

呼び出し元がアトミックユーザレコード処理復号/検証コールバックコンテキストを ctx に設定できるようにします。

Parameters:

- none パラメータなし。

See:

- [wolfSSL_CTX_SetDecryptVerifyCb](#)
- [wolfSSL_GetDecryptVerifyCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.271 function wolfSSL_GetDecryptVerifyCtx

```
void * wolfSSL_GetDecryptVerifyCtx(  
    WOLFSSL * ssl  
)
```

呼び出し元が、wolfSSL_SetDecryptVerifyCtx() で以前に保存されたアトミックユーザレコード処理復号/検証コールバックコンテキストを取得できるようにします。

Parameters:

- none パラメータなし。

See:

- [wolfSSL_CTX_SetDecryptVerifyCb](#)
- [wolfSSL_SetDecryptVerifyCtx](#)

Return:

- pointer 成功した場合、呼び出しはコンテキストへの有効なポインタを返します。
- NULL 空のコンテキストの場合に返されます。

Example

none

C.52.2.272 function wolfSSL_GetMacSecret

```
const unsigned char * wolfSSL_GetMacSecret(  
    WOLFSSL * ssl,  
    int verify  
)
```


ハンドシェイクプロセスから Hmac/Mac シークレットの取得を可能にします。verify パラメータは、これがピアメッセージの検証用であるかどうかを指定します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。
- **verify** これがピアメッセージの検証用であるかどうかを指定します。

See: `wolfSSL_GetHmacSize`

Return:

- pointer 成功した場合、呼び出しはシークレットへの有効なポインタを返します。シークレットのサイズは、`wolfSSL_GetHmacSize()` から取得できます。
- NULL エラー状態の場合に返されます。

Example

none

C.52.2.273 function wolfSSL_GetClientWriteKey

```
const unsigned char * wolfSSL_GetClientWriteKey(  
    WOLFSSL *  
)
```

ハンドシェイクプロセスからクライアント書き込み鍵の取得を可能にします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- `wolfSSL_GetKeySize`
- `wolfSSL_GetClientWriteIV`

Return:

- pointer 成功した場合、呼び出しは鍵への有効なポインタを返します。鍵のサイズは、`wolfSSL_GetKeySize()` から取得できます。
- NULL エラー状態の場合に返されます。

Example

none

C.52.2.274 function wolfSSL_GetClientWriteIV

```
const unsigned char * wolfSSL_GetClientWriteIV(  
    WOLFSSL *  
)
```

ハンドシェイクプロセスからクライアント書き込み IV(初期化ベクトル) の取得を可能にします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- `wolfSSL_GetCipherBlockSize()`
- `wolfSSL_GetClientWriteKey()`

Return:

- pointer 成功した場合、呼び出しは IV への有効なポインタを返します。IV のサイズは、wolfSSL_GetCipherBlockSize() から取得できます。
- NULL エラー状態の場合に返されます。

Example

none

C.52.2.275 function wolfSSL_GetServerWriteKey

```
const unsigned char * wolfSSL_GetServerWriteKey(  
    WOLFSSL *  
)
```

ハンドシェイクプロセスからサーバ書き込み鍵の取得を可能にします。

Parameters:

- ssl wolfSSL_new()を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- wolfSSL_GetKeySize
- wolfSSL_GetServerWriteIV

Return:

- pointer 成功した場合、呼び出しは鍵への有効なポインタを返します。鍵のサイズは、wolfSSL_GetKeySize() から取得できます。
- NULL エラー状態の場合に返されます。

Example

none

C.52.2.276 function wolfSSL_GetServerWriteIV

```
const unsigned char * wolfSSL_GetServerWriteIV(  
    WOLFSSL *  
)
```

ハンドシェイクプロセスからサーバ書き込み IV(初期化ベクトル) の取得を可能にします。

Parameters:

- ssl wolfSSL_new()を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- wolfSSL_GetCipherBlockSize
- wolfSSL_GetClientWriteKey

Return:

- pointer 成功した場合、呼び出しは IV への有効なポインタを返します。IV のサイズは、wolfSSL_GetCipherBlockSize() から取得できます。
- NULL エラー状態の場合に返されます。

C.52.2.277 function wolfSSL_GetKeySize

```
int wolfSSL_GetKeySize(  
    WOLFSSL * ssl  
)
```

ハンドシェイクプロセスから鍵サイズの取得を可能にします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- `wolfSSL_GetClientWriteKey`
- `wolfSSL_GetServerWriteKey`

Return:

- `size` 成功した場合、呼び出しは鍵サイズをバイト単位で返します。
- `BAD_FUNC_ARG` エラー状態の場合に返されます。

Example

none

C.52.2.278 function wolfSSL_GetIVSize

```
int wolfSSL_GetIVSize(  
    WOLFSSL * ssl  
)
```

WOLFSSL 構造体に保持されている specs 構造体の iv_size メンバを返します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_GetKeySize`
- `wolfSSL_GetClientWriteIV`
- `wolfSSL_GetServerWriteIV`

Return:

- `iv_size` `ssl->specs.iv_size` に保持されている値を返します。
- `BAD_FUNC_ARG` WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );  
WOLFSSL* ssl = wolfSSL_new(ctx);  
int ivSize;  
...  
ivSize = wolfSSL_GetIVSize(ssl);  
  
if(ivSize > 0){  
    // ivSize は specs.iv_size 値を保持しています。  
}
```

C.52.2.279 function wolfSSL_GetSide

```
int wolfSSL_GetSide(  
    WOLFSSL * ssl  
)
```

この WOLFSSL 接続のサイド (側) の取得を可能にします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- `wolfSSL_GetClientWriteKey`
- `wolfSSL_GetServerWriteKey`

Return:

- success 成功した場合、呼び出しは WOLFSSL オブジェクトのサイドに応じて、WOLFSSL_SERVER_END または WOLFSSL_CLIENT_END のいずれかを返します。
- BAD_FUNC_ARG エラー状態の場合に返されます。

Example

none

C.52.2.280 function `wolfSSL_IsTLSv1_1`

```
int wolfSSL_IsTLSv1_1(  
    WOLFSSL * ssl  
)
```

呼び出し元が、ネゴシエートされたプロトコルバージョンが少なくとも TLS バージョン 1.1 以上であるかどうかを判断できるようにします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。

See: `wolfSSL_GetSide`

Return:

- true/false 成功した場合、呼び出しは真の場合は 1、偽の場合は 0 を返します。
- BAD_FUNC_ARG エラー状態の場合に返されます。

Example

none

C.52.2.281 function `wolfSSL_GetBulkCipher`

```
int wolfSSL_GetBulkCipher(  
    WOLFSSL * ssl  
)
```

呼び出し元がハンドシェイクからネゴシエートされたバルク暗号アルゴリズムを判定できるようにします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- `wolfSSL_GetCipherBlockSize`
- `wolfSSL_GetKeySize`

Return:

- 成功時、以下のいずれかを返します。`wolfssl_cipher_null`、`wolfssl_des`、`wolfssl_triple_des`、`wolfssl_aes`、`wolfssl_aes_gcm`、`wolfssl_aes_ccm`、`wolfssl_camellia`。
- BAD_FUNC_ARG エラー状態の場合に返されます。

Example

none

C.52.2.282 function wolfSSL_GetCipherBlockSize

```
int wolfSSL_GetCipherBlockSize(  
    WOLFSSL * ssl  
)
```

呼び出し元がハンドシェイクからネゴシエートされた暗号ブロックサイズを判定できるようにします。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- [wolfSSL_GetBulkCipher](#)
- [wolfSSL_GetKeySize](#)

Return:

- size 成功時、暗号ブロックサイズのバイト数を返します。
- BAD_FUNC_ARG エラー状態の場合に返されます。

Example

none

C.52.2.283 function wolfSSL_GetAeadMacSize

```
int wolfSSL_GetAeadMacSize(  
    WOLFSSL * ssl  
)
```

呼び出し元がハンドシェイクからネゴシエートされた aead mac サイズを判定できるようにします。暗号タイプ WOLFSSL_AEAD_TYPE 用。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- [wolfSSL_GetBulkCipher](#)
- [wolfSSL_GetKeySize](#)

Return:

- size 成功時、aead mac サイズのバイト数を返します。
- BAD_FUNC_ARG エラー状態の場合に返されます。

Example

none

C.52.2.284 function wolfSSL_GetHmacSize

```
int wolfSSL_GetHmacSize(  
    WOLFSSL * ssl  
)
```

呼び出し元がハンドシェイクからネゴシエートされた (h)mac サイズを判定できるようにします。WOLF-SSL_AEAD_TYPE 以外の暗号タイプ用。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- `wolfSSL_GetBulkCipher`
- `wolfSSL_GetHmacType`

Return:

- size 成功時、(h)mac サイズのバイト数を返します。
- BAD_FUNC_ARG エラー状態の場合に返されます。

Example

none

C.52.2.285 function wolfSSL_GetHmacType

```
int wolfSSL_GetHmacType(  
    WOLFSSL * ssl  
)
```

呼び出し元がハンドシェイクからネゴシエートされた (h)mac タイプを判定できるようにします。WOLF-SSL_AEAD_TYPE 以外の暗号タイプ用。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- `wolfSSL_GetBulkCipher`
- `wolfSSL_GetHmacSize`

Return:

- 成功時、以下のいずれかを返します。MD5、SHA、SHA256、SHA384。
- BAD_FUNC_ARG エラー状態の場合に返される可能性があります。
- SSL_FATAL_ERROR エラー状態の場合に返される可能性があります。

Example

none

C.52.2.286 function wolfSSL_GetCipherType

```
int wolfSSL_GetCipherType(  
    WOLFSSL * ssl  
)
```

呼び出し元がハンドシェイクからネゴシエートされた暗号タイプを判定できるようにします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- `wolfSSL_GetBulkCipher`
- `wolfSSL_GetHmacType`

Return:

- 成功時、以下のいずれかを返します。WOLFSSL_BLOCK_TYPE、WOLFSSL_STREAM_TYPE、WOLFSSL_AEAD_TYPE。
- BAD_FUNC_ARG エラー状態の場合に返されます。

Example

none

C.52.2.287 function wolfSSL_SetTlsHmacInner

```
int wolfSSL_SetTlsHmacInner(  
    WOLFSSL * ssl,  
    byte * inner,  
    word32 sz,  
    int content,  
    int verify  
)
```

呼び出し元がメッセージの送受信のために Hmac Inner ベクトルを設定できるようにします。結果は inner に書き込まれ、少なくとも wolfSSL_GetHmacSize() バイトである必要があります。メッセージのサイズは sz で指定され、content はメッセージのタイプ、verify はこれがピアメッセージの検証であるかを指定します。WOLFSSL_AEAD_TYPE を除く暗号タイプで有効です。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_GetBulkCipher](#)
- [wolfSSL_GetHmacType](#)

Return:

- 1 成功時。
- BAD_FUNC_ARG エラー状態の場合に返されます。

Example

none

C.52.2.288 function wolfSSL_CTX_SetEccSignCb

```
void wolfSSL_CTX_SetEccSignCb(  
    WOLFSSL_CTX * ctx,  
    CallbackEccSign cb  
)
```

呼び出し元が ECC 署名用の公開鍵コールバックを設定できるようにします。コールバックは成功時に 0、エラー時に 0 未満を返す必要があります。ssl と ctx ポインタはユーザの便宜のために利用可能です。in は署名する入力バッファで、inSz は入力の長さを示します。out は署名の結果を格納する出力バッファです。outSz は入出力変数で、呼び出し時の出力バッファのサイズを指定し、返す前に署名の実際のサイズをそこに格納する必要があります。keyDer は ASN1 形式の ECC 秘密鍵で、keySz は鍵の長さをバイト単位で表します。コールバックの例は wolfssl/test.h myEccSign() にあります。

Parameters:

- **none** パラメータなし。

See:

- `wolfSSL_SetEccSignCtx`
- `wolfSSL_GetEccSignCtx`

Return: none 戻り値なし。

Example

none

C.52.2.289 function `wolfSSL_SetEccSignCtx`

```
void wolfSSL_SetEccSignCtx(  
    WOLFSSL * ssl,  
    void * ctx  
)
```

呼び出し元が公開鍵 ECC 署名コールバックコンテキストを ctx に設定できるようにします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。
- **ctx** 格納するユーザコンテキストへのポインタ。

See:

- `wolfSSL_CTX_SetEccSignCb`
- `wolfSSL_GetEccSignCtx`

Return: none 戻り値なし。

Example

none

C.52.2.290 function `wolfSSL_GetEccSignCtx`

```
void * wolfSSL_GetEccSignCtx(  
    WOLFSSL * ssl  
)
```

呼び出し元が以前に `wolfSSL_SetEccSignCtx()` で格納された公開鍵 ECC 署名コールバックコンテキストを取得できるようにします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL オブジェクトへのポインタ。

See:

- `wolfSSL_CTX_SetEccSignCb`
- `wolfSSL_SetEccSignCtx`

Return:

- pointer 成功時、コンテキストへの有効なポインタを返します。
- NULL 空のコンテキストの場合に返されます。

Example

none

C.52.2.291 function wolfSSL_CTX_SetEccSignCtx

```
void wolfSSL_CTX_SetEccSignCtx(  
    WOLFSSL_CTX * ctx,  
    void * userCtx  
)
```

呼び出し元が公開鍵 ECC 署名コールバックコンテキストを ctx に設定できるようにします。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX 構造体へのポインタ。
- **ctx** 格納するユーザコンテキストへのポインタ。

See:

- `wolfSSL_CTX_SetEccSignCb`
- `wolfSSL_CTX_GetEccSignCtx`

Return: none 戻り値なし。

Example

none

C.52.2.292 function wolfSSL_CTX_GetEccSignCtx

```
void * wolfSSL_CTX_GetEccSignCtx(  
    WOLFSSL_CTX * ctx  
)
```

呼び出し元が以前に `wolfSSL_SetEccSignCtx()` で格納された公開鍵 ECC 署名コールバックコンテキストを取得できるようにします。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- `wolfSSL_CTX_SetEccSignCb`
- `wolfSSL_CTX_SetEccSignCtx`

Return:

- pointer 成功時、コンテキストへの有効なポインタを返します。
- NULL 空のコンテキストの場合に返されます。

Example

none

C.52.2.293 function wolfSSL_CTX_SetEccVerifyCb

```
void wolfSSL_CTX_SetEccVerifyCb(  
    WOLFSSL_CTX * ctx,  
    CallbackEccVerify cb  
)
```

呼び出し元が ECC 検証用の公開鍵コールバックを設定できるようにします。コールバックは成功時に 0、エラー時に 0 未満を返す必要があります。ssl と ctx ポインタはユーザの便宜のために利用可能です。sig は検証する署名で、sigSz は署名の長さを示します。hash はメッセージのダイジェストを含む入力バッファで、hashSz はハッシュの長さをバイト単位で示します。result は検証の結果を格納する出力変数で、成功

時は 1、失敗時は 0 です。keyDer は ASN1 形式の ECC 秘密鍵で、keySz は鍵の長さをバイト単位で表します。コールバックの例は wolfssl/test.h myEccVerify() にあります。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_SetEccVerifyCtx](#)
- [wolfSSL_GetEccVerifyCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.294 function wolfSSL_SetEccVerifyCtx

```
void wolfSSL_SetEccVerifyCtx(  
    WOLFSSL * ssl,  
    void * ctx  
)
```

呼び出し元が公開鍵 ECC 検証コールバックコンテキストを ctx に設定できるようにします。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_CTX_SetEccVerifyCb](#)
- [wolfSSL_GetEccVerifyCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.295 function wolfSSL_GetEccVerifyCtx

```
void * wolfSSL_GetEccVerifyCtx(  
    WOLFSSL * ssl  
)
```

呼び出し元が以前に wolfSSL_SetEccVerifyCtx() で格納された公開鍵 ECC 検証コールバックコンテキストを取得できるようにします。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_CTX_SetEccVerifyCb](#)
- [wolfSSL_SetEccVerifyCtx](#)

Return:

- pointer 成功時、コンテキストへの有効なポインタを返します。
- NULL 空のコンテキストの場合に返されます。

Example

none

C.52.2.296 function wolfSSL_CTX_SetRsaSignCb

```
void wolfSSL_CTX_SetRsaSignCb(  
    WOLFSSL_CTX * ctx,  
    CallbackRsaSign cb  
)
```

呼び出し元が RSA 署名用の公開鍵コールバックを設定できるようにします。コールバックは成功時に 0、エラー時に 0 未満を返す必要があります。ssl と ctx ポインタはユーザの便宜のために利用可能です。in は署名する入力バッファで、inSz は入力長の長さを示します。out は署名の結果を格納する出力バッファです。outSz は入出力変数で、呼び出し時の出力バッファのサイズを指定し、返す前に署名の実際のサイズをそこに格納する必要があります。keyDer は ASN1 形式の RSA 秘密鍵で、keySz は鍵の長さをバイト単位で表します。コールバックの例は wolfssl/test.h myRsaSign() にあります。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_SetRsaSignCtx](#)
- [wolfSSL_GetRsaSignCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.297 function wolfSSL_SetRsaSignCtx

```
void wolfSSL_SetRsaSignCtx(  
    WOLFSSL * ssl,  
    void * ctx  
)
```

呼び出し元が公開鍵 RSA 署名コールバックコンテキストを ctx に設定できるようにします。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_CTX_SetRsaSignCb](#)
- [wolfSSL_GetRsaSignCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.298 function wolfSSL_GetRsaSignCtx

```
void * wolfSSL_GetRsaSignCtx(  
    WOLFSSL * ssl  
)
```

呼び出し元が以前に wolfSSL_SetRsaSignCtx() で格納された公開鍵 RSA 署名コールバックコンテキストを取得できるようにします。

Parameters:

- **none** パラメータなし。
- **none** パラメータなし。

See:

- [wolfSSL_CTX_SetRsaSignCb](#)
- [wolfSSL_SetRsaSignCtx](#)

Return:

- pointer 成功時、コンテキストへの有効なポインタを返します。
- NULL 空のコンテキストの場合に返されます。

Example

none

C.52.2.299 function wolfSSL_CTX_SetRsaVerifyCb

```
void wolfSSL_CTX_SetRsaVerifyCb(  
    WOLFSSL_CTX * ctx,  
    CallbackRsaVerify cb  
)
```

呼び出し元が RSA 検証用の公開鍵コールバックを設定できるようにします。コールバックは成功時に平文のバイト数、エラー時に 0 未満を返す必要があります。ssl と ctx ポインタはユーザの便宜のために利用可能です。sig は検証する署名で、sigSz は署名の長さを示します。out は復号プロセスとパディングの後、検証バッファの先頭に設定する必要があります。keyDer は ASN1 形式の RSA 公開鍵で、keySz は鍵の長さをバイト単位で表します。コールバックの例は wolfssl/test.h myRsaVerify() にあります。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_SetRsaVerifyCtx](#)
- [wolfSSL_GetRsaVerifyCtx](#)

Return: none 戻り値なし。

C.52.2.300 function wolfSSL_SetRsaVerifyCtx

```
void wolfSSL_SetRsaVerifyCtx(  
    WOLFSSL * ssl,  
    void * ctx  
)
```

呼び出し元が公開鍵 RSA 検証コールバックコンテキストを ctx に設定できるようにします。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_CTX_SetRsaVerifyCb](#)
- [wolfSSL_GetRsaVerifyCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.301 function wolfSSL_GetRsaVerifyCtx

```
void * wolfSSL_GetRsaVerifyCtx(  
    WOLFSSL * ssl  
)
```

呼び出し元が wolfSSL_SetRsaVerifyCtx() で以前に保存された公開鍵 RSA 検証コールバックコンテキストを取得できるようにします。

Parameters:

- none パラメータなし。

See:

- wolfSSL_CTX_SetRsaVerifyCb
- wolfSSL_SetRsaVerifyCtx

Return:

- pointer 成功した場合、呼び出しはコンテキストへの有効なポインタを返します。
- NULL 空のコンテキストに対して返されます。

Example

none

C.52.2.302 function wolfSSL_CTX_SetRsaEncCb

```
void wolfSSL_CTX_SetRsaEncCb(  
    WOLFSSL_CTX * ctx,  
    CallbackRsaEnc cb  
)
```

呼び出し元が RSA 公開暗号化のための公開鍵コールバックを設定できるようにします。コールバックは成功の場合は 0 を、エラーの場合は 0 未満を返す必要があります。ssl と ctx ポインタはユーザーの利便性のために利用可能です。in は暗号化する入力バッファで、inSz は入力の長さを示します。out は暗号化の結果を格納する出力バッファです。outSz は入出力変数で、呼び出し時に出力バッファのサイズを指定し、戻る前に暗号化の実際のサイズを格納する必要があります。keyDer は ASN1 形式の RSA 公開鍵で、keySz はバイト単位の鍵の長さです。コールバックの例は wolfssl/test.h の myRsaEnc() にあります。

Parameters:

- none パラメータなし。

See:

- wolfSSL_SetRsaEncCtx
- wolfSSL_GetRsaEncCtx

Return: none 戻り値なし。

Examples

none

C.52.2.303 function wolfSSL_SetRsaEncCtx

```
void wolfSSL_SetRsaEncCtx(  
    WOLFSSL * ssl,  
    void * ctx  
)
```

呼び出し元が公開鍵 RSA 公開暗号化コールバックコンテキストを ctx に設定できるようにします。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_CTX_SetRsaEncCb](#)
- [wolfSSL_GetRsaEncCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.304 function wolfSSL_GetRsaEncCtx

```
void * wolfSSL_GetRsaEncCtx(  
    WOLFSSL * ssl  
)
```

呼び出し元が wolfSSL_SetRsaEncCtx() で以前に保存された公開鍵 RSA 公開暗号化コールバックコンテキストを取得できるようにします。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_CTX_SetRsaEncCb](#)
- [wolfSSL_SetRsaEncCtx](#)

Return:

- pointer 成功した場合、呼び出しはコンテキストへの有効なポインタを返します。
- NULL 空のコンテキストに対して返されます。

Example

none

C.52.2.305 function wolfSSL_CTX_SetRsaDecCb

```
void wolfSSL_CTX_SetRsaDecCb(  
    WOLFSSL_CTX * ctx,  
    CallbackRsaDec cb  
)
```

呼び出し元が RSA 秘密復号のための公開鍵コールバックを設定できるようにします。コールバックは成功の場合は平文のバイト数を、エラーの場合は 0 未満を返す必要があります。ssl と ctx ポインタはユーザーの利便性のために利用可能です。in は復号する入力バッファで、inSz は入力の高さを示します。out は復号処理とパディングの後、復号バッファの先頭に設定する必要があります。keyDer は ASN1 形式の RSA 秘密鍵で、keySz はバイト単位の鍵の高さです。コールバックの例は wolfssl/test.h の myRsaDec() にあります。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_SetRsaDecCtx](#)
- [wolfSSL_GetRsaDecCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.306 function wolfSSL_SetRsaDecCtx

```
void wolfSSL_SetRsaDecCtx(  
    WOLFSSL * ssl,  
    void * ctx  
)
```

呼び出し元が公開鍵 RSA 秘密復号コールバックコンテキストを ctx に設定できるようにします。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_CTX_SetRsaDecCb](#)
- [wolfSSL_GetRsaDecCtx](#)

Return: none 戻り値なし。

Example

none

C.52.2.307 function wolfSSL_GetRsaDecCtx

```
void * wolfSSL_GetRsaDecCtx(  
    WOLFSSL * ssl  
)
```

呼び出し元が wolfSSL_SetRsaDecCtx() で以前に保存された公開鍵 RSA 秘密復号コールバックコンテキストを取得できるようにします。

Parameters:

- **none** パラメータなし。

See:

- [wolfSSL_CTX_SetRsaDecCb](#)
- [wolfSSL_SetRsaDecCtx](#)

Return:

- pointer 成功した場合、呼び出しはコンテキストへの有効なポインタを返します。
- NULL 空のコンテキストに対して返されます。

Example

none

C.52.2.308 function wolfSSL_CTX_SetCACb

```
void wolfSSL_CTX_SetCACb(
    WOLFSSL_CTX * ctx,
    CallbackCACache cb
)
```

この関数は、新しい CA 証明書が wolfSSL にロードされたときに呼び出されるコールバックを SSL コンテキスト (WOLFSSL_CTX) に登録します。コールバックには DER エンコードされた証明書を含むバッファが渡されます。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された SSL コンテキストへのポインタ。
- **callback** wolfSSL コンテキスト ctx の CA コールバックとして登録される関数。この関数のシグネチャは、上記の概要セクションに示されているものに従う必要があります。

See: `wolfSSL_CTX_load_verify_locations`

Return: none 戻り値なし。

Example

```
WOLFSSL_CTX* ctx = 0;

// CA コールバックのプロトタイプ
int MyCACallback(unsigned char *der, int sz, int type);

// SSL コンテキストにカスタム CA コールバックを登録
wolfSSL_CTX_SetCACb(ctx, MyCACallback);

int MyCACallback(unsigned char* der, int sz, int type)
{
    // カスタム CA コールバック関数、DER エンコードされた証明書は
    // サイズ sz の "der" に格納され、タイプは "type"
}
```

C.52.2.309 function wolfSSL_CertManagerNew_ex

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew_ex(
    void * heap
)
```

新しい証明書マネージャーコンテキストを割り当て、初期化します。このコンテキストは SSL の必要性とは独立して使用できます。証明書のロード、証明書の検証、失効ステータスのチェックに使用できます。

Parameters:

- **none** パラメータなし。

See: `wolfSSL_CertManagerFree`

Return:

- WOLFSSL_CERT_MANAGER 成功した場合、呼び出しは有効な WOLFSSL_CERT_MANAGER ポインタを返します。
- NULL エラー状態の場合に返されます。

C.52.2.310 function wolfSSL_CertManagerNew


```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew(
    void
)
```

新しい証明書マネージャーコンテキストを割り当て、初期化します。このコンテキストは SSL の必要性とは独立して使用できます。証明書のロード、証明書の検証、失効ステータスのチェックに使用できます。

Parameters:

- **none** パラメータなし。

See: `wolfSSL_CertManagerFree`

Return:

- WOLFSSL_CERT_MANAGER 成功した場合、呼び出しは有効な WOLFSSL_CERT_MANAGER ポインタを返します。
- NULL エラー状態の場合に返されます。

Example

```
#import <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
cm = wolfSSL_CertManagerNew();
if (cm == NULL) {
    // 新しい証明書マネージャーの作成エラー
}
```

C.52.2.311 function `wolfSSL_CertManagerFree`

```
void wolfSSL_CertManagerFree(
    WOLFSSL_CERT_MANAGER * cm
)
```

証明書マネージャーコンテキストに関連するすべてのリソースを解放します。証明書マネージャーを使用する必要がなくなったときにこれを呼び出します。

Parameters:

- **cm** `wolfSSL_CertManagerNew()` を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: `wolfSSL_CertManagerNew`

Return: none 戻り値なし。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
...
wolfSSL_CertManagerFree(cm);
```

C.52.2.312 function `wolfSSL_CertManagerLoadCA`

```
int wolfSSL_CertManagerLoadCA(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    const char * d
)
```

マネージャーコンテキストへの CA 証明書ロードの場所を指定します。PEM 証明書 CAfile には複数の信頼された CA 証明書を含めることができます。CApath が NULL でない場合、PEM 形式の CA 証明書を含むディレクトリを指定します。

Parameters:

- **cm** `wolfSSL_CertManagerNew()`を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **file** ロードする CA 証明書を含むファイル名へのポインタ。
- **path** ロードする CA 証明書を含むディレクトリパスの名前へのポインタ。証明書ディレクトリが不要な場合は NULL ポインタを使用できます。

See: `wolfSSL_CertManagerVerify`

Return:

- SSL_SUCCESS 成功した場合に返されます。
- SSL_BAD_FILETYPE ファイルの形式が間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足の状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードが失敗した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラーです。
- SSL_FATAL_ERROR 失敗時に返されます。

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) {
    // 証明書マネージャーへの CA 証明書のロードエラー
}
```

C.52.2.313 function `wolfSSL_CertManagerLoadCABuffer`

```
int wolfSSL_CertManagerLoadCABuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int format
)
```

`wolfSSL_CTX_load_verify_buffer` を呼び出し、関数に渡された cm の情報を失わないように一時的な cm を使用してその結果を返すことで、CA バッファをロードします。

Parameters:

- **cm** `wolfSSL_CertManagerNew()`を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **in** 証明書情報のバッファ。
- **sz** バッファの長さ。
- **format** 証明書の形式、PEM または DER。

See:

- `wolfSSL_CTX_load_verify_buffer`
- `ProcessChainBuffer`

- ProcessBuffer
- cm_pick_method

Return:

- SSL_FATAL_ERROR WOLFSSL_CERT_MANAGER 構造体が NULL、または wolfSSL_CTX_new() が NULL を返した場合に返されます。
- SSL_SUCCESS 正常に実行された場合に返されます。

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){
    // エラーが返されました。失敗ケースのコードブロック。
}
```

C.52.2.314 function wolfSSL_CertManagerUnloadCAs

```
int wolfSSL_CertManagerUnloadCAs(
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は CA 署名者リストをアンロードします。

Parameters:

- **cm** `wolfSSL_CertManagerNew()` を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: UnlockMutex

Return:

- SSL_SUCCESS 関数の正常な実行時に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER が NULL の場合に返されます。
- BAD_MUTEX_E mutex エラーが発生した場合に返されます。

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){
    // 失敗ケース。
}
```

C.52.2.315 function wolfSSL_CertManagerUnloadIntermediateCerts

```
int wolfSSL_CertManagerUnloadIntermediateCerts(
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は CA 署名者リストに追加された中間証明書をアンロードします。

Parameters:

- **cm** `wolfSSL_CertManagerNew()`を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: UnlockMutex

Return:

- SSL_SUCCESS 関数の正常な実行時に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER が NULL の場合に返されます。
- BAD_MUTEX_E mutex エラーが発生した場合に返されます。

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){
    // 失敗ケース。
}
```

C.52.2.316 function wolfSSL_CertManagerUnload_trust_peers

```
int wolfSSL_CertManagerUnload_trust_peers(
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は、Trusted Peer リンクリストを解放し、トラステッドピアリストのロックを解除します。

Parameters:

- **cm** `wolfSSL_CertManagerNew()`を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: UnLockMutex

Return:

- SSL_SUCCESS 関数が正常に完了した場合。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER が NULL の場合。
- BAD_MUTEX_E WOLFSSL_CERT_MANAGER 構造体のメンバである tpLock が 0(null) の場合、mutex エラー。

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){
    // 関数は正常に実行されませんでした。
}
```

C.52.2.317 function wolfSSL_CertManagerVerify

```
int wolfSSL_CertManagerVerify(
    WOLFSSL_CERT_MANAGER * cm,
```

```

    const char * f,
    int format
)

```

Certificate Manager コンテキストで検証する証明書を指定します。フォーマットは `SSL_FILETYPE_PEM` または `SSL_FILETYPE_ASN1` を指定できます。

Parameters:

- **cm** `wolfSSL_CertManagerNew()` を使用して作成された `WOLFSSL_CERT_MANAGER` 構造体へのポインタ。
- **fname** 検証する証明書を含むファイルの名前へのポインタ。
- **format** 検証する証明書のフォーマット - `SSL_FILETYPE_ASN1` または `SSL_FILETYPE_PEM`。

See:

- `wolfSSL_CertManagerLoadCA`
- `wolfSSL_CertManagerVerifyBuffer`

Return:

- `SSL_SUCCESS` 成功時。
- `ASN_SIG_CONFIRM_E` 署名を検証できなかった場合に返されます。
- `ASN_SIG_OID_E` 署名タイプがサポートされていない場合に返されます。
- `CRL_CERT_REVOKED` この証明書が失効している場合に返されるエラー。
- `CRL_MISSING` 現在の発行者 CRL が利用できない場合に返されるエラー。
- `ASN_BEFORE_DATE_E` 現在の日付が before date より前の場合に返されます。
- `ASN_AFTER_DATE_E` 現在の日付が after date より後の場合に返されます。
- `SSL_BAD_FILETYPE` ファイルのフォーマットが間違っている場合に返されます。
- `SSL_BAD_FILE` ファイルが存在しない、読み取れない、または破損している場合に返されます。
- `MEMORY_E` メモリ不足状態が発生した場合に返されます。
- `ASN_INPUT_E` ファイルの Base16 デコードに失敗した場合に返されます。
- `BAD_FUNC_ARG` ポインタが提供されていない場合に返されるエラー。

Example

```

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerVerify(cm, "path/to/cert-file.pem",
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // 証明書の検証エラー
}

```

C.52.2.318 function `wolfSSL_CertManagerVerifyBuffer`

```

int wolfSSL_CertManagerVerifyBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int format
)

```

Certificate Manager コンテキストで検証する証明書バッファを指定します。フォーマットは `SSL_FILETYPE_PEM` または `SSL_FILETYPE_ASN1` を指定できます。

Parameters:

- **cm** wolfSSL_CertManagerNew()を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **buff** 検証する証明書を含むバッファ。
- **sz** バッファ buf のサイズ。
- **format** buf 内にある検証する証明書のフォーマット - SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS 成功時。
- ASN_SIG_CONFIRM_E 署名を検証できなかった場合に返されます。
- ASN_SIG_OID_E 署名タイプがサポートされていない場合に返されます。
- CRL_CERT_REVOKED この証明書が失効している場合に返されるエラー。
- CRL_MISSING 現在の発行者 CRL が利用できない場合に返されるエラー。
- ASN_BEFORE_DATE_E 現在の日付が before date より前の場合に返されます。
- ASN_AFTER_DATE_E 現在の日付が after date より後の場合に返されます。
- SSL_BAD_FILETYPE ファイルのフォーマットが間違っている場合に返されます。
- SSL_BAD_FILE ファイルが存在しない、読み取れない、または破損している場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E ファイルの Base16 デコードに失敗した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラー。

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) {
    // 証明書の検証エラー
}
```

C.52.2.319 function wolfSSL_CertManagerSetVerify

```
void wolfSSL_CertManagerSetVerify(
    WOLFSSL_CERT_MANAGER * cm,
    VerifyCallback verify_callback
)
```

この関数は、Certificate Manager 内に verifyCallback 関数を設定します。存在する場合、ロードされた各証明書に対して呼び出されます。検証エラーがある場合、verify callback を使用してエラーをオーバーライドできます。

Parameters:

- **cm** wolfSSL_CertManagerNew()を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **verify_callback** コールバックルーチンへの VerifyCallback 関数ポインタ。

See: wolfSSL_CertManagerVerify

Return: none 戻り値はありません。

Example

```
#include <wolfssl/ssl.h>

int myVerify(int preverify, WOLFSSL_X509_STORE_CTX* store)
{ // 証明書のカスタム検証を実行 }

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(Protocol define);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
wolfSSL_CertManagerSetVerify(cm, myVerify);
```

C.52.2.320 function wolfSSL_CertManagerCheckCRL

```
int wolfSSL_CertManagerCheckCRL(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * der,
    int sz
)
```

オプションが有効な場合、CRL をチェックし、証明書を CRL リストと比較します。

Parameters:

- **cm** WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **der** DER フォーマットの証明書へのポインタ。
- **sz** 証明書のサイズ。

See:

- CheckCertCRL
- ParseCertRelative
- wolfSSL_CertManagerSetCRL_CB
- InitDecodedCert

Return:

- SSL_SUCCESS 関数が期待どおりに戻った場合に返されます。WOLFSSL_CERT_MANAGER 構造体の crlEnabled メンバがオンになっている場合。
- MEMORY_E 割り当てられたメモリが失敗した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER が NULL の場合。

Example

```
WOLFSSL_CERT_MANAGER* cm;
byte* der;
int sz; // der のサイズ
...
if(wolfSSL_CertManagerCheckCRL(cm, der, sz) != SSL_SUCCESS){
    // エラーが返されました。失敗ケースを処理。
}
```

C.52.2.321 function wolfSSL_CertManagerEnableCRL

```
int wolfSSL_CertManagerEnableCRL(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

Certificate Manager で証明書を検証する際に、Certificate Revocation List チェックをオンにします。デフォルトでは、CRL チェックはオフです。options には、リーフ証明書のみ (デフォルト) ではなくチェーン内の各証明書に対して CRL チェックを実行する WOLFSSL_CRL_CHECKALL が含まれます。

Parameters:

- **cm** `wolfSSL_CertManagerNew()` を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **options** Certification Manager(cm) を有効にする際に使用するオプション。

See: `wolfSSL_CertManagerDisableCRL`

Return:

- SSL_SUCCESS 成功した場合、呼び出しは戻ります。
- NOT_COMPILED_IN wolfSSL が CRL を有効にしてビルドされていない場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラー。
- SSL_FAILURE CRL コンテキストを適切に初期化できない場合に返されます。

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerEnableCRL(cm, 0);
if (ret != SSL_SUCCESS) {
    // cert manager の有効化エラー
}

...
```

C.52.2.322 function `wolfSSL_CertManagerDisableCRL`

```
int wolfSSL_CertManagerDisableCRL(
    WOLFSSL_CERT_MANAGER * cm
)
```

Certificate Manager で証明書を検証する際に、Certificate Revocation List チェックをオフにします。デフォルトでは、CRL チェックはオフです。この関数を使用して、以前に CRL チェックを有効にしていたこの Certificate Manager コンテキストに対して、CRL チェックを一時的または永続的に無効にできます。

Parameters:

- **cm** `wolfSSL_CertManagerNew()` を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: `wolfSSL_CertManagerEnableCRL`

Return:

- SSL_SUCCESS 成功した場合、呼び出しは戻ります。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラー。

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
```



```

WOLFSSL_CERT_MANAGER* cm;
...
ret = wolfSSL_CertManagerDisableCRL(cm);
if (ret != SSL_SUCCESS) {
    // cert manager の無効化エラー
}
...

```

C.52.2.323 function wolfSSL_CertManagerLoadCRL

```

int wolfSSL_CertManagerLoadCRL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * path,
    int type,
    int monitor
)

```

エラーチェックを行い、LoadCRL() に渡して失効チェックのために CRL に証明書をロードします。更新された CRL をロードするには、まず wolfSSL_CertManagerFreeCRL を呼び出してから、新しい CRL をロードします。

Parameters:

- **cm** wolfSSL_CertManagerNew() を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **path** CRL パスを保持する定数 char 型ポインタ。
- **type** ロードする証明書のタイプ。
- **monitor** LoadCRL() での監視を要求します。

See:

- wolfSSL_CertManagerEnableCRL
- wolfSSL_LoadCRL
- wolfSSL_CertManagerFreeCRL

Return:

- SSL_SUCCESS wolfSSL_CertManagerLoadCRL にエラーがなく、LoadCRL が正常に戻った場合。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合。
- SSL_FATAL_ERROR wolfSSL_CertManagerEnableCRL が SSL_SUCCESS 以外を返した場合。
- BAD_PATH_ERROR path が NULL の場合。
- MEMORY_E LoadCRL がヒープメモリの割り当てに失敗した場合。

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.52.2.324 function wolfSSL_CertManagerLoadCRLBuffer

```

int wolfSSL_CertManagerLoadCRLBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,

```

```
    int type
)
```

この関数は、BufferLoadCRL を呼び出して CRL ファイルをロードします。

Parameters:

- **cm** WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **buff** 定数 byte 型でバッファです。
- **sz** バッファのサイズを表す long int 型。
- **type** 証明書タイプを保持する long integer 型。

See:

- BufferLoadCRL
- [wolfSSL_CertManagerEnableCRL](#)

Return:

- SSL_SUCCESS 関数がエラーなく完了した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER が NULL の場合に返されます。
- SSL_FATAL_ERROR WOLFSSL_CERT_MANAGER に関連するエラーがある場合に返されます。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
const unsigned char* buff;
long sz; バッファのサイズ
int type; 証明書タイプ
...
int ret = wolfSSL_CertManagerLoadCRLBuffer(cm, buff, sz, type);
if(ret == SSL_SUCCESS){
    return ret;
} else {
    // 失敗ケース。
}
```

C.52.2.325 function wolfSSL_CertManagerSetCRL_Cb

```
int wolfSSL_CertManagerSetCRL_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbMissingCRL cb
)
```

この関数は、CRL Certificate Manager コールバックを設定します。HAVE_CRL が定義されており、一致する CRL レコードが見つからない場合、cbMissingCRL が呼び出されます (wolfSSL_CertManagerSetCRL_Cb で設定)。これにより、外部から CRL を取得してロードできます。

Parameters:

- **cm** 証明書の情報を保持する WOLFSSL_CERT_MANAGER 構造体。
- **cb** WOLFSSL_CERT_MANAGER の cbMissingCRL メンバに設定される (*CbMissingCRL) への関数ポインタ。

See:

- CbMissingCRL
- [wolfSSL_SetCRL_Cb](#)

Return:

- SSL_SUCCESS 関数とサブルーチンの実行が成功した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
void cb(const char* url){
    // 関数本体。
}
...
CbMissingCRL cb = CbMissingCRL;
...
if(ctx){
    return wolfSSL_CertManagerSetCRL_Cb(SSL_CM(ssl), cb);
}
```

C.52.2.326 function wolfSSL_CertManagerSetCRLUpdate_Cb

```
int wolfSSL_CertManagerSetCRLUpdate_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbUpdateCRL cb
)
```

この関数は、CRL 更新コールバックを設定します。HAVE_CRL と HAVE_CRL_UPDATE_CB が定義されており、CRL が追加されるときに同じ発行者でより低い CRL 番号を持つエントリが存在する場合、既存のエントリと、それを置き換える新しいエントリの詳細と共に CbUpdateCRL が呼び出されます。

Parameters:

- **cm** 証明書の情報を保持する WOLFSSL_CERT_MANAGER 構造体。
- **cb** WOLFSSL_CERT_MANAGER の cbUpdateCRL メンバに設定される (CbUpdateCRL) への関数ポインタ。シグネチャ要件: `void (CbUpdateCRL)(CrlInfo old, CrlInfo new);`

See: CbUpdateCRL

Return:

- SSL_SUCCESS 関数とサブルーチンが正常に実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
void cb(CrlInfo *old, CrlInfo *new){
    // 関数本体。
}
...
CbUpdateCRL cb = CbUpdateCRL;
...
if(ctx){
```

```

    return wolfSSL_CertManagerSetCRLUpdate_Cb(SSL_CM(ssl), cb);
}

```

C.52.2.327 function wolfSSL_CertManagerGetCRLInfo

```

int wolfSSL_CertManagerGetCRLInfo(
    WOLFSSL_CERT_MANAGER * cm,
    CrlInfo * info,
    const byte * buff,
    long sz,
    int type
)

```

この関数は、エンコードされた CRL バッファから解析された CRL 情報を含む構造体を生成します。

Parameters:

- **cm** WOLFSSL_CERT_MANAGER 構造体。
- **info** CRL 情報を受け取る呼び出し元管理の CrlInfo 構造体へのポインタ。
- **buff** エンコードされた CRL を含む入力バッファ。
- **sz** buff 内の入力 CRL データの長さ (バイト単位)。
- **type** WOLFSSL_FILETYPE_PEM または WOLFSSL_FILETYPE_DER
- **cm** [wolfSSL_CertManagerNew\(\)](#) を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See:

- CbUpdateCRL
- [wolfSSL_SetCRL_Cb](#)
- [wolfSSL_CertManagerLoadCRL](#)

Return:

- SSL_SUCCESS 関数とサブルーチンが正常に実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。
- SSL_SUCCESS 関数とサブルーチンが正常に実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。

Example

```
#include <wolfssl/ssl.h>
```

```

CrlInfo info;
WOLFSSL_CERT_MANAGER* cm = NULL;

```

```
cm = wolfSSL_CertManagerNew();
```

```
// ファイルからバッファへ crl データを読み込む
```

```

wolfSSL_CertManagerGetCRLInfo(cm, &info, crlData, crlDataLen,
    WOLFSSL_FILETYPE_PEM);

```

この関数は、証明書マネージャに保存されている CRL を解放します。アプリケーションは、wolfSSL_CertManagerFreeCRL を呼び出してから新しい CRL を読み込むことで、CRL を更新できます。

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.52.2.328 function wolfSSL_CertManagerCheckOCSP

```
int wolfSSL_CertManagerCheckOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * der,
    int sz
)
```

この関数は、WOLFSSL_CERT_MANAGER のメンバである ocsEnabled を有効にして、OCSP チェックオプションが有効になっていることを示します。

Parameters:

- **cm** `wolfSSL_CertManagerNew()` を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **der** 証明書への byte ポインタ。
- **sz** DER 証明書のサイズを表す int 型。

See:

- ParseCertRelative
- CheckCertOCSP

Return:

- SSL_SUCCESS 関数の実行が成功した場合に返されます。WOLFSSL_CERT_MANAGER の ocsEnabled メンバが有効になります。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合、またはサブルーチンに許可されていない引数値が渡された場合に返されます。
- MEMORY_E この関数またはサブルーチン内でメモリの割り当てにエラーがある場合に返されます。

Example

```
#import <wolfssl/ssl.h>

WOLFSSL* ssl = wolfSSL_new(ctx);
byte* der;
int sz; // der のサイズ
...
if(wolfSSL_CertManagerCheckOCSP(cm, der, sz) != SSL_SUCCESS){
    // 失敗ケース。
}
```

C.52.2.329 function wolfSSL_CertManagerEnableOCSP

```
int wolfSSL_CertManagerEnableOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

OCSP がオフになっている場合にオンにし、設定オプションでコンパイルされている場合に有効にします。

Parameters:

- **cm** `wolfSSL_CertManagerNew()`を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **options** WOLFSSL_CERT_MANAGER 構造体の値を設定するために使用されます。

See: `wolfSSL_CertManagerNew`

Return:

- SSL_SUCCESS 関数呼び出しが成功した場合に返されます。
- BAD_FUNC_ARG cm 構造体が NULL の場合。
- MEMORY_E WOLFSSL_OCSP 構造体の値が NULL の場合。
- SSL_FAILURE WOLFSSL_OCSP 構造体の初期化が失敗した場合。
- NOT_COMPILED_IN 正しい機能を有効にしてコンパイルされていないビルド。

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){
    // 失敗ケース。
}
```

C.52.2.330 function `wolfSSL_CertManagerDisableOCSP`

```
int wolfSSL_CertManagerDisableOCSP(
    WOLFSSL_CERT_MANAGER * cm
)
```

OCSP 証明書失効を無効にします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_DisableCRL`

Return:

- SSL_SUCCESS `wolfSSL_CertManagerDisableCRL` が WOLFSSL_CERT_MANAGER 構造体の `crEnabled` メンバを正常に無効にしました。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL でした。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CertManagerDisableOCSP(ssl) != SSL_SUCCESS){
    // 失敗ケース。
}
```

C.52.2.331 function wolfSSL_CertManagerSetOCSPOverrideURL

```
int wolfSSL_CertManagerSetOCSPOverrideURL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * url
)
```

この関数は、URL を WOLFSSL_CERT_MANAGER 構造体の ocsOverrideURL メンバにコピーします。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `ocsOverrideURL`
- `wolfSSL_SetOCSP_OverrideURL`

Return:

- SSL_SUCCESS 関数が期待通りに実行できた場合。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合。
- MEMEORY_E 証明書マネージャの ocsOverrideURL メンバにメモリを割り当てることができなかった場合。

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){
    // 失敗ケース。
}
```

C.52.2.332 function wolfSSL_CertManagerSetOCSP_Cb

```
int wolfSSL_CertManagerSetOCSP_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

この関数は、WOLFSSL_CERT_MANAGER 内の OCSP コールバックを設定します。

Parameters:

- **cm** WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **ioCb** CbOCSPIO 型の関数ポインタ。
- **respFreeCb** CbOCSPRespFree 型の関数ポインタ。
- **ioCbCtx** I/O コールバックユーザ登録コンテキストへの void ポインタ変数。

See:

- `wolfSSL_CertManagerSetOCSPOverrideURL`
- `wolfSSL_CertManagerCheckOCSP`
- `wolfSSL_CertManagerEnableOCSPStapling`
- `wolfSSL_EnableOCSP`
- `wolfSSL_DisableOCSP`

- `wolfSSL_SetOCSP_Cb`

Return:

- SSL_SUCCESS 実行が成功した場合に返されます。引数は WOLFSSL_CERT_MANAGER 構造体に保存されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER が NULL の場合に返されます。

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.52.2.333 function wolfSSL_CertManagerEnableOCSPStapling

```
int wolfSSL_CertManagerEnableOCSPStapling(
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は、OCSP ステープリングがオンになっていない場合にオンにし、オプションを設定します。

Parameters:

- **cm** WOLFSSL_CTX 構造体のメンバである WOLFSSL_CERT_MANAGER 構造体へのポインタ。

See: `wolfSSL_CTX_EnableOCSPStapling`

Return:

- SSL_SUCCESS エラーがなく、関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合、またはサブルーチンに許可されていない引数値が渡された場合に返されます。
- MEMORY_E メモリの割り当てに問題があった場合に返されます。
- SSL_FAILURE OCSP 構造体の初期化が失敗した場合に返されます。
- NOT_COMPILED_IN wolfSSL が HAVE_CERTIFICATE_STATUS_REQUEST オプションでコンパイルされていない場合に返されます。

Example

```
int wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX* ctx){
...
    return wolfSSL_CertManagerEnableOCSPStapling(ctx->cm);
```

C.52.2.334 function wolfSSL_EnableCRL

```
int wolfSSL_EnableCRL(
    WOLFSSL * ssl,
    int options
)
```

CRL 証明書失効を有効にします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **options** WOLFSSL_CERT_MANAGER 構造体の `crCheckAll` メンバの設定を決定するために使用される整数。

See:

- [wolfSSL_CertManagerEnableCRL](#)
- [InitCRL](#)

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなく返された場合。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合に返されます。
- MEMORY_E メモリの割り当てが失敗した場合に返されます。
- SSL_FAILURE InitCRL 関数が正常に返されなかった場合に返されます。
- NOT_COMPILED_IN コンパイル時に HAVE_CRL が有効になっていませんでした。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if (wolfSSL_EnableCRL(ssl, WOLFSSL_CRL_CHECKALL) != SSL_SUCCESS){
    // 失敗ケース。この関数またはサブルーチンが SSL_SUCCESS を返しませんでした。
}
```

C.52.2.335 function wolfSSL_DisableCRL

```
int wolfSSL_DisableCRL(
    WOLFSSL * ssl
)
```

CRL 証明書失効を無効にします。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_CertManagerDisableCRL](#)
- [wolfSSL_CertManagerDisableOCSP](#)

Return:

- SSL_SUCCESS wolfSSL_CertMangerDisableCRL が WOLFSSL_CERT_MANAGER 構造体の crlEnabled メンバを正常に無効にしました。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL でした。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_DisableCRL(ssl) != SSL_SUCCESS){
    // 失敗ケース
}
```

C.52.2.336 function wolfSSL_LoadCRL

```
int wolfSSL_LoadCRL(
    WOLFSSL * ssl,
    const char * path,
    int type,
    int monitor
)
```

失効チェックのための証明書を読み込むために LoadCRL を最終的に呼び出すラッパー関数です。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **path** `crl` ファイルへのパスを保持する定数文字ポインタ。
- **type** 証明書のタイプを表す整数。
- **monitor** 要求された場合にモニタパスを検証するために使用される整数変数。

See:

- `wolfSSL_CertManagerLoadCRL`
- `wolfSSL_CertManagerEnableCRL`
- `LoadCRL`

Return:

- WOLFSSL_SUCCESS 関数とすべてのサブルーチンがエラーなく実行された場合に返されます。
- SSL_FATAL_ERROR サブルーチンの 1 つが正常に返されなかった場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER または WOLFSSL 構造体が NULL の場合。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* crlPemDir;
...
if(wolfSSL_LoadCRL(ssl, crlPemDir, SSL_FILETYPE_PEM, 0) != SSL_SUCCESS){
    // 失敗ケース。SSL_SUCCESS を返しませんでした。
}
```

C.52.2.337 function wolfSSL_SetCRL_Cb

```
int wolfSSL_SetCRL_Cb(
    WOLFSSL * ssl,
    CbMissingCRL cb
)
```

WOLFSSL_CERT_MANAGER 構造体内の CRL コールバックを設定します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** `CbMissingCRL` への関数ポインタ。

See:

- `CbMissingCRL`
- `wolfSSL_CertManagerSetCRL_Cb`

Return:

- SSL_SUCCESS 関数またはサブルーチンがエラーなく実行された場合に返されます。WOLFSSL_CERT_MANAGER の `cbMissingCRL` メンバが設定されます。
- BAD_FUNC_ARG WOLFSSL または WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
void cb(const char* url) // 必要なシングネチャ
{
    // 関数本体
}
```

```

}
...
int crlCb = wolfSSL_SetCRL_Cb(ssl, cb);
if(crlCb != SSL_SUCCESS){
    // コールバックが正しく設定されませんでした
}

```

C.52.2.338 function wolfSSL_EnableOCSP

```

int wolfSSL_EnableOCSP(
    WOLFSSL * ssl,
    int options
)

```

この関数は、OCSP 証明書検証を有効にします。options の値は、以下のオプションの 1 つ以上を OR 演算することで形成されます。WOLFSSL_OCSP_URL_OVERRIDE - 証明書内の URL の代わりにオーバーライド URL を使用します。オーバーライド URL は wolfSSL_CTX_SetOCSP_OverrideURL() 関数を使用して指定されます。WOLFSSL_OCSP_CHECKALL - すべての OCSP チェックをオンに設定します。WOLFSSL_OCSP_NO_NONCE - OCSP 要求作成時の nonce オプションを設定します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **options** 設定チェックに使用される wolfSSL_CertMangerENableOCSP() に渡される整数型。

See: wolfSSL_CertManagerEnableOCSP

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG この関数またはサブルーチンの引数が無効な引数値を受け取った場合に返されます。
- MEMORY_E 構造体または他の変数のメモリ割り当てでエラーが発生した場合に返されます。
- NOT_COMPILED_IN wolfSSL が HAVE_OCSP オプション付きでコンパイルされていない場合に返されます。

Example

```

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int options; // オプション定数で初期化
...
int ret = wolfSSL_EnableOCSP(ssl, options);
if(ret != SSL_SUCCESS){
    // OCSP が有効になっていません
}

```

C.52.2.339 function wolfSSL_DisableOCSP

```

int wolfSSL_DisableOCSP(
    WOLFSSL * ssl
)

```

OCSP 証明書失効オプションを無効にします。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_CertManagerDisableOCSP

Return:

- SSL_SUCCESS 関数とそのサブルーチンがエラーなしで返された場合に返されます。WOLFSSL_CERT_MANAGER 構造体の ocsEnabled メンバが正常に設定されました。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_DisableOCSP(ssl) != SSL_SUCCESS){
    // エラーで返されました。このブロック内が失敗ケースです
}
```

C.52.2.340 function wolfSSL_SetOCSP_OverrideURL

```
int wolfSSL_SetOCSP_OverrideURL(
    WOLFSSL * ssl,
    const char * url
)
```

この関数は、WOLFSSL_CERT_MANAGER 構造体の ocsOverrideURL メンバを設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **url** WOLFSSL_CERT_MANAGER 構造体の ocsOverrideURL メンバに格納される URL への定数 char ポインタ。

See: [wolfSSL_CertManagerSetOCSPOverrideURL](#)

Return:

- SSL_SUCCESS 関数の実行が成功した場合に返されます。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合、またはサブルーチンに許可されていない引数が渡された場合に返されます。
- MEMORY_E サブルーチンでメモリ割り当てエラーが発生した場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
char url[URLSZ];
...
if(wolfSSL_SetOCSP_OverrideURL(ssl, url)){
    // オーバーライド URL が新しい値に設定されました
}
```

C.52.2.341 function wolfSSL_SetOCSP_Cb

```
int wolfSSL_SetOCSP_Cb(
    WOLFSSL * ssl,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

この関数は、WOLFSSL_CERT_MANAGER 構造体に OCSP コールバックを設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **ioCb** CbOCSPIO 型への関数ポインタ。

- **respFreeCb** CbOCSPRespFree 型への関数ポインタで、レスポンスメモリを解放する呼び出しです。
- **ioCbCtx** CM の ocsplIOCtx メンバに保持される void ポインタ。

See:

- **wolfSSL_CertManagerSetOCSP_Cb**
- CbOCSPIO
- CbOCSPRespFree

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。CM の ocsplIOCb、ocsplRespFreeCb、ocsplIOCtx メンバが設定されます。
- BAD_FUNC_ARG WOLFSSL または WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
int OCSPIO_CB(void* , const char*, int , unsigned char* , int,
unsigned char**){ // このシグネチャが必要です
    // 関数本体
}
...
void OCSPRespFree_CB(void* , unsigned char* ){ // このシグネチャが必要です
    // 関数本体
}
...
void* ioCbCtx;
CbOCSPRespFree CB_OCSPRespFree;

if(wolfSSL_SetOCSP_Cb(ssl, OCSPIO_CB( pass args ), CB_OCSPRespFree,
    ioCbCtx) != SSL_SUCCESS){
    // コールバックが設定されませんでした
}
```

C.52.2.342 function wolfSSL_CTX_EnableCRL

```
int wolfSSL_CTX_EnableCRL(
    WOLFSSL_CTX * ctx,
    int options
)
```

CTX を通じて CRL 証明書検証を有効にします。

Parameters:

- **ssl** **wolfSSL_new()** を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- **wolfSSL_CertManagerEnableCRL**
- InitCRL
- **wolfSSL_CTX_DisableCRL**

Return:

- SSL_SUCCESS この関数とそのサブルーチンがエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG CTX 構造体が NULL の場合、またはサブルーチンで無効な引数が渡された場合に返されます。
- MEMORY_E 関数の実行中にメモリ割り当てエラーが発生した場合に返されます。

- SSL_FAILURE WOLFSSL_CERT_MANAGER の crl メンバが正しく初期化できなかった場合に返されます。
- NOT_COMPILED_IN wolfSSL が HAVE_CRL オプション付きでコンパイルされていません。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CTX_EnableCRL(ssl->ctx, options) != SSL_SUCCESS){
    // 関数が失敗しました
}
```

C.52.2.343 function wolfSSL_CTX_DisableCRL

```
int wolfSSL_CTX_DisableCRL(
    WOLFSSL_CTX * ctx
)
```

この関数は、CTX 構造体で CRL 検証を無効にします。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See: [wolfSSL_CertManagerDisableCRL](#)

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。WOLFSSL_CERT_MANAGER 構造体の crlEnabled メンバが 0 に設定されます。
- BAD_FUNC_ARG CTX 構造体または CM 構造体のいずれかが NULL 値の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CTX_DisableCRL(ssl->ctx) != SSL_SUCCESS){
    // 失敗ケース
}
```

C.52.2.344 function wolfSSL_CTX_LoadCRL

```
int wolfSSL_CTX_LoadCRL(
    WOLFSSL_CTX * ctx,
    const char * path,
    int type,
    int monitor
)
```

この関数は、[wolfSSL_CertManagerLoadCRL\(\)](#) を通じて CRL を WOLFSSL_CTX 構造体にロードします。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **path** 証明書へのパス。
- **type** 証明書のタイプを保持する整数変数。
- **monitor** モニタパスが要求されているかを判定するために使用される整数変数。

See:

- `wolfSSL_CertManagerLoadCRL`
- `LoadCRL`

Return:

- `SSL_SUCCESS` - 関数とそのサブルーチンがエラーなしで実行された場合に返されます。
- `BAD_FUNC_ARG` - この関数またはサブルーチンに `NULL` 構造体が渡された場合に返されます。
- `BAD_PATH_ERROR` - `path` 変数が `NULL` として開かれた場合に返されます。
- `MEMORY_E` - メモリの割り当てが失敗した場合に返されます。

Example

```
WOLFSSL_CTX* ctx;
const char* path;
...
return wolfSSL_CTX_LoadCRL(ctx, path, SSL_FILETYPE_PEM, 0);
```

C.52.2.345 function wolfSSL_CTX_SetCRL_Cb

```
int wolfSSL_CTX_SetCRL_Cb(
    WOLFSSL_CTX * ctx,
    CbMissingCRL cb
)
```

この関数は、`wolfSSL_CertManagerSetCRL_Cb` を呼び出すことにより、コールバック引数を `WOLFSSL_CERT_MANAGER` 構造体の `cbMissingCRL` メンバに設定します。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された `WOLFSSL_CTX` 構造体へのポインタ。
- **cb** `CbMissingCRL` 型のコールバック関数へのポインタ。シグネチャ要件: `void (CbMissingCRL)(const char url);`

See:

- `wolfSSL_CertManagerSetCRL_Cb`
- `CbMissingCRL`

Return:

- `SSL_SUCCESS` 実行が成功した場合に返されます。`WOLFSSL_CERT_MANAGER` 構造体のメンバ `cbMissingCRL` が `cb` に正常に設定されました。
- `BAD_FUNC_ARG` `WOLFSSL_CTX` または `WOLFSSL_CERT_MANAGER` が `NULL` の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
...
void cb(const char* url) // 必要なシグネチャ
{
    // 関数本体
}
...
if (wolfSSL_CTX_SetCRL_Cb(ctx, cb) != SSL_SUCCESS){
    // 失敗ケース、cb が正しく設定されませんでした
}
```

C.52.2.346 function wolfSSL_CTX_EnableOCSP

```
int wolfSSL_CTX_EnableOCSP(
    WOLFSSL_CTX * ctx,
```

```
    int options
)
```

この関数は、wolfSSL の OCSP 機能の動作を設定するオプションを設定します。options の値は、以下のオプションの 1 つ以上を OR 演算することで形成されます。WOLFSSL_OCSP_URL_OVERRIDE - 証明書内の URL の代わりにオーバーライド URL を使用します。オーバーライド URL は wolfSSL_CTX_SetOCSP_OverrideURL() 関数を使用して指定されます。WOLFSSL_OCSP_CHECKALL - すべての OCSP チェックをオンに設定します。WOLFSSL_OCSP_NO_NONCE - OCSP 要求作成時の nonce オプションを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。
- **options** OCSP オプションを設定するために使用される値。

See:

- wolfSSL_CertManagerEnableOCSP
- wolfSSL_EnableOCSP

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 失敗時に返されます。
- NOT_COMPILED_IN この関数が呼び出されたが、wolfSSL のコンパイル時に OCSP サポートが有効になっていなかった場合に返されます。

この関数は、wolfSSL が OCSP サポート付きでコンパイルされている場合 (-enable-ocsp、#define HAVE_OCSP) にのみ OCSP オプションを設定します。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
int options; // オプション定数で初期化
...
int ret = wolfSSL_CTX_EnableOCSP(ctx, options);
if(ret != SSL_SUCCESS){
    // OCSP が有効になっていません
}
```

C.52.2.347 function wolfSSL_CTX_DisableOCSP

```
int wolfSSL_CTX_DisableOCSP(
    WOLFSSL_CTX * ctx
)
```

この関数は、WOLFSSL_CERT_MANAGER 構造体の ocsEnabled メンバに影響を与えることにより、OCSP 証明書失効チェックを無効にします。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_DisableOCSP
- wolfSSL_CertManagerDisableOCSP

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。CM の ocsEnabled メンバが無効にされました。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(!wolfSSL_CTX_DisableOCSP(ssl->ctx)){
    // OCSP が無効になっていません
}
```

C.52.2.348 function wolfSSL_CTX_SetOCSP_OverrideURL

```
int wolfSSL_CTX_SetOCSP_OverrideURL(
    WOLFSSL_CTX * ctx,
    const char * url
)
```

この関数は、OCSP が使用する URL を手動で設定します。デフォルトでは、wolfSSL_CTX_EnableOCSP を使用して WOLFSSL_OCSP_URL_OVERRIDE オプションが設定されていない限り、OCSP は個々の証明書で見つかった URL を使用します。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。
- **url** wolfSSL が使用する OCSP URL へのポインタ。

See: wolfSSL_CTX_OCSP_set_options

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 失敗時に返されます。
- NOT_COMPILED_IN この関数が呼び出されたが、wolfSSL のコンパイル時に OCSP サポートが有効になっていなかった場合に返されます。

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_OCSP_set_override_url(ctx, "custom-url-here");
```

C.52.2.349 function wolfSSL_CTX_SetOCSP_Cb

```
int wolfSSL_CTX_SetOCSP_Cb(
    WOLFSSL_CTX * ctx,
    CbOCSP_IO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

WOLFSSL_CTX 構造体に OCSP のコールバックを設定します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **ioCb** 関数ポインタである CbOCSP_IO 型。
- **respFreeCb** 関数ポインタである CbOCSPRespFree 型。
- **ioCbCtx** WOLFSSL_CERT_MANAGER に保持される void ポインタ。

See:

- wolfSSL_CertManagerSetOCSP_Cb

- CbOCSPiO
- CbOCSPRespFree

Return:

- SSL_SUCCESS 関数が正常に実行された場合に返されます。CM 内の ocsplOCb、ocspRespFreeCb、ocspIOCtx メンバが正常に設定されました。
- BAD_FUNC_ARG WOLFSSL_CTX または WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
...
CbOCSPiO ocsplOCb;
CbOCSPRespFree ocspRespFreeCb;
...
void* ioCbCtx;

int isSetOCSP = wolfSSL_CTX_SetOCSP_Cb(ctx, ocsplOCb,
ocspRespFreeCb, ioCbCtx);

if(isSetOCSP != SSL_SUCCESS){
    // 関数が正常に返されませんでした
}
```

C.52.2.350 function wolfSSL_CTX_EnableOCSPStapling

```
int wolfSSL_CTX_EnableOCSPStapling(
    WOLFSSL_CTX * ctx
)
```

この関数は、wolfSSL_CertManagerEnableOCSPStapling() を呼び出して OCSP ステープリングを有効にします。

Parameters:

- **ctx** wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_CertManagerEnableOCSPStapling
- InitOCSP

Return:

- SSL_SUCCESS エラーがなく、関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL、またはサブルーチンに許可されていない引数値が渡された場合に返されます。
- MEMORY_E メモリの割り当てに問題があった場合に返されます。
- SSL_FAILURE OCSP 構造体の初期化が失敗した場合に返されます。
- NOT_COMPILED_IN wolfSSL が HAVE_CERTIFICATE_STATUS_REQUEST オプション付きでコンパイルされていない場合に返されます。

Example

```
WOLFSSL* ssl = WOLFSSL_new();
ssl->method.version; // 希望するプロトコルに設定
...
if(!wolfSSL_CTX_EnableOCSPStapling(ssl->ctx)){
```

```
    // OCSP ステージングが有効になっていません  
}
```

C.52.2.351 function wolfSSL_KeepArrays

```
void wolfSSL_KeepArrays(  
    WOLFSSL * ssl  
)
```

通常、SSL ハンドシェイクの終了時に、wolfSSL は一時配列を解放します。ハンドシェイクが始まる前にこの関数を呼び出すと、wolfSSL は一時配列を解放しなくなります。一時配列は wolfSSL_get_keys() や PSK ヒントなどに必要になる場合があります。ユーザーが一時的配列を使い終わったら、wolfSSL_FreeArrays() を呼び出してリソースを即座に解放するか、あるいは関連する SSL オブジェクトが解放されるときにリソースが解放されます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_FreeArrays

Return: none 戻り値なし。

Example

```
WOLFSSL* ssl;  
...  
wolfSSL_KeepArrays(ssl);
```

C.52.2.352 function wolfSSL_FreeArrays

```
void wolfSSL_FreeArrays(  
    WOLFSSL * ssl  
)
```

通常、SSL ハンドシェイクの終了時に、wolfSSL は一時配列を解放します。ハンドシェイクの前に wolfSSL_KeepArrays() が呼び出されていた場合、wolfSSL は一時配列を解放しません。この関数は一時配列を明示的に解放し、ユーザーが一時的配列を使い終わり、SSL オブジェクトが解放されるのを待たずにこれらのリソースを解放したい場合に呼び出す必要があります。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_KeepArrays

Return: none 戻り値なし。

Example

```
WOLFSSL* ssl;  
...  
wolfSSL_FreeArrays(ssl);
```

C.52.2.353 function wolfSSL_UseSNI

```
int wolfSSL_UseSNI(  
    WOLFSSL * ssl,  
    unsigned char type,  
    const void * data,
```

```
    unsigned short size
)
```

この関数は、'ssl' パラメータで渡された SSL オブジェクトで Server Name Indication の使用を有効にします。つまり、wolfSSL クライアントは ClientHello で SNI 拡張を送信し、wolfSSL サーバーは SNI 不一致の場合、ClientHello + SNI に対して ServerHello + 空の SNI または fatal アラートで応答します。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL オブジェクトへのポインタ。
- **type** data で渡されるサーバー名のタイプを示します。既知のタイプは、enum { WOLFSSL_SNI_HOST_NAME = 0 } です。
- **data** サーバー名データへのポインタ。
- **size** サーバー名データのサイズ。

See:

- wolfSSL_new
- wolfSSL_CTX_UseSNI

Return:

- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラーです。ssl が NULL、data が NULL、type が未知の値 (以下を参照)。
- MEMORY_E メモリが不足している場合に返されるエラーです。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキスト作成失敗
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl 作成失敗
}
ret = wolfSSL_UseSNI(ssl, WOLFSSL_SNI_HOST_NAME, "www.yassl.com",
    strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni 使用失敗
}
```

C.52.2.354 function wolfSSL_CTX_UseSNI

```
int wolfSSL_CTX_UseSNI(
    WOLFSSL_CTX * ctx,
    unsigned char type,
    const void * data,
    unsigned short size
)
```

この関数は、'ctx' パラメータで渡された SSL コンテキストから作成された SSL オブジェクトに対して Server Name Indication の使用を有効にします。つまり、wolfSSL クライアントは ClientHello で SNI 拡張を送信し、wolfSSL サーバーは SNI 不一致の場合、ClientHello + SNI に対して ServerHello + 空の SNI または fatal アラートで応答します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された SSL コンテキストへのポインタ。
- **type** data で渡されるサーバー名のタイプを示します。既知のタイプは、enum { WOLFSSL_SNI_HOST_NAME = 0 } です。
- **data** サーバー名データへのポインタ。
- **size** サーバー名データのサイズ。

See:

- `wolfSSL_CTX_new`
- `wolfSSL_UseSNI`

Return:

- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラーです。ctx が NULL、data が NULL、type が未知の値 (以下を参照)。
- MEMORY_E メモリが不足している場合に返されるエラーです。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキスト作成失敗
}
ret = wolfSSL_CTX_UseSNI(ctx, WOLFSSL_SNI_HOST_NAME, "www.yassl.com",
    strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni 使用失敗
}
```

C.52.2.355 function wolfSSL_SNI_SetOptions

```
void wolfSSL_SNI_SetOptions(
    WOLFSSL * ssl,
    unsigned char type,
    unsigned char options
)
```

この関数は、'ssl' パラメータで渡された SSL オブジェクトで Server Name Indication を使用する SSL セッションの動作を設定するために、サーバー側で呼び出されます。オプションについては以下で説明します。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL オブジェクトへのポインタ。
- **type** data で渡されるサーバー名のタイプを示します。既知のタイプは、enum { WOLFSSL_SNI_HOST_NAME = 0 } です。
- **options** 選択されたオプションを含むビット単位のセマフォ。利用可能なオプションは、enum { WOLFSSL_SNI_CONTINUE_ON_MISMATCH = 0x01, WOLFSSL_SNI_ANSWER_ON_MISMATCH = 0x02 } です。通常、クライアントから提供されたホスト名がサーバーと一致しない場合、サーバーは fatal レベルの unrecognized_name(112) アラートを送信してハンドシェイクを中止します。
- **WOLFSSL_SNI_CONTINUE_ON_MISMATCH** このオプションが設定されている場合、サーバーはセッションを中止する代わりに SNI 応答を送信しません。
- **WOLFSSL_SNI_ANSWER_ON_MISMATCH** このオプションが設定されている場合、サーバーはセッションを中止する代わりに、ホスト名が一致しているかのように SNI 応答を送信します。

See:

- `wolfSSL_new`
- `wolfSSL_UseSNI`
- `wolfSSL_CTX_SNI_SetOptions`

Return: none 戻り値なし。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキスト作成失敗
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl 作成失敗
}
ret = wolfSSL_UseSNI(ssl, 0, "www.yassl.com", strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni 使用失敗
}
wolfSSL_SNI_SetOptions(ssl, WOLFSSL_SNI_HOST_NAME,
    WOLFSSL_SNI_CONTINUE_ON_MISMATCH);
```

C.52.2.356 function `wolfSSL_CTX_SNI_SetOptions`

```
void wolfSSL_CTX_SNI_SetOptions(
    WOLFSSL_CTX * ctx,
    unsigned char type,
    unsigned char options
)
```

この関数は、'ctx' パラメータで渡された SSL コンテキストから作成された SSL オブジェクトに対して Server Name Indication を使用する SSL セッションの動作を設定するために、サーバー側で呼び出されます。オプションについては以下で説明します。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **type** data で渡されるサーバー名のタイプを示します。既知のタイプは、enum { WOLFSSL_SNI_HOST_NAME = 0 } です。
- **options** 選択されたオプションを含むビット単位のセマフォ。利用可能なオプションは、enum { WOLFSSL_SNI_CONTINUE_ON_MISMATCH = 0x01, WOLFSSL_SNI_ANSWER_ON_MISMATCH = 0x02 } です。通常、クライアントから提供されたホスト名がサーバーと一致しない場合、サーバーは fatal レベルの unrecognized_name(112) アラートを送信してハンドシェイクを中止します。
- **WOLFSSL_SNI_CONTINUE_ON_MISMATCH** このオプションが設定されている場合、サーバーはセッションを中止する代わりに SNI 応答を送信しません。
- **WOLFSSL_SNI_ANSWER_ON_MISMATCH** このオプションが設定されている場合、サーバーはセッションを中止する代わりに、ホスト名が一致しているかのように SNI 応答を送信します。

See:

- `wolfSSL_CTX_new`
- `wolfSSL_CTX_UseSNI`
- `wolfSSL_SNI_SetOptions`

Return: none 戻り値なし。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキスト作成失敗
}
ret = wolfSSL_CTX_UseSNI(ctx, 0, "www.yassl.com", strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni 使用失敗
}
wolfSSL_CTX_SNI_SetOptions(ctx, WOLFSSL_SNI_HOST_NAME,
WOLFSSL_SNI_CONTINUE_ON_MISMATCH);
```

C.52.2.357 function wolfSSL_SNI_GetFromBuffer

```
int wolfSSL_SNI_GetFromBuffer(
    const unsigned char * clientHello,
    unsigned int helloSz,
    unsigned char type,
    unsigned char * sni,
    unsigned int * inOutSz
)
```

この関数は、クライアントがセッションを開始するために送信した Client Hello メッセージからクライアントによって提供された Server Name Indication を取得するために、サーバー側で呼び出されます。SNI を取得するためにコンテキストやセッションのセットアップは必要ありません。

Parameters:

- **buffer** クライアントから提供されたデータ (Client Hello) へのポインタ。
- **bufferSz** Client Hello メッセージのサイズ。
- **type** buffer から取得されるサーバー名のタイプを示します。既知のタイプは、enum { WOLFSSL_SNI_HOST_NAME = 0 } です。
- **sni** 出力が格納される場所へのポインタ。
- **inOutSz** 出力サイズへのポインタ。この値は MIN("SNI の長さ", inOutSz) に更新されます。

See:

- [wolfSSL_UseSNI](#)
- [wolfSSL_CTX_UseSNI](#)
- [wolfSSL_SNI_GetRequest](#)

Return:

- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラーです。buffer が NULL、bufferSz <= 0、sni が NULL、inOutSz が NULL または <= 0。
- BUFFER_ERROR 不正な形式の Client Hello メッセージがある場合に返されるエラーです。
- INCOMPLETE_DATA 抽出を完了するのに十分なデータがない場合に返されるエラーです。

Example

```
unsigned char buffer[1024] = {0};
unsigned char result[32] = {0};
int length = 32;
// Client Hello を buffer に読み込み...
```

```
ret = wolfSSL_SNI_GetFromBuffer(buffer, sizeof(buffer), 0, result, &length);
if (ret != WOLFSSL_SUCCESS) {
    // sni 取得失敗
}
```

C.52.2.358 function wolfSSL_SNI_Status

```
unsigned char wolfSSL_SNI_Status(
    WOLFSSL * ssl,
    unsigned char type
)
```

この関数は SNI オブジェクトのステータスを取得します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **type** SNI タイプ。

See:

- TLSX_SNI_Status
- TLSX_SNI_find
- TLSX_Find

Return:

- value SNI が NULL でない場合、この関数は SNI 構造体の status メンバーのバイト値を返します。
- 0 SNI オブジェクトが NULL の場合。

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.52.2.359 function wolfSSL_SNI_GetRequest

```
unsigned short wolfSSL_SNI_GetRequest(
    WOLFSSL * ssl,
    unsigned char type,
    void ** data
)
```

この関数は、SSL セッションでクライアントによって提供された Server Name Indication を取得するために、サーバー側で呼び出されます。

Parameters:

- **ssl** `wolfSSL_new()` で作成された SSL オブジェクトへのポインタ。
- **type** data で取得されるサーバー名のタイプを示します。既知のタイプは、enum { WOLFSSL_SNI_HOST_NAME = 0 } です。
- **data** クライアントから提供されたデータへのポインタ。

See:

- wolfSSL_UseSNI
- wolfSSL_CTX_UseSNI

Return: size 提供された SNI データのサイズ。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキスト作成失敗
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl 作成失敗
}
ret = wolfSSL_UseSNI(ssl, 0, "www.yassl.com", strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni 使用失敗
}
if (wolfSSL_accept(ssl) == SSL_SUCCESS) {
    void *data = NULL;
    unsigned short size = wolfSSL_SNI_GetRequest(ssl, 0, &data);
}
```

C.52.2.360 function wolfSSL_UseALPN

```
int wolfSSL_UseALPN(
    WOLFSSL * ssl,
    char * protocol_name_list,
    unsigned int protocol_name_listSz,
    unsigned char options
)
```

wolfSSL セッションで ALPN の使用をセットアップします。

Parameters:

- **ssl** 使用する wolfSSL セッション。
- **protocol_name_list** 使用するプロトコル名のリスト。カンマ区切りの文字列が必要です。
- **protocol_name_listSz** プロトコル名リストのサイズ。
- **options** WOLFSSL_ALPN_CONTINUE_ON_MISMATCH または WOLFSSL_ALPN_FAILED_ON_MISMATCH。

See: TLSX_UseALPN

Return:

- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG ssl または protocol_name_list が null、または protocol_name_listSz が大きすぎる、または options がサポートされていない何かを含む場合に返されます。
- MEMORY_ERROR プロトコルリストのメモリ割り当てエラー。
- SSL_FAILURE 失敗時。

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
```

```

WOLFSSL_METHOD method = // 任意の wolfSSL メソッド
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)
{
    // セッションチケット設定エラー
}

```

C.52.2.361 function wolfSSL_ALPN_GetProtocol

```

int wolfSSL_ALPN_GetProtocol(
    WOLFSSL * ssl,
    char ** protocol_name,
    unsigned short * size
)

```

この関数は、サーバーによって設定されたプロトコル名を取得します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **protocol_name** プロトコル名を表し、ALPN 構造体に保持される char へのポインタ。
- **size** protocol_name のサイズを表す word16 型。

See:

- TLSX_ALPN_GetRequest
- TLSX_Find

Return:

- SSL_SUCCESS エラーがスローされずに正常に実行された場合に返されます。
- SSL_FATAL_ERROR 拡張が見つからなかった場合、またはピアとのプロトコルマッチがなかった場合に返されます。また、受け入れられたプロトコル名が複数ある場合にもエラーがスローされます。
- SSL_ALPN_NOT_FOUND ピアとのプロトコルマッチが見つからなかったことを示して返されます。
- BAD_FUNC_ARG 関数に NULL 引数が渡された場合に返されます。

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){
    // ALPN プロトコルを送信
}

```

C.52.2.362 function wolfSSL_ALPN_GetPeerProtocol

```

int wolfSSL_ALPN_GetPeerProtocol(
    WOLFSSL * ssl,
    char ** list,

```

```
    unsigned short * listSz
)
```

この関数は、SSL オブジェクトから alpn_client_list データをバッファにコピーします。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **list** バッファへのポインタ。SSL オブジェクトからのデータがここにコピーされます。
- **listSz** バッファサイズ。

See: wolfSSL_UseALPN

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。SSL オブジェクトの alpn_client_list メンバが list パラメータにコピーされました。
- BAD_FUNC_ARG list または listSz パラメータが NULL の場合に返されます。
- BUFFER_ERROR list バッファに問題がある場合に返されます (NULL であるか、サイズが 0 の場合)。
- MEMORY_ERROR メモリの動的割り当てに問題があった場合に返されます。

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){
    // クライアントが送信したプロトコル名のリスト
}
```

C.52.2.363 function wolfSSL_UseMaxFragment

```
int wolfSSL_UseMaxFragment(
    WOLFSSL * ssl,
    unsigned char mfl
)
```

この関数は、'ssl' パラメータで渡された SSL オブジェクトで Maximum Fragment Length の使用を有効にするために、クライアント側で呼び出されます。これは、wolfSSL クライアントによって ClientHello で Maximum Fragment Length 拡張が送信されることを意味します。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL オブジェクトへのポインタ。
- **mfl** セッションに要求される Maximum Fragment Length を示します。利用可能なオプションは: enum { WOLFSSL_MFL_2_9 = 1, 512 バイト WOLFSSL_MFL_2_10 = 2, 1024 バイト WOLFSSL_MFL_2_11 = 3, 2048 バイト WOLFSSL_MFL_2_12 = 4, 4096 バイト WOLFSSL_MFL_2_13 = 5, 8192 バイト wolfSSL 専用!!! };

See:

- wolfSSL_new

- `wolfSSL_CTX_UseMaxFragment`

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラー: ssl が NULL、mfl が範囲外。
- MEMORY_E メモリが不足している場合に返されるエラー。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキストの作成に失敗
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl の作成に失敗
}
ret = wolfSSL_UseMaxFragment(ssl, WOLFSSL_MFL_2_11);
if (ret != 0) {
    // max fragment の使用に失敗
}
```

C.52.2.364 function wolfSSL_CTX_UseMaxFragment

```
int wolfSSL_CTX_UseMaxFragment(
    WOLFSSL_CTX * ctx,
    unsigned char mfl
)
```

この関数は、'ctx' パラメータで渡された SSL コンテキストから作成された SSL オブジェクトに対して Maximum Fragment Length の使用を有効にするために、クライアント側で呼び出されます。これは、wolfSSL クライアントによって ClientHello で Maximum Fragment Length 拡張が送信されることを意味します。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **mfl** セッションに要求される Maximum Fragment Length を示します。利用可能なオプションは: enum { WOLFSSL_MFL_2_9 = 1 512 バイト、WOLFSSL_MFL_2_10 = 2 1024 バイト、WOLFSSL_MFL_2_11 = 3 2048 バイト WOLFSSL_MFL_2_12 = 4 4096 バイト、WOLFSSL_MFL_2_13 = 5 8192 バイト wolfSSL 専用!!!、WOLFSSL_MFL_2_13 = 6 256 バイト wolfSSL 専用!!! };

See:

- `wolfSSL_CTX_new`
- `wolfSSL_UseMaxFragment`

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラー: ctx が NULL、mfl が範囲外。
- MEMORY_E メモリが不足している場合に返されるエラー。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
```

```

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキストの作成に失敗
}
ret = wolfSSL_CTX_UseMaxFragment(ctx, WOLFSSL_MFL_2_11);
if (ret != 0) {
    // max fragment の使用に失敗
}

```

C.52.2.365 function wolfSSL_UseTruncatedHMAC

```

int wolfSSL_UseTruncatedHMAC(
    WOLFSSL * ssl
)

```

この関数は、'ssl' パラメータで渡された SSL オブジェクトで Truncated HMAC の使用を有効にするために、クライアント側で呼び出されます。これは、wolfSSL クライアントによって ClientHello で Truncated HMAC 拡張が送信されることを意味します。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL オブジェクトへのポインタ。

See:

- wolfSSL_new
- wolfSSL_CTX_UseMaxFragment

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラー: ssl が NULL。
- MEMORY_E メモリが不足している場合に返されるエラー。

Example

```

int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキストの作成に失敗
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl の作成に失敗
}
ret = wolfSSL_UseTruncatedHMAC(ssl);
if (ret != 0) {
    // truncated HMAC の使用に失敗
}

```

C.52.2.366 function wolfSSL_CTX_UseTruncatedHMAC

```

int wolfSSL_CTX_UseTruncatedHMAC(
    WOLFSSL_CTX * ctx
)

```

この関数は、'ctx' パラメータで渡された SSL コンテキストから作成された SSL オブジェクトに対して Truncated HMAC の使用を有効にするために、クライアント側で呼び出されます。これは、wolfSSL クライアントによって ClientHello で Truncated HMAC 拡張が送信されることを意味します。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された SSL コンテキストへのポインタ。

See:

- wolfSSL_CTX_new
- wolfSSL_UseMaxFragment

Return:

- SSL_SUCCESS 成功時。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラー: ctx が NULL。
- MEMORY_E メモリが不足している場合に返されるエラー。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキストの作成に失敗
}
ret = wolfSSL_CTX_UseTruncatedHMAC(ctx);
if (ret != 0) {
    // truncated HMAC の使用に失敗
}
```

C.52.2.367 function wolfSSL_UseOCSPStapling

```
int wolfSSL_UseOCSPStapling(
    WOLFSSL * ssl,
    unsigned char status_type,
    unsigned char options
)
```

ステープリングは CA に連絡する必要性を排除します。ステープリングは、OCSP で提示される証明書失効チェックのコストを削減します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **status_type** TLSX_UseCertificateStatusRequest() に渡され、CertificateStatusRequest 構造体に格納されるバイト型。
- **options** TLSX_UseCertificateStatusRequest() に渡され、CertificateStatusRequest 構造体に格納されるバイト型。

See:

- TLSX_UseCertificateStatusRequest
- wolfSSL_CTX_UseOCSPStapling

Return:

- SSL_SUCCESS TLSX_UseCertificateStatusRequest がエラーなく実行された場合に返されます。
- MEMORY_E メモリの割り当てにエラーがある場合に返されます。
- BAD_FUNC_ARG 関数に渡された引数が NULL または許容できない値である場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if (wolfSSL_UseOCSPStapling(ssl, WOLFSSL_CSR2_OCSP,
    WOLFSSL_CSR2_OCSP_USE_NONCE) != SSL_SUCCESS){
    // 失敗ケース
}
```

C.52.2.368 function wolfSSL_CTX_UseOCSPStapling

```
int wolfSSL_CTX_UseOCSPStapling(
    WOLFSSL_CTX * ctx,
    unsigned char status_type,
    unsigned char options
)
```

この関数は、ハンドシェイク中に証明書ステータスを要求します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **status_type** TLSX_UseCertificateStatusRequest() に渡され、CertificateStatusRequest 構造体に格納されるバイト型。
- **options** TLSX_UseCertificateStatusRequest() に渡され、CertificateStatusRequest 構造体に格納されるバイト型。

See:

- [wolfSSL_UseOCSPStaplingV2](#)
- [wolfSSL_UseOCSPStapling](#)
- [TLSX_UseCertificateStatusRequest](#)

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなく実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合、またはサブルーチンに許可されていない値が渡された場合に返されます。
- MEMORY_E 関数またはサブルーチンがメモリの適切な割り当てに失敗した場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte statusRequest = 0; // ステータスリクエストを初期化
...
switch(statusRequest){
    case WOLFSSL_CSR_OCSP:
        if(wolfSSL_CTX_UseOCSPStapling(ssl->ctx, WOLFSSL_CSR_OCSP,
            WOLF_CSR_OCSP_USE_NONCE) != SSL_SUCCESS){
            // UseCertificateStatusRequest が失敗
        }
    // switch ケースを続ける
```

C.52.2.369 function wolfSSL_UseOCSPStaplingV2

```
int wolfSSL_UseOCSPStaplingV2(
    WOLFSSL * ssl,
    unsigned char status_type,
```

```
    unsigned char options
)
```

この関数は、OCSP のステータスタイプとオプションを設定します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **status_type** OCSP ステータスタイプをロードするバイト型。
- **options** `wolfSSL_SNI_SetOptions()` と `wolfSSL_CTX_SNI_SetOptions()` で設定される OCSP オプションを保持するバイト型。

See:

- TLSX_UseCertificateStatusRequestV2
- `wolfSSL_SNI_SetOptions`
- `wolfSSL_CTX_SNI_SetOptions`

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなく実行された場合に返されます。
- MEMORY_E メモリ割り当てエラーがあった場合に返されます。
- BAD_FUNC_ARG 関数またはサブルーチンに NULL または許容されない引数が渡された場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if (wolfSSL_UseOCSPStaplingV2(ssl, WOLFSSL_CSR2_OCSP_MULTI, 0) != SSL_SUCCESS){
    // 正しく実行されませんでした。失敗ケースのコードブロック
}
```

C.52.2.370 function wolfSSL_CTX_UseOCSPStaplingV2

```
int wolfSSL_CTX_UseOCSPStaplingV2(
    WOLFSSL_CTX * ctx,
    unsigned char status_type,
    unsigned char options
)
```

OCSP ステープリング用の証明書ステータス要求を作成して初期化します。

Parameters:

- **ctx** `wolfSSL_CTX_new()` を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **status_type** CertificateStatusRequest 構造体にあるバイト型で、WOLFSSL_CSR2_OCSP または WOLFSSL_CSR2_OCSP_MULTI のいずれかである必要があります。
- **options** CertificateStatusRequestItemV2 構造体に保持されるバイト型。

See:

- TLSX_UseCertificateStatusRequestV2
- `wc_RNG_GenerateBlock`
- TLSX_Push

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなく実行された場合。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL、または side 変数がクライアント側でない場合に返されます。
- MEMORY_E メモリの割り当てに失敗した場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
byte status_type;
byte options;
...
if(wolfSSL_CTX_UseOCSPStaplingV2(ctx, status_type, options); != SSL_SUCCESS){
    // 失敗ケース
}
```

C.52.2.371 function wolfSSL_UseSupportedCurve

```
int wolfSSL_UseSupportedCurve(
    WOLFSSL * ssl,
    word16 name
)
```

この関数は、'ssl' パラメータで渡された SSL オブジェクトで Supported Elliptic Curves 拡張の使用を有効にするために、クライアント側で呼び出されます。これは、wolfSSL クライアントによって ClientHello で有効化されたサポートされる曲線が送信されることを意味します。この関数は、複数の曲線を有効にするために複数回呼び出すことができます。

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL オブジェクトへのポインタ。
- **name** セッションでサポートされる曲線を示します。利用可能なオプションは以下の通りです: 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 成功時。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラー: ssl が NULL、name が不明な値 (下記参照)。
- MEMORY_E メモリが不足している場合に返されるエラー。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキストの作成に失敗しました。
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl の作成に失敗しました。
}
ret = wolfSSL_UseSupportedCurve(ssl, WOLFSSL_ECC_SECP256R1);
if (ret != 0) {
    // 楕円曲線拡張の使用に失敗しました。
}
```

C.52.2.372 function wolfSSL_CTX_UseSupportedCurve

```
int wolfSSL_CTX_UseSupportedCurve(
    WOLFSSL_CTX * ctx,
    word16 name
)
```

この関数は、ctx パラメータで渡された SSL コンテキストから作成された SSL オブジェクトに対して、サポートされる楕円曲線拡張の使用を有効にするためにクライアント側で呼び出されます。これは、有効にされたサポートされる曲線が wolfSSL クライアントによって ClientHello で送信されることを意味します。この関数は、複数の曲線を有効にするために複数回呼び出すことができます。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された SSL コンテキストへのポインタ。
- **name** セッションでサポートされる曲線を示します。利用可能なオプションは以下の通りです: 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 成功時。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラー: ctx が NULL、name が不明な値 (下記参照)。
- MEMORY_E メモリが不足している場合に返されるエラー。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // コンテキストの作成に失敗しました。
}
ret = wolfSSL_CTX_UseSupportedCurve(ctx, WOLFSSL_ECC_SECP256R1);
if (ret != 0) {
    // 楕円曲線拡張の使用に失敗しました。
}
```

C.52.2.373 function wolfSSL_UseSecureRenegotiation

```
int wolfSSL_UseSecureRenegotiation(
    WOLFSSL * ssl
)
```

この関数は、提供された WOLFSSL 構造体に対して安全な再ネゴシエーションを強制します。これは推奨されません。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- TLSX_Find
- TLSX_UseSecureRenegotiation

Return:

- SSL_SUCCESS 安全な再ネゴシエーションの設定に成功しました。
- BAD_FUNC_ARG ssl が NULL の場合にエラーを返します。
- MEMORY_E 安全な再ネゴシエーション用のメモリを割り当てできない場合にエラーを返します。

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // 何らかの wolfSSL メソッド
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSecureRenegotiation(ssl) != SSL_SUCCESS)
{
    // 安全な再ネゴシエーションの設定エラー
}
```

C.52.2.374 function wolfSSL_Rehandshake

```
int wolfSSL_Rehandshake(
    WOLFSSL * ssl
)
```

この関数は安全な再ネゴシエーションハンドシェイクを実行します。wolfSSL はこの機能を推奨していないため、これはユーザが強制するものです。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_negotiate`
- `wc_InitSha512`
- `wc_InitSha384`
- `wc_InitSha256`
- `wc_InitSha`
- `wc_InitMd5`

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合、またはサブルーチンで受け入れられない引数が渡された場合に返されます。
- SECURE_RENEGOTIATION_E ハンドシェイクの再ネゴシエーションでエラーがあった場合に返されます。
- SSL_FATAL_ERROR サーバまたはクライアント設定にエラーがあり、再ネゴシエーションを完了できなかった場合に返されます。wolfSSL_negotiate() を参照してください。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_Rehandshake(ssl) != SSL_SUCCESS){
    // エラーが発生し、再ハンドシェイクは成功しませんでした。
}
```

C.52.2.375 function wolfSSL_UseSessionTicket

```
int wolfSSL_UseSessionTicket(  
    WOLFSSL * ssl  
)
```

提供された WOLFSSL 構造体にセッションチケットを使用するよう強制します。定数 HAVE_SESSION_TICKET が定義されており、定数 NO_WOLFSSL_CLIENT が定義されていない必要があります。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS セッションチケットの使用設定に成功しました。
- BAD_FUNC_ARG ssl が NULL の場合に返されます。
- MEMORY_E セッションチケット設定のためのメモリ割り当てエラー。

Example

```
wolfSSL_Init();  
WOLFSSL_CTX* ctx;  
WOLFSSL* ssl;  
WOLFSSL_METHOD method = // 何らかの wolfSSL メソッド  
ctx = wolfSSL_CTX_new(method);  
ssl = wolfSSL_new(ctx);  
  
if(wolfSSL_UseSessionTicket(ssl) != SSL_SUCCESS)  
{  
    // セッションチケットの設定エラー  
}
```

C.52.2.376 function wolfSSL_CTX_UseSessionTicket

```
int wolfSSL_CTX_UseSessionTicket(  
    WOLFSSL_CTX * ctx  
)
```

この関数は、wolfSSL コンテキストにセッションチケットを使用するよう設定します。

Parameters:

- **ctx** 使用する WOLFSSL_CTX 構造体。

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS 関数が正常に実行されました。
- BAD_FUNC_ARG ctx が NULL の場合に返されます。
- MEMORY_E 内部関数でメモリ割り当てエラー。

Example

```
wolfSSL_Init();  
WOLFSSL_CTX* ctx;  
WOLFSSL_METHOD method = // 何らかの wolfSSL メソッド ;  
ctx = wolfSSL_CTX_new(method);  
  
if(wolfSSL_CTX_UseSessionTicket(ctx) != SSL_SUCCESS)
```

```
{
    // セッションチケットの設定エラー
}
```

C.52.2.377 function wolfSSL_get_SessionTicket

```
int wolfSSL_get_SessionTicket(
    WOLFSSL * ssl,
    unsigned char * buf,
    word32 * bufSz
)
```

この関数は、Session 構造体の ticket メンバをバッファにコピーします。buf が NULL で bufSz が非 NULL の場合、bufSz はチケット長に設定されます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** メモリバッファを表す byte ポインタ。
- **bufSz** バッファサイズを表す word32 ポインタ。

See:

- [wolfSSL_UseSessionTicket](#)
- [wolfSSL_set_SessionTicket](#)

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- BAD_FUNC_ARG ssl または bufSz が NULL の場合、または bufSz が非 NULL で buf が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buf;
word32 bufSz; // buf サイズで初期化
...
if(wolfSSL_get_SessionTicket(ssl, buf, bufSz) <= 0){
    // バッファに何も書き込まれませんでした。
} else {
    // バッファは ssl->session->ticket の内容を保持しています。
}
```

C.52.2.378 function wolfSSL_set_SessionTicket

```
int wolfSSL_set_SessionTicket(
    WOLFSSL * ssl,
    const unsigned char * buf,
    word32 bufSz
)
```

この関数は、WOLFSSL 構造体内の WOLFSSL_SESSION 構造体の ticket メンバを設定します。関数に渡されたバッファはメモリにコピーされます。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** セッション構造体の ticket メンバに読み込まれる byte ポインタ。

- **bufSz** バッファのサイズを表す word32 型。

See: [wolfSSL_set_SessionTicket_cb](#)

Return:

- SSL_SUCCESS 関数の実行が成功した場合に返されます。関数はエラーなく返されました。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合に返されます。bufSz 引数がゼロでないのに buf 引数が NULL の場合にも返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buffer; // 読み込むファイル
word32 bufSz;
...
if(wolfSSL_KeepArrays(ssl, buffer, bufSz) != SSL_SUCCESS){
    // バッファをメモリに読み込む際にエラーが発生しました。
}
```

C.52.2.379 function wolfSSL_set_SessionTicket_cb

```
int wolfSSL_set_SessionTicket_cb(
    WOLFSSL * ssl,
    CallbackSessionTicket cb,
    void * ctx
)
```

この関数は、セッションチケットコールバックを設定します。CallbackSessionTicket 型は、次のシグネチャを持つ関数ポインタです: int (CallbackSessionTicket)(WOLFSSL, const unsigned char, int, void)

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** CallbackSessionTicket 型への関数ポインタ。
- **ctx** WOLFSSL 構造体の session_ticket_ctx メンバへの void ポインタ。

See:

- [wolfSSL_get_SessionTicket](#)
- CallbackSessionTicket
- sessionTicketCB

Return:

- SSL_SUCCESS 関数がエラーなく実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合に返されます。

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.52.2.380 function wolfSSL_send_SessionTicket

```
int wolfSSL_send_SessionTicket(
    WOLFSSL * ssl
)
```

この関数は、TLS v1.3 ハンドシェイクが確立された後、クライアントにセッションチケットを送信します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_get_SessionTicket`
- `CallbackSessionTicket`
- `sessionTicketCB`

Return:

- WOLFSSL_SUCCESS 新しいセッションチケットが送信された場合に返されます。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL、または TLS v1.3 を使用していない場合に返されます。
- SIDE_ERROR サーバでない場合に返されます。
- NOT_READY_ERROR ハンドシェイクが完了していない場合に返されます。
- WOLFSSL_FATAL_ERROR メッセージの作成または送信に失敗した場合に返されます。

Example

```
int ret;
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
ret = wolfSSL_send_SessionTicket(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // 新しいセッションチケットが送信されませんでした。
}
```

C.52.2.381 function wolfSSL_CTX_set_TicketEncCb

```
int wolfSSL_CTX_set_TicketEncCb(
    WOLFSSL_CTX * ctx,
    SessionTicketEncCb cb
)
```

この関数は、RFC 5077 で規定されているセッションチケットをサポートするサーバのセッションチケット鍵暗号化コールバック関数を設定します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX オブジェクトへのポインタ。
- **cb** セッションチケットを暗号化/復号するユーザコールバック関数
- **ssl(Callback)** `wolfSSL_new()`で作成された WOLFSSL オブジェクトへのポインタ
- **key_name(Callback)** このチケットコンテキスト用の一意の鍵名、ランダムに生成されるべきです
- **iv(Callback)** このチケット用の一意の IV、最大 128 ビット、ランダムに生成されるべきです
- **mac(Callback)** このチケット用の最大 256 ビットの mac
- **enc(Callback)** この暗号化パラメータが true の場合、ユーザは key_name、iv、mac を入力し、長さ inLen のチケットをインプレースで暗号化し、結果の出力長を outLen に設定する必要があります。WOLFSSL_TICKET_RET_OK を返すことで、wolfSSL に暗号化が成功したことを伝えます。この暗号化パラメータが false の場合、ユーザは key_name、iv、mac を使用して長さ inLen のチケットのインプレース復号を実行する必要があります。結果の復号長は outLen に設定する必要があります。WOLFSSL_TICKET_RET_OK を返すことで、wolfSSL に復号されたチケットを使用して続行するよう伝えます。WOLFSSL_TICKET_RET_CREATE を返すことで、wolfSSL に復号されたチケットを使用する

が、クライアントに送信する新しいチケットも生成するよう伝えます。これは最近鍵をロールした場合に完全なハンドシェイクを強制したくない場合に役立ちます。WOLFSSL_TICKET_RET_REJECT を返すことで、wolfSSL にこのチケットを拒否し、完全なハンドシェイクを実行し、通常のセッション再開用の新しい標準セッション ID を作成するよう伝えます。WOLFSSL_TICKET_RET_FATAL を返すことで、wolfSSL に致命的エラーで接続試行を終了するよう伝えます。

- **ticket(Callback)** 暗号化されたチケットの入出力バッファ。enc パラメータを参照してください。
- **inLen(Callback)** ticket パラメータの入力長。
- **outLen(Callback)** ticket パラメータの結果出力長。コールバックに入るとき、outLen は ticket バッファで利用可能な最大サイズを示します。
- **userCtx(Callback)** wolfSSL_CTX_set_TicketEncCtx() で設定されたユーザコンテキスト

See:

- wolfSSL_CTX_set_TicketHint
- wolfSSL_CTX_set_TicketEncCtx

Return:

- SSL_SUCCESS セッションの設定に成功した場合に返されます。
- BAD_FUNC_ARG 失敗時に返されます。これは、関数に無効な引数が渡された場合に発生します。

Example

wolfssl/test.h の myTicketEncCb() を参照してください。
サンプルサーバとサンプル echoserver で使用されています。

C.52.2.382 function wolfSSL_CTX_set_TicketHint

```
int wolfSSL_CTX_set_TicketHint(
    WOLFSSL_CTX * ctx,
    int hint
)
```

この関数は、クライアントに中継されるセッションチケットヒントを設定します。サーバ側での使用。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された WOLFSSL_CTX オブジェクトへのポインタ。
- **hint** チケットが有効である可能性がある秒数。クライアントへのヒント。

See: wolfSSL_CTX_set_TicketEncCb

Return:

- SSL_SUCCESS セッションの設定に成功した場合に返されます。
- BAD_FUNC_ARG 失敗時に返されます。これは、関数に無効な引数が渡された場合に発生します。

Example

none

C.52.2.383 function wolfSSL_CTX_set_TicketEncCtx

```
int wolfSSL_CTX_set_TicketEncCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

この関数は、コールバック用のセッションチケット暗号化ユーザコンテキストを設定します。サーバ側で使用します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX オブジェクトへのポインタ。
- **userCtx** コールバック用のユーザコンテキスト。

See: `wolfSSL_CTX_set_TicketEncCb`

Return:

- SSL_SUCCESS セッションの設定に成功した場合に返されます。
- BAD_FUNC_ARG 失敗時に返されます。これは関数に無効な引数が渡されたことが原因です。

Example

none

C.52.2.384 function `wolfSSL_CTX_get_TicketEncCtx`

```
void * wolfSSL_CTX_get_TicketEncCtx(
    WOLFSSL_CTX * ctx
)
```

この関数は、コールバック用のセッションチケット暗号化ユーザコンテキストを取得します。サーバ側で使用します。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX オブジェクトへのポインタ。

See: `wolfSSL_CTX_set_TicketEncCtx`

Return:

- userCtx セッションの取得に成功した場合に返されます。
- NULL 失敗時に返されます。これは関数に無効な引数が渡された場合、またはユーザコンテキストが設定されていない場合に発生します。

Example

none

C.52.2.385 function `wolfSSL_SetHsDoneCb`

```
int wolfSSL_SetHsDoneCb(
    WOLFSSL * ssl,
    HandShakeDoneCb cb,
    void * user_ctx
)
```

この関数は、ハンドシェイク完了コールバックを設定します。WOLFSSL 構造体の hsDoneCb と hsDoneCtx メンバがこの関数で設定されます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** HandShakeDoneCb 型の関数ポインタで、次の形式のシグネチャを持ちます: `int (HandShakeDoneCb)(WOLFSSL, void*)`;
- **user_ctx** ユーザ登録コンテキストへの void ポインタ。

See: `HandShakeDoneCb`

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。WOLFSSL 構造体の hsDoneCb と hsDoneCtx メンバが設定されます。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
int myHsDoneCb(WOLFSSL* ssl, void* user_ctx){
    // コールバック関数
}
...
wolfSSL_SetHsDoneCb(ssl, myHsDoneCb, NULL);
```

C.52.2.386 function wolfSSL_PrintSessionStats

```
int wolfSSL_PrintSessionStats(
    void
)
```

この関数は、セッションの統計情報を出力します。

Parameters:

- **none** パラメータなし。

See: [wolfSSL_get_session_stats](#)

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで返された場合に返されます。セッション統計が正常に取得され、出力されました。
- BAD_FUNC_ARG サブルーチン wolfSSL_get_session_stats() に許容できない引数が渡された場合に返されます。
- BAD_MUTEX_E サブルーチンで mutex エラーが発生した場合に返されます。

Example

```
// 統計を取得するためのセッションオブジェクトが必要です
if(wolfSSL_PrintSessionStats(void) != SSL_SUCCESS){
    // セッション統計を出力しませんでした
}
```

C.52.2.387 function wolfSSL_get_session_stats

```
int wolfSSL_get_session_stats(
    unsigned int * active,
    unsigned int * total,
    unsigned int * peak,
    unsigned int * maxSessions
)
```

この関数は、セッションの統計情報を取得します。

Parameters:

- **active** 総現在セッション数を表す word32 ポインタ。
- **total** 総セッション数を表す word32 ポインタ。
- **peak** ピークセッション数を表す word32 ポインタ。
- **maxSessions** 最大セッション数を表す word32 ポインタ。

See: [wolfSSL_PrintSessionStats](#)

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで返された場合に返されます。セッション統計が正常に取得され、出力されました。
- BAD_FUNC_ARG サブルーチン wolfSSL_get_session_stats() に許容できない引数が渡された場合に返されます。
- BAD_MUTEX_E サブルーチンで mutex エラーが発生した場合に返されます。

Example

```
int wolfSSL_PrintSessionStats(void){
...
ret = wolfSSL_get_session_stats(&totalSessionsNow,
    &totalSessionsSeen, &peak, &maxSessions);
...
return ret;
```

C.52.2.388 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
)
```

この関数は、cr と sr の値をコピーし、wc_PRF(疑似乱数関数) に渡し、その値を返します。

Parameters:

- **ms** Arrays 構造体に保持されているマスターシークレット。
- **msLen** マスターシークレットの長さ。
- **pms** Arrays 構造体に保持されているプリマスターシークレット。
- **pmsLen** プリマスターシークレットの長さ。
- **cr** クライアントランダム。
- **sr** サーバランダム。
- **tls1_2** バージョンが少なくとも TLS バージョン 1.2 であることを示します。
- **hash_type** ハッシュタイプを示します。

See:

- wc_PRF
- MakeTlsMasterSecret

Return:

- 0 成功時。
- BUFFER_E バッファのサイズでエラーが発生する場合に返されます。
- MEMORY_E サブルーチンが動的メモリの割り当てに失敗した場合に返されます。

Example

```
WOLFSSL* ssl;
```

MakeTlsMasterSecret で呼び出され、以下のように必要な情報を取得します：

```
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.52.2.389 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
)

```

TLS 鍵を導出するための外部向けラッパー。

Parameters:

- **key_data** DeriveTlsKeys で割り当てられ、最終ハッシュを保持するために wc_PRF に渡されるバイトポインタ。
- **keyLen** DeriveTlsKeys で WOLFSSL 構造体の specs メンバから導出される word32 型。
- **ms** WOLFSSL 構造体内の arrays 構造体に保持されているマスターシークレットを保持する定数ポインタ型。
- **msLen** 列挙定義 SECRET_LEN でマスターシークレットの長さを保持する word32 型。
- **sr** WOLFSSL 構造体内の arrays 構造体の serverRandom メンバへの定数バイトポインタ。
- **cr** WOLFSSL 構造体内の arrays 構造体の clientRandom メンバへの定数バイトポインタ。
- **tls1_2** IsAtLeastTLSv1_2() から返される整数型。
- **hash_type** WOLFSSL 構造体に保持されている整数型。

See:

- wc_PRF
- DeriveTlsKeys
- IsAtLeastTLSv1_2

Return:

- 0 成功時に返されます。
- BUFFER_E labLen と seedLen の合計 (合計サイズを計算) が最大サイズを超えた場合に返されます。
- MEMORY_E メモリの割り当てに失敗した場合に返されます。

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.52.2.390 function wolfSSL_connect_ex

```
int wolfSSL_connect_ex(  
    WOLFSSL * ssl,  
    HandShakeCallback hsCb,  
    TimeoutCallback toCb,  
    WOLFSSL_TIMEVAL timeout  
)
```

wolfSSL_connect_ex() は、HandShake コールバックを設定できる拡張です。これは、デバッガが利用できず、スニффイングが実用的でない場合に、組み込みシステムのデバッグサポートに役立ちます。HandShake コールバックは、ハンドシェイクエラーが発生したかどうかに関わらず呼び出されます。SSL パケットの最大数が既知であるため、動的メモリは使用されません。パケット名は packetNames[] を通じてアクセスできます。接続拡張はまた、タイムアウト値と共に Timeout コールバックを設定できます。これは、ユーザが TCP スタックのタイムアウトを待ちたくない場合に便利です。この拡張は、いずれか、両方、またはどちらのコールバックもなしで呼び出すことができます。

Parameters:

- **none** パラメータなし。

See: [wolfSSL_accept_ex](#)

Return:

- SSL_SUCCESS 成功時。
- GETTIME_ERROR gettimeofday() がエラーに遭遇した場合に返されます。
- SETTIMER_ERROR setitimer() がエラーに遭遇した場合に返されます。
- SIGACT_ERROR sigaction() がエラーに遭遇した場合に返されます。
- SSL_FATAL_ERROR 基礎となる SSL_connect() 呼び出しがエラーに遭遇した場合に返されます。

Example

none

C.52.2.391 function wolfSSL_accept_ex

```
int wolfSSL_accept_ex(  
    WOLFSSL * ssl,  
    HandShakeCallback hsCb,  
    TimeoutCallback toCb,  
    WOLFSSL_TIMEVAL timeout  
)
```

wolfSSL_accept_ex() は、HandShake コールバックを設定できる拡張です。これは、デバッガが利用できず、スニффイングが実用的でない場合に、組み込みシステムのデバッグサポートに役立ちます。HandShake コールバックは、ハンドシェイクエラーが発生したかどうかに関わらず呼び出されます。SSL パケットの最大数が既知であるため、動的メモリは使用されません。パケット名は packetNames[] を通じてアクセスできます。接続拡張はまた、タイムアウト値と共に Timeout コールバックを設定できます。これは、ユーザが TCP スタックのタイムアウトを待ちたくない場合に便利です。この拡張は、いずれか、両方、またはどちらのコールバックもなしで呼び出すことができます。

Parameters:

- **none** パラメータなし。

See: [wolfSSL_connect_ex](#)

Return:

- SSL_SUCCESS 成功時。

- GETTIME_ERROR gettimeofday() がエラーに遭遇した場合に返されます。
- SETTIMER_ERROR setitimer() がエラーに遭遇した場合に返されます。
- SIGACT_ERROR sigaction() がエラーに遭遇した場合に返されます。
- SSL_FATAL_ERROR 基礎となる SSL_accept() 呼び出しがエラーに遭遇した場合に返されます。

Example

none

C.52.2.392 function wolfSSL_BIO_set_fp

```
long wolfSSL_BIO_set_fp(  
    WOLFSSL_BIO * bio,  
    XFILE fp,  
    int c  
)
```

これは、BIO の内部ファイルポインタを設定するために使用されます。

Parameters:

- **bio** ペアを設定する WOLFSSL_BIO 構造体。
- **fp** bio に設定するファイルポインタ。
- **c** ファイルクローズ動作フラグ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- **wolfSSL_BIO_get_fp**
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS ファイルポインタの設定に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。

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);  
// ret 値を確認
```

C.52.2.393 function wolfSSL_BIO_get_fp

```
long wolfSSL_BIO_get_fp(  
    WOLFSSL_BIO * bio,  
    XFILE * fp  
)
```

これは、BIO の内部ファイルポインタを取得するために使用されます。

Parameters:

- **bio** ペアを設定する WOLFSSL_BIO 構造体。
- **fp** bio に設定するファイルポインタ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- **wolfSSL_BIO_set_fp**
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS ファイルポインタの取得に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret = wolfSSL_BIO_get_fp(bio, &fp);
// ret 値を確認
```

C.52.2.394 function wolfSSL_check_private_key

```
int wolfSSL_check_private_key(
    const WOLFSSL * ssl
)
```

この関数は、秘密鍵が使用されている証明書と一致していることを確認します。

Parameters:

- **ssl** 確認する WOLFSSL 構造体。

See:

- **wolfSSL_new**
- **wolfSSL_free**

Return:

- SSL_SUCCESS 一致に成功した場合。
- SSL_FAILURE エラーケースが発生した場合。
- <0 SSL_FAILURE 以外のすべてのエラーケースは負の値です。

Example

```
WOLFSSL* ssl;
int ret;
// ssl を作成してセットアップ
ret = wolfSSL_check_private_key(ssl);
// ret 値を確認
```

C.52.2.395 function wolfSSL_X509_get_ext_by_NID

```
int wolfSSL_X509_get_ext_by_NID(
    const WOLFSSL_X509 * x,
    int nid,
    int lastpos
)
```

この関数は、渡された NID 値に一致する拡張インデックスを検索して返します。

Parameters:

- **x509** 拡張を検索するために解析する証明書。
- **nid** 見つける拡張 OID。
- **lastpos** lastPos 以降の拡張から検索を開始します。最初は-1 に設定します。

Return:

- ≥ 0 成功時、拡張インデックスが返されます。
- -1 拡張が見つからない場合、またはエラーが発生した場合。

Example

```
const WOLFSSL_X509* x509;
int lastPos = -1;
int idx;

idx = wolfSSL_X509_get_ext_by_NID(x509, NID_basic_constraints, lastPos);
```

C.52.2.396 function wolfSSL_X509_get_ext_d2i

```
void * wolfSSL_X509_get_ext_d2i(
    const WOLFSSL_X509 * x509,
    int nid,
    int * c,
    int * idx
)
```

この関数は渡された NID 値に一致する拡張を検索して返します。

Parameters:

- **x509** 拡張を解析する証明書。
- **nid** 検索する拡張 OID。
- **c** NULL でない場合、複数の拡張が見つかった場合は-2、見つからなかった場合は-1、見つかってクリティカルでない場合は 0、見つかってクリティカルな場合は 1 に設定されます。
- **idx** NULL の場合は最初に一致した拡張を返します。それ以外の場合、x509 に格納されていなければ idx から開始します。

See: wolfSSL_sk_ASN1_OBJECT_free

Return:

- pointer 成功した場合、STACK_OF(WOLFSSL_ASN1_OBJECT) ポインタが返されます。
- NULL 拡張が見つからない、またはエラーが発生した場合。

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);
// sk が NULL でないか確認してから使用。使用後は sk を解放する必要があります
```

C.52.2.397 function wolfSSL_X509_digest

```
int wolfSSL_X509_digest(
    const WOLFSSL_X509 * x509,
    const WOLFSSL_EVP_MD * digest,
    unsigned char * buf,
```



```
    unsigned int * len
)
```

この関数は DER 証明書のハッシュを返します。

Parameters:

- **x509** ハッシュを取得する証明書。
- **digest** 使用するハッシュアルゴリズム。
- **buf** ハッシュを保持するバッファ。
- **len** バッファの長さ。

See: none

Return:

- SSL_SUCCESS ハッシュの作成に成功した場合。
- SSL_FAILURE 不正な入力またはハッシュ失敗時に返されます。

Example

```
WOLFSSL_X509* x509;
unsigned char buffer[64];
unsigned int bufferSz;
int ret;

ret = wolfSSL_X509_digest(x509, wolfSSL_EVP_sha256(), buffer, &bufferSz);
// ret 値を確認
```

C.52.2.398 function wolfSSL_use_certificate

```
int wolfSSL_use_certificate(
    WOLFSSL * ssl,
    WOLFSSL_X509 * x509
)
```

ハンドシェイク中に使用する WOLFSSL 構造体の証明書を設定するために使用されます。

Parameters:

- **ssl** 証明書を設定する WOLFSSL 構造体。
- **x509** 使用する証明書。

See:

- **wolfSSL_new**
- **wolfSSL_free**

Return:

- SSL_SUCCESS 引数の設定に成功した場合。
- SSL_FAILURE NULL 引数が渡された場合。

Example

```
WOLFSSL* ssl;
WOLFSSL_X509* x509
int ret;
// ssl オブジェクトと x509 を作成
ret = wolfSSL_use_certificate(ssl, x509);
// ret 値を確認
```

C.52.2.399 function wolfSSL_use_certificate_ASN1

```
int wolfSSL_use_certificate_ASN1(
    WOLFSSL * ssl,
    const unsigned char * der,
    int derSz
)
```

ハンドシェイク中に使用する WOLFSSL 構造体の証明書を設定するために使用されます。DER 形式のバッファが必要です。

Parameters:

- **ssl** 証明書を設定する WOLFSSL 構造体。
- **der** 使用する DER 証明書。
- **derSz** 渡された DER バッファのサイズ。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- SSL_SUCCESS 引数の設定に成功した場合。
- SSL_FAILURE NULL 引数が渡された場合。

Example

```
WOLFSSL* ssl;
unsigned char* der;
int derSz;
int ret;
// ssl オブジェクトを作成して DER 変数を設定
ret = wolfSSL_use_certificate_ASN1(ssl, der, derSz);
// ret 値を確認
```

C.52.2.400 function wolfSSL_use_PrivateKey

```
int wolfSSL_use_PrivateKey(
    WOLFSSL * ssl,
    WOLFSSL_EVP_PKEY * pkey
)
```

WOLFSSL 構造体の秘密鍵を設定するために使用されます。

Parameters:

- **ssl** 引数を設定する WOLFSSL 構造体。
- **pkey** 使用する秘密鍵。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- SSL_SUCCESS 引数の設定に成功した場合。
- SSL_FAILURE NULL の ssl が渡された場合。すべてのエラーケースは負の値になります。

Example

```
WOLFSSL* ssl;
WOLFSSL_EVP_PKEY* pkey;
int ret;
// ssl オブジェクトを作成して秘密鍵をセットアップ
ret = wolfSSL_use_PrivateKey(ssl, pkey);
// ret 値を確認
```

C.52.2.401 function wolfSSL_use_PrivateKey_ASN1

```
int wolfSSL_use_PrivateKey_ASN1(
    int pri,
    WOLFSSL * ssl,
    const unsigned char * der,
    long derSz
)
```

WOLFSSL 構造体の秘密鍵を設定するために使用されます。DER 形式の鍵バッファが必要です。

Parameters:

- **pri** 秘密鍵のタイプ。
- **ssl** 引数を設定する WOLFSSL 構造体。
- **der** DER 鍵を保持するバッファ。
- **derSz** der バッファのサイズ。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)
- [wolfSSL_use_PrivateKey](#)

Return:

- SSL_SUCCESS 秘密鍵の解析と設定に成功した場合。
- SSL_FAILURE NULL の ssl が渡された場合。すべてのエラーケースは負の値になります。

Example

```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// ssl オブジェクトを作成して秘密鍵をセットアップ
ret = wolfSSL_use_PrivateKey_ASN1(1, ssl, pkey, pkeySz);
// ret 値を確認
```

C.52.2.402 function wolfSSL_use_RSAPrivateKey_ASN1

```
int wolfSSL_use_RSAPrivateKey_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

WOLFSSL 構造体の秘密鍵を設定するために使用されます。DER 形式の RSA 鍵バッファが必要です。

Parameters:

- **ssl** 引数を設定する WOLFSSL 構造体。
- **der** DER 鍵を保持するバッファ。

- **derSz** der バッファのサイズ。

See:

- `wolfSSL_new`
- `wolfSSL_free`
- `wolfSSL_use_PrivateKey`

Return:

- `SSL_SUCCESS` 秘密鍵の解析と設定に成功した場合。
- `SSL_FAILURE` `NULL` の `ssl` が渡された場合。すべてのエラーケースは負の値になります。

Example

```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// ssl オブジェクトを作成して RSA 秘密鍵をセットアップ
ret = wolfSSL_use_RSAPrivateKey_ASN1(ssl, pkey, pkeySz);
// ret 値を確認
```

C.52.2.403 function `wolfSSL_DSA_dup_DH`

```
WOLFSSL_DH * wolfSSL_DSA_dup_DH(
    const WOLFSSL_DSA * r
)
```

この関数は `dsa` のパラメータを新しく作成された `WOLFSSL_DH` 構造体に複製します。

Parameters:

- **dsa** 複製する `WOLFSSL_DSA` 構造体。

See: none

Return:

- `WOLFSSL_DH` 複製に成功した場合、`WOLFSSL_DH` 構造体を返します。
- `NULL` 失敗時。

Example

```
WOLFSSL_DH* dh;
WOLFSSL_DSA* dsa;
// dsa をセットアップ
dh = wolfSSL_DSA_dup_DH(dsa);

// dh が null でないか確認
```

C.52.2.404 function `wolfSSL_SESSION_get_master_key`

```
int wolfSSL_SESSION_get_master_key(
    const WOLFSSL_SESSION * ses,
    unsigned char * out,
    int outSz
)
```

ハンドシェイク完了後にマスターキーを取得するために使用されます。

Parameters:

- **ses** マスターシークレットバッファを取得する WOLFSSL_SESSION 構造体。
- **out** データを保持するバッファ。
- **outSz** 渡された out バッファのサイズ (0 の場合、関数は必要な最大バッファサイズを返します)。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- > 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムデータバッファがない、またはエラー状態の場合は 0 を返します。
- max 渡された outSz が 0 の場合、必要な最大バッファサイズが返されます。

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// ハンドシェイクを完了してセッション構造体を取得
bufferSz = wolfSSL_SESSION_get_master_secret(ses, NULL, 0);
buffer = malloc(bufferSz);
ret = wolfSSL_SESSION_get_master_secret(ses, buffer, bufferSz);
// ret 値を確認
```

C.52.2.405 function wolfSSL_SESSION_get_master_key_length

```
int wolfSSL_SESSION_get_master_key_length(
    const WOLFSSL_SESSION * ses
)
```

マスターシークレットキーの長さを取得するために使用されます。

Parameters:

- **ses** マスターシークレットバッファを取得する WOLFSSL_SESSION 構造体。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return: size マスターシークレットキーのサイズを返します。

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// ハンドシェイクを完了してセッション構造体を取得
bufferSz = wolfSSL_SESSION_get_master_secret_length(ses);
buffer = malloc(bufferSz);
// ret 値を確認
```

C.52.2.406 function wolfSSL_CTX_set_cert_store

```
void wolfSSL_CTX_set_cert_store(
    WOLFSSL_CTX * ctx,
```

```
    WOLFSSL_X509_STORE * str  
)
```

ctx の WOLFSSL_X509_STORE 構造体の setter 関数です。

Parameters:

- **ctx** 証明書ストアポインタを設定する WOLFSSL_CTX 構造体へのポインタ。
- **str** ctx に設定する WOLFSSL_X509_STORE へのポインタ。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)

Return: none 戻り値なし。

Example

```
WOLFSSL_CTX ctx;  
WOLFSSL_X509_STORE* st;  
// ctx と st をセットアップ  
st = wolfSSL_CTX_set_cert_store(ctx, st);  
// st を使用
```

C.52.2.407 function wolfSSL_d2i_X509_bio

```
WOLFSSL_X509 * wolfSSL_d2i_X509_bio(  
    WOLFSSL_BIO * bio,  
    WOLFSSL_X509 ** x509  
)
```

この関数は bio から DER バッファを取得し、それを WOLFSSL_X509 構造体に変換します。

Parameters:

- **bio** DER 証明書バッファを持つ WOLFSSL_BIO 構造体へのポインタ。
- **x509** 作成された新しい WOLFSSL_X509 構造体に設定されるポインタ。

See: none

Return:

- pointer 成功時に WOLFSSL_X509 構造体ポインタを返します。
- Null 失敗時に NULL を返します。

Example

```
WOLFSSL_BIO* bio;  
WOLFSSL_X509* x509;  
// DER を bio にロード  
x509 = wolfSSL_d2i_X509_bio(bio, NULL);  
// または  
wolfSSL_d2i_X509_bio(bio, &x509);  
// 返された x509 を使用 (NULL をチェック)
```

C.52.2.408 function wolfSSL_CTX_get_cert_store

```
WOLFSSL_X509_STORE * wolfSSL_CTX_get_cert_store(  
    WOLFSSL_CTX * ctx  
)
```

ctx の WOLFSSL_X509_STORE 構造体の getter 関数です。

Parameters:

- **ctx** 証明書ストアポインタを取得する WOLFSSL_CTX 構造体へのポインタ。

See:

- `wolfSSL_CTX_new`
- `wolfSSL_CTX_free`
- `wolfSSL_CTX_set_cert_store`

Return:

- WOLFSSL_X509_STORE* ポインタの取得に成功した場合。
- NULL NULL 引数が渡された場合に返されます。

Example

```
WOLFSSL_CTX ctx;  
WOLFSSL_X509_STORE* st;  
// ctx をセットアップ  
st = wolfSSL_CTX_get_cert_store(ctx);  
// st を使用
```

C.52.2.409 function wolfSSL_BIO_ctrl_pending

```
size_t wolfSSL_BIO_ctrl_pending(  
    WOLFSSL_BIO * b  
)
```

読み取り保留中のバイト数を取得します。BIO タイプが BIO_BIO の場合、ペアから読み取るバイト数です。BIO が SSL オブジェクトを含む場合、SSL オブジェクトからの保留中のデータです (wolfSSL_pending(ssl))。BIO_MEMORY タイプの場合、メモリバッファのサイズを返します。

Parameters:

- **bio** すでに作成されている WOLFSSL_BIO 構造体へのポインタ。

See:

- `wolfSSL_BIO_make_bio_pair`
- `wolfSSL_BIO_new`

Return: > =0 保留中のバイト数。

Example

```
WOLFSSL_BIO* bio;    int pending;  
bio = wolfSSL_BIO_new();  
...  
pending = wolfSSL_BIO_ctrl_pending(bio);
```

C.52.2.410 function wolfSSL_get_server_random

```
size_t wolfSSL_get_server_random(  
    const WOLFSSL * ssl,  
    unsigned char * out,  
    size_t outlen  
)
```

ハンドシェイク中にサーバーから送信されたランダムデータを取得するために使用されます。

Parameters:

- **ssl** クライアントのランダムデータバッファを取得する WOLFSSL 構造体。
- **out** ランダムデータを保持するバッファ。
- **outSz** 渡された out バッファのサイズ (0 の場合、関数は必要な最大バッファサイズを返します)。

See:

- `wolfSSL_new`
- `wolfSSL_free`

Return:

- > 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムデータバッファがない場合、またはエラー状態の場合、0 を返します。
- max 渡された outSz が 0 の場合、必要な最大バッファサイズを返します。

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);
// ret 値を確認
```

C.52.2.411 function wolfSSL_get_client_random

```
size_t wolfSSL_get_client_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outSz
)
```

ハンドシェイク中にクライアントから送信されたランダムデータを取得するために使用されます。

Parameters:

- **ssl** クライアントのランダムデータバッファを取得する WOLFSSL 構造体。
- **out** ランダムデータを保持するバッファ。
- **outSz** 渡された out バッファのサイズ (0 の場合、関数は必要な最大バッファサイズを返します)。

See:

- `wolfSSL_new`
- `wolfSSL_free`

Return:

- > 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムデータバッファがない場合、またはエラー状態の場合、0 を返します。
- max 渡された outSz が 0 の場合、必要な最大バッファサイズを返します。

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);
// ret 値を確認
```

C.52.2.412 function wolfSSL_CTX_get_default_passwd_cb

```
wc_pem_password_cb * wolfSSL_CTX_get_default_passwd_cb(
    WOLFSSL_CTX * ctx
)
```

ctx 内に設定されたパスワードコールバックの getter 関数です。

Parameters:

- **ctx** コールバックを取得する WOLFSSL_CTX 構造体。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)

Return:

- func 成功時にはコールバック関数を返します。
- NULL ctx が NULL の場合、NULL を返します。

Example

```
WOLFSSL_CTX* ctx;
wc_pem_password_cb cb;
// ctx をセットアップ
cb = wolfSSL_CTX_get_default_passwd_cb(ctx);
//cb を使用
```

C.52.2.413 function wolfSSL_CTX_get_default_passwd_cb_userdata

```
void * wolfSSL_CTX_get_default_passwd_cb_userdata(
    WOLFSSL_CTX * ctx
)
```

ctx 内に設定されたパスワードコールバックユーザーデータの getter 関数です。

Parameters:

- **ctx** ユーザーデータを取得する WOLFSSL_CTX 構造体。

See:

- [wolfSSL_CTX_new](#)
- [wolfSSL_CTX_free](#)

Return:

- pointer 成功時にはユーザーデータポインタを返します。
- NULL ctx が NULL の場合、NULL を返します。

Example

```
WOLFSSL_CTX* ctx;
void* data;
// ctx をセットアップ
```

```
data = wolfSSL_CTX_get_default_passwd_cb(ctx);
//data を使用
```

C.52.2.414 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
)
```

この関数は wolfSSL_PEM_read_bio_X509 と同じように動作します。AUX は、信頼された/拒否されたユーースケースや人間が読みやすいフレンドリ名などの追加情報を含むことを意味します。

Parameters:

- **bp** PEM バッファを取得する WOLFSSL_BIO 構造体。
- **x** 関数の副作用によって WOLFSSL_X509 を設定する場合。
- **cb** パスワードコールバック。
- **u** NULL 終端のユーザーパスワード。

See: wolfSSL_PEM_read_bio_X509

Return:

- WOLFSSL_X509 PEM バッファの解析に成功した場合、WOLFSSL_X509 構造体が返されます。
- Null PEM バッファの解析に失敗した場合。

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// bio をセットアップ
X509 = wolfSSL_PEM_read_bio_X509_AUX(bio, NULL, NULL, NULL);
//x509 が null でないことを確認してから使用
```

C.52.2.415 function wolfSSL_CTX_set_tmp_dh

```
long wolfSSL_CTX_set_tmp_dh(
    WOLFSSL_CTX * ctx,
    WOLFSSL_DH * dh
)
```

WOLFSSL_CTX 構造体の dh メンバを Diffie-Hellman パラメータで初期化します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **dh** WOLFSSL_DH 構造体へのポインタ。

See: wolfSSL_BN_bn2bin

Return:

- SSL_SUCCESS 関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG ctx または dh 構造体が NULL の場合に返されます。
- SSL_FATAL_ERROR 構造体の値の設定にエラーがあった場合に返されます。
- MEMORY_E メモリの割り当てに失敗した場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL_DH* dh;
...
return wolfSSL_CTX_set_tmp_dh(ctx, dh);
```

C.52.2.416 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
)
```

この関数は、bio 内の PEM バッファから DSA パラメータを取得します。

Parameters:

- **bio** PEM メモリポインタを取得するための WOLFSSL_BIO 構造体へのポインタ。
- **x** 新しい WOLFSSL_DSA 構造体に設定されるポインタ。
- **cb** パスワードコールバック関数。
- **u** null 終端のパスワード文字列。

See: none

Return:

- WOLFSSL_DSA PEM バッファの解析に成功した場合、WOLFSSL_DSA 構造体を作成されて返されます。
- Null PEM バッファの解析に失敗した場合。

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_DSA* dsa;
// bio をセットアップ
dsa = wolfSSL_PEM_read_bio_DSAParams(bio, NULL, NULL, NULL);

// dsa が NULL でないことを確認してから dsa を使用
```

C.52.2.417 function wolfSSL_ERR_peek_last_error

```
unsigned long wolfSSL_ERR_peek_last_error(
    void
)
```

この関数は、WOLFSSL_ERROR から発生した最後のエラーの絶対値を返します。

Parameters:

- **none** パラメータはありません。

See: [wolfSSL_ERR_print_errors_fp](#)

Return: error 最後のエラーの絶対値を返します。

Example

```
unsigned long err;
...
err = wolfSSL_ERR_peek_last_error();
// err 値を検査
```

C.52.2.418 function WOLF_STACK_OF

```
WOLF_STACK_OF(  
    WOLFSSL_X509  
) const
```

この関数は、ピアの証明書チェーンを取得します。

Parameters:

- `ssl` `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_X509_get_issuer_name`
- `wolfSSL_X509_get_subject_name`
- `wolfSSL_X509_get_isCA`

Return:

- pointer ピアの Certificate スタックへのポインタを返します。
- NULL ピア証明書がない場合に返されます。

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);  
if(chain){  
    // ピア証明書チェーンへのポインタがあります  
}
```

C.52.2.419 function wolfSSL_CTX_clear_options

```
long wolfSSL_CTX_clear_options(  
    WOLFSSL_CTX * ctx,  
    long opt  
)
```

この関数は、WOLFSSL_CTX オブジェクトのオプションビットをリセットします。

Parameters:

- `ctx` SSL コンテキストへのポインタ。

See:

- `wolfSSL_CTX_new`
- `wolfSSL_new`
- `wolfSSL_free`

Return: option 新しいオプションビット。

Example

```
WOLFSSL_CTX* ctx = 0;  
...  
wolfSSL_CTX_clear_options(ctx, SSL_OP_NO_TLSv1);
```

C.52.2.420 function wolfSSL_set_jobject

```
int wolfSSL_set_jobject(  
    WOLFSSL * ssl,  
    void * objPtr  
)
```

この関数は、WOLFSSL 構造体の jobjectRef メンバを設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **objPtr** jobjectRef に設定される void ポインタ。

See: [wolfSSL_get_jobject](#)

Return:

- SSL_SUCCESS jobjectRef が objPtr に適切に設定された場合に返されます。
- SSL_FAILURE 関数が適切に実行されず、jobjectRef が設定されていない場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );  
WOLFSSL* ssl = WOLFSSL_new();  
void* objPtr = &obj;  
...  
if(wolfSSL_set_jobject(ssl, objPtr)){  
    // 成功ケース  
}
```

C.52.2.421 function wolfSSL_get_jobject

```
void * wolfSSL_get_jobject(  
    WOLFSSL * ssl  
)
```

この関数は、WOLFSSL 構造体の jobjectRef メンバを返します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See: [wolfSSL_set_jobject](#)

Return:

- value WOLFSSL 構造体が NULL でない場合、関数は jobjectRef 値を返します。
- NULL WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );  
WOLFSSL* ssl = wolfSSL(ctx);  
...  
void* jobject = wolfSSL_get_jobject(ssl);  
  
if(jobject != NULL){  
    // 成功ケース  
}
```

C.52.2.422 function wolfSSL_set_msg_callback

```
int wolfSSL_set_msg_callback(  
    WOLFSSL * ssl,  
    SSL_Msg_Cb cb  
)
```

この関数は、ssl 内にコールバックを設定します。コールバックはハンドシェイクメッセージを監視するためのものです。cb の NULL 値はコールバックをリセットします。

Parameters:

- **ssl** コールバック引数を設定する WOLFSSL 構造体。

See: [wolfSSL_set_msg_callback_arg](#)

Return:

- SSL_SUCCESS 成功時。
- SSL_FAILURE NULL の ssl が渡された場合。

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);  
// ret を確認
```

C.52.2.423 function wolfSSL_set_msg_callback_arg

```
int wolfSSL_set_msg_callback_arg(  
    WOLFSSL * ssl,  
    void * arg  
)
```

この関数は、ssl 内に関連するコールバックコンテキスト値を設定します。値はコールバック引数に渡されます。

Parameters:

- **ssl** コールバック引数を設定する WOLFSSL 構造体。

See: [wolfSSL_set_msg_callback](#)

Return: none 戻り値はありません。

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);  
// ret を確認  
wolfSSL_set_msg_callback(ssl, arg);
```

C.52.2.424 function wolfSSL_X509_get_next_altname

```
char * wolfSSL_X509_get_next_altname(
    WOLFSSL_X509 *
```

この関数は、ピア証明書から次の代替名 (もしあれば) を返します。

Parameters:

- **cert** wolfSSL_X509 構造体へのポインタ。

See:

- [wolfSSL_X509_get_issuer_name](#)
- [wolfSSL_X509_get_subject_name](#)

Return:

- NULL 次の代替名がない場合。
- cert->altNamesNext->name WOLFSSL_X509 構造体から返されます。altName リストからの文字列値が存在する場合に返されます。

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){
    //別の alt name はありません。
}
```

C.52.2.425 function wolfSSL_X509_get_notBefore

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notBefore(
    WOLFSSL_X509 *
```

この関数は、x509 が NULL かどうかを確認し、NULL でない場合は x509 構造体の notBefore メンバを返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

See: [wolfSSL_X509_get_notAfter](#)

Return:

- pointer x509 構造体の notBefore メンバへの ASN1_TIME を持つ構造体へのポインタ。
- NULL x509 構造体が NULL の場合、関数は NULL を返します。

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){
    //x509 オブジェクトが NULL でした。
}
```

C.52.2.426 function wolfSSL_connect

```
int wolfSSL_connect(
    WOLFSSL * ssl
)
```

この関数はクライアント側で呼び出され、サーバとの SSL/TLS ハンドシェイクを開始します。この関数が呼び出されるとき、基礎となる通信チャンネルはすでに設定されています。wolfSSL_connect() は、ブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、wolfSSL_connect() は、基礎となる I/O が wolfSSL_connect() がハンドシェイクを続行するために必要とするものを満たすことができないときに返されます。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE を返します。呼び出しプロセスは、基礎となる I/O の準備ができたときに wolfSSL_connect() の呼び出しを繰り返す必要があり、wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキングの場合、wolfSSL_connect() はハンドシェイクが完了するかエラーが発生するまで返されません。wolfSSL は、OpenSSL とは異なるアプローチで証明書検証を行います。クライアントのデフォルトポリシーはサーバを検証することです。つまり、サーバを検証するための CA を読み込まない場合、接続エラー、検証不可 (-155) が発生します。サーバの検証が失敗しても SSL_connect が成功し、セキュリティを低下させる OpenSSL の動作を模倣したい場合は、SSL_new() を呼び出す前に SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); を呼び出すことでこれを実現できます。ただし、推奨されません。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功した場合。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出してください。

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.52.2.427 function wolfSSL_send_hrr_cookie

```
int wolfSSL_send_hrr_cookie(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

この関数は、HelloRetryRequest メッセージに Cookie を含める必要があることを示すために、サーバ側で呼び出されます。また、プロトコル DTLS v1.3 を使用する場合、ハンドシェイクには常に Cookie 交換が含

まれることを示します。プロトコル DTLS v1.3 を使用する場合、Cookie 交換はデフォルトで有効になっていることに注意してください。Cookie は現在のトランスクリプトのハッシュを保持しており、別のサーバプロセスが応答の ClientHello を処理できるようにします。secret は、Cookie データの完全性チェックを生成する際に使用されます。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **secret** secret を保持するバッファへのポインタ。NULL を渡すと、新しいランダムな secret を生成することを示します。
- **secretSz** secret のサイズ (バイト単位)。0 を渡すと、デフォルトサイズを使用することを示します: `WC_SHA256_DIGEST_SIZE` (SHA-256 が利用できない場合は `WC_SHA_DIGEST_SIZE`)。

See:

- `wolfSSL_new`
- `wolfSSL_disable_hrr_cookie`

Return:

- `BAD_FUNC_ARG ssl` が NULL または TLS v1.3 を使用していない場合。
- `SIDE_ERROR` クライアントで呼び出された場合。
- `WOLFSSL_SUCCESS` 成功した場合。
- `MEMORY_ERROR` secret を保存するための動的メモリの割り当てに失敗した場合。
- `Another` 内部エラーの場合は他の負の値。

Example

```
int ret;
WOLFSSL* ssl;
char secret[32];
...
ret = wolfSSL__send_hrr_cookie(ssl, secret, sizeof(secret));
if (ret != WOLFSSL_SUCCESS) {
    // Cookie と secret の使用設定に失敗しました。
}
```

C.52.2.428 function `wolfSSL_disable_hrr_cookie`

```
int wolfSSL_disable_hrr_cookie(
    WOLFSSL * ssl
)
```

この関数は、HelloRetryRequest メッセージに Cookie を含めてはならないこと、およびプロトコル DTLS v1.3 を使用している場合、ハンドシェイクに Cookie 交換を含めないことを示すために、サーバ側で呼び出されます。プロトコル DTLS v1.3 を使用する際に Cookie 交換を行わないと、サーバが DoS/増幅攻撃に対して脆弱になる可能性があることに注意してください。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_send_hrr_cookie`

Return:

- `WOLFSSL_SUCCESS` 成功した場合。
- `BAD_FUNC_ARG ssl` が NULL または TLS v1.3 を使用していない場合。
- `SIDE_ERROR` クライアントで呼び出された場合。

C.52.2.429 function wolfSSL_CTX_no_ticket_TLSv13

```
int wolfSSL_CTX_no_ticket_TLSv13(
    WOLFSSL_CTX * ctx
)
```

この関数は、ハンドシェイクが完了した後、再開セッションチケットの送信を停止するために、サーバで呼び出されます。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX 構造体へのポインタ。

See: `wolfSSL_no_ticket_TLSv13`

Return:

- BAD_FUNC_ARG ctx が NULL または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。
- 0 成功した場合。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_ticket_TLSv13(ctx);
if (ret != 0) {
    // no ticket の設定に失敗しました。
}
```

C.52.2.430 function wolfSSL_no_ticket_TLSv13

```
int wolfSSL_no_ticket_TLSv13(
    WOLFSSL * ssl
)
```

この関数は、ハンドシェイクが完了した後、再開セッションチケットの送信を停止するために、サーバで呼び出されます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_CTX_no_ticket_TLSv13`

Return:

- BAD_FUNC_ARG ssl が NULL または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。
- 0 成功した場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_ticket_TLSv13(ssl);
if (ret != 0) {
    // no ticket の設定に失敗しました。
}
```

C.52.2.431 function wolfSSL_CTX_no_dhe_psk

```
int wolfSSL_CTX_no_dhe_psk(
    WOLFSSL_CTX * ctx
)
```

この関数は、事前共有鍵を認証に使用するハンドシェイクの際に、Diffie-Hellman(DH) スタイルの鍵交換を禁止するために、TLS v1.3 wolfSSL コンテキストで呼び出されます。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX 構造体へのポインタ。

See: `wolfSSL_no_dhe_psk`

Return:

- BAD_FUNC_ARG ctx が NULL または TLS v1.3 を使用していない場合。
- 0 成功した場合。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_dhe_psk(ctx);
if (ret != 0) {
    // PSK ハンドシェイクの DHE 無効化の設定に失敗しました。
}
```

C.52.2.432 function wolfSSL_no_dhe_psk

```
int wolfSSL_no_dhe_psk(
    WOLFSSL * ssl
)
```

この関数は、事前共有鍵を認証に使用するハンドシェイクの際に、Diffie-Hellman(DH) スタイルの鍵交換を禁止するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See: `wolfSSL_CTX_no_dhe_psk`

Return:

- BAD_FUNC_ARG ssl が NULL または TLS v1.3 を使用していない場合。
- 0 成功した場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_dhe_psk(ssl);
if (ret != 0) {
    // PSK ハンドシェイクの DHE 無効化の設定に失敗しました。
}
```

C.52.2.433 function wolfSSL_update_keys

```
int wolfSSL_update_keys(  
    WOLFSSL * ssl  
)
```

この関数は、鍵のロールオーバーを強制するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。KeyUpdate メッセージがピアに送信され、暗号化用の新しい鍵が計算されます。ピアは KeyUpdate メッセージを送り返し、その後新しい復号鍵が計算されます。この関数は、ハンドシェイクが完了した後にのみ呼び出すことができます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_write

Return:

- BAD_FUNC_ARG ssl が NULL または TLS v1.3 を使用していない場合。
- WANT_WRITE 書き込みの準備ができていない場合。
- WOLFSSL_SUCCESS 成功した場合。

Example

```
int ret;  
WOLFSSL * ssl;  
...  
ret = wolfSSL_update_keys(ssl);  
if (ret == WANT_WRITE) {  
    // I/O の準備ができれば再度呼び出す必要があります。  
}  
else if (ret != WOLFSSL_SUCCESS) {  
    // 鍵更新の送信に失敗しました。  
}
```

C.52.2.434 function wolfSSL_key_update_response

```
int wolfSSL_key_update_response(  
    WOLFSSL * ssl,  
    int * required  
)
```

この関数は、鍵のロールオーバーが進行中かどうかを判断するために、TLS v1.3 クライアントまたはサーバ wolfSSL で呼び出されます。wolfSSL_update_keys() が呼び出されると、KeyUpdate メッセージが送信され、暗号化鍵が更新されます。復号鍵は、応答を受信したときに更新されます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **required** 鍵更新応答が不要な場合は 0。鍵更新応答が必要な場合は 1。

See: wolfSSL_update_keys

Return:

- 0 成功した場合。
- BAD_FUNC_ARG ssl が NULL または TLS v1.3 を使用していない場合。

Example

```

int ret;
WOLFSSL* ssl;
int required;
...
ret = wolfSSL_key_update_response(ssl, &required);
if (ret != 0) {
    // 不正なパラメータ
}
if (required) {
    // 暗号化鍵が更新され、復号鍵を変更するための応答を待っています。
}

```

C.52.2.435 function wolfSSL_CTX_allow_post_handshake_auth

```

int wolfSSL_CTX_allow_post_handshake_auth(
    WOLFSSL_CTX * ctx
)

```

この関数は、サーバからの要求に応じてクライアント証明書をハンドシェイク後に送信できるようにするために、TLS v1.3 クライアント wolfSSL コンテキストで呼び出されます。これは、クライアント認証が必要なページとそうでないページを持つ Web サーバに接続する際に便利です。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- `wolfSSL_allow_post_handshake_auth`
- `wolfSSL_request_certificate`

Return:

- BAD_FUNC_ARG ctx が NULL または TLS v1.3 を使用していない場合。
- SIDE_ERROR サーバで呼び出された場合。
- 0 成功した場合。

Example

```

int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_allow_post_handshake_auth(ctx);
if (ret != 0) {
    // ハンドシェイク後認証の許可に失敗しました。
}

```

C.52.2.436 function wolfSSL_allow_post_handshake_auth

```

int wolfSSL_allow_post_handshake_auth(
    WOLFSSL * ssl
)

```

この関数は、サーバからの要求に応じてクライアント証明書をハンドシェイク後に送信できるようにするために、TLS v1.3 クライアント wolfSSL で呼び出されます。Post-Handshake Client Authentication 拡張が ClientHello で送信されます。これは、クライアント認証が必要なページとそうでないページを持つ Web サーバに接続する際に便利です。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_CTX_allow_post_handshake_auth`
- `wolfSSL_request_certificate`

Return:

- `BAD_FUNC_ARG` ssl が NULL または TLS v1.3 を使用していない場合。
- `SIDE_ERROR` サーバで呼び出された場合。
- 0 成功した場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_allow_post_handshake_auth(ssl);
if (ret != 0) {
    // ハンドシェイク後認証の許可に失敗しました。
}
```

C.52.2.437 function `wolfSSL_request_certificate`

```
int wolfSSL_request_certificate(
    WOLFSSL * ssl
)
```

この関数は、TLS v1.3 クライアントからクライアント証明書を要求します。これは、Web サーバがクライアント認証を必要とするページと必要としないページの両方を提供している場合に便利です。接続上で最大 256 回の要求を送信できます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_allow_post_handshake_auth`
- `wolfSSL_write`

Return:

- `BAD_FUNC_ARG` ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- `WANT_WRITE` 書き込みの準備ができていない場合。
- `SIDE_ERROR` クライアントで呼び出された場合。
- `NOT_READY_ERROR` ハンドシェイクが完了していないときに呼び出された場合。
- `POST_HAND_AUTH_ERROR` ポストハンドシェイク認証が許可されていない場合。
- `MEMORY_E` 動的メモリ割り当てが失敗した場合。
- `WOLFSSL_SUCCESS` 成功時。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_request_certificate(ssl);
if (ret == WANT_WRITE) {
    // I/O の準備ができれば再度呼び出す必要があります
}
else if (ret != WOLFSSL_SUCCESS) {
```

```
// クライアント証明書の要求に失敗しました
}
```

C.52.2.438 function wolfSSL_CTX_set1_sigalgs_list

```
int wolfSSL_CTX_set1_sigalgs_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

この関数は、wolfSSL コンテキストで優先順位に従って許可する署名アルゴリズムのリストを設定します。リストは null 終端のテキスト文字列で、コロン区切りのリストです。各要素は “+” の形式（例: “RSA-PSS+SHA256:ECDSA+SHA384”）で指定します。Edwards 曲線アルゴリズムである “ED25519” および “ED448” はハッシュがアルゴリズムに内包されるため、サフィックスを付けずに記述します。コンテキストに既に保持されているリストは置き換えられます。

Parameters:

- **ctx** `wolfSSL_CTX_new()` で作成された WOLFSSL_CTX 構造体へのポインタ。
- **list** “+” 要素（または digest を伴わない “ED25519” / “ED448”）のコロン区切りリスト。

See:

- `wolfSSL_set1_sigalgs_list`
- `wolfSSL_CTX_set1_groups_list`

Return:

- WOLFSSL_SUCCESS 成功時。
- WOLFSSL_FAILURE ポインタパラメータが NULL の場合、suites 構造体の確保に失敗した場合、またはリスト中のいずれかのトークンが認識されないか現在のビルドでサポートされていない場合。

認識される公開鍵トークンとハッシュトークン、および各トークンを利用するために必要なビルドオプションを以下の表に示します。

公開鍵トークン:

トークン	必要なビルドオプション
RSA	!NO_RSA
RSA-PSS / PSS	!NO_RSA かつ WC_RSA_PSS
ECDSA	HAVE_ECC
ED25519	HAVE_ED25519（ハッシュ指定不要）
ED448	HAVE_ED448（ハッシュ指定不要）
DSA	!NO_DSA
SM2	WOLFSSL_SM2 かつ WOLFSSL_SM3（ハッシュは SM3 固定）

ハッシュトークン:

トークン	必要なビルドオプション
SHA256	!NO_SHA256
SHA384	WOLFSSL_SHA384
SHA512	WOLFSSL_SHA512
SHA224	WOLFSSL_SHA224
SM3	WOLFSSL_SM3
SHA1	!NO_SHA かつ (!NO_OLD_TLS または WOLFSSL_ALLOW_TLS_SHA1)

TLS 1.3 における注意事項: RFC 8446 の規定により、RSA PKCS#1 v1.5、DSA、SHA-1、SHA-224 はハッシュ署名としては使用できず、リストに含まれていてもネゴシエーション時に除外されます。“RSA-PSS+SHAxxx”を指定した場合は `rsa_pss_rsae_shaxxx` と `rsa_pss_pss_shaxxx` の両方のスキームが追加されます。Brainpool ECDSA 署名スキーム (RFC 8734) は本文字列インタフェースでは選択できず、HAVE_ECC_BRAINPOOL が有効な場合に自動でネゴシエーションされます。

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) {
    // 署名アルゴリズムリストの設定に失敗しました
}
```

C.52.2.439 function wolfSSL_set1_sigalgs_list

```
int wolfSSL_set1_sigalgs_list(
    WOLFSSL * ssl,
    const char * list
)
```

この関数は、wolfSSL セッションで優先順位に従って許可する署名アルゴリズムのリストを設定します。リストの形式、認識される公開鍵トークンおよびハッシュトークンは `wolfSSL_CTX_set1_sigalgs_list()` の説明を参照してください。セッションに既に保持されているリストは置き換えられます。

Parameters:

- `ssl` `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- `list` “+” 要素（または digest を伴わない “ED25519” / “ED448”）のコロン区切りリスト。

See:

- `wolfSSL_CTX_set1_sigalgs_list`
- `wolfSSL_set1_groups_list`

Return:

- WOLFSSL_SUCCESS 成功時。
- WOLFSSL_FAILURE ポインタパラメータが NULL の場合、suites 構造体の確保に失敗した場合、またはリスト中のいずれかのトークンが認識されないか現在のビルドでサポートされていない場合。

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) {
    // 署名アルゴリズムリストの設定に失敗しました
}
```

C.52.2.440 function wolfSSL_CTX_set1_groups_list

```
int wolfSSL_CTX_set1_groups_list(
    WOLFSSL_CTX * ctx,
```



```
const char * list
)
```

この関数は、wolfSSL コンテキストで優先順位に従って許可する鍵交換グループ（名前付き楕円曲線および KEM）のリストを設定します。リストは null 終端のテキスト文字列で、グループ名をコロンで区切って指定します（例: "P-384:P-256:X25519"）。TLS v1.3 接続で使用する鍵交換パラメータを設定するために本関数を呼び出します（本関数は HAVE_ECC、WOLFSSL_TLS13、HAVE_SUPPORTED_CURVES がすべて定義された場合にのみ提供されます）。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された WOLFSSL_CTX 構造体へのポインタ。
- **list** 鍵交換グループ名のコロン区切りリストである文字列。

See:

- wolfSSL_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return:

- WOLFSSL_FAILURE ポインタパラメータが NULL の場合、グループが WOLFSSL_MAX_GROUP_COUNT を超える場合、グループ名が認識されない場合、または TLS v1.3 を使用していない場合。
- WOLFSSL_SUCCESS 成功時。

認識されるグループ名と必要なビルドオプションを以下に示します。名前は表とケースセンシティブに比較されます。

NIST / SEC 曲線（HAVE_ECC が必要）：

名前	曲線 / グループ
P-160	secp160r1
P-160-2	secp160r2
P-192	secp192r1 (prime192v1)
P-224	secp224r1
P-256	secp256r1 (prime256v1) — "prime256v1" / "secp256r1" でも指定可
P-384	secp384r1 — "secp384r1" でも指定可
P-521	secp521r1 — "secp521r1" でも指定可
K-160	secp160k1
K-192	secp192k1
K-224	secp224k1
K-256	secp256k1

Brainpool 曲線（HAVE_ECC に加えて WOLFSSL_CUSTOM_CURVES と HAVE_ECC_BRAINPOOL が必要。通常は -enable-eccustcurves=all で有効化）：

名前	曲線
B-256	brainpoolP256r1
B-384	brainpoolP384r1
B-512	brainpoolP512r1

Edwards / Montgomery 曲線:

名前	必要なビルドオプション
X25519	HAVE_CURVE25519
X448	HAVE_CURVE448

SM2 (WOLFSSL_SM2 が必要) :

名前	グループ
SM2	sm2p256v1
sm2p256v1	sm2p256v1 (別名)

ML-KEM (耐量子) グループ (WOLFSSL_HAVE_MLKEM かつ!WOLFSSL_NO_ML_KEM が必要) :

名前
ML_KEM_512
ML_KEM_768
ML_KEM_1024

ML-KEM ハイブリッドグループは、上記に加えて HAVE_ECC、および WOLFSSL_WC_MLKEM または HAVE_LIBOQS、さらに WOLFSSL_PQC_HYBRIDS (“extra” セットには WOLFSSL_EXTRA_PQC_HYBRIDS) が必要です:

名前	必要なハイブリッドフラグ
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

レガシー Kyber グループ (WOLFSSL_MLKEM_KYBER が必要。ハイブリッドはさらに HAVE_ECC と、WOLFSSL_WC_MLKEM または HAVE_LIBOQS が必要) :

名前
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

上記の名前に加え、HAVE_FIPS および HAVE_SELFTEST が定義されていないビルドでは、wolfCrypt に登録されている任意の曲線名 (wc_ecc_get_curve_idx_from_name()) で参照、例: "brainpoolP256r1") も受理されます。

リストに指定した名前の順序はそのまま保持され、TLS 1.3 における KeyShare の優先順位として使用されます。

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) {
    // グループリストの設定に失敗しました
}
```

C.52.2.441 function wolfSSL_set1_groups_list

```
int wolfSSL_set1_groups_list(
    WOLFSSL * ssl,
    const char * list
)
```

この関数は、wolfSSL セッションで優先順位に従って許可する鍵交換グループ (名前付き楕円曲線および KEM) のリストを設定します。リストの形式および認識されるグループ名は `wolfSSL_CTX_set1_groups_list()` の説明を参照してください。

Parameters:

- `ssl` `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- `list` 鍵交換グループのコロン区切りリストである文字列。

See:

- `wolfSSL_CTX_set1_groups_list`
- `wolfSSL_CTX_set_groups`
- `wolfSSL_set_groups`
- `wolfSSL_UseKeyShare`
- `wolfSSL_preferred_group`

Return:

- WOLFSSL_FAILURE ポインタパラメータが NULL の場合、グループが WOLFSSL_MAX_GROUP_COUNT を超える場合、グループ名が認識されない場合、または TLS v1.3 を使用していない場合。
- WOLFSSL_SUCCESS 成功時。

Example

```
int ret;
WOLFSSL* ssl;
const char* list = "P-384:P-256";
...
ret = wolfSSL_CTX_set1_groups_list(ssl, list);
if (ret != WOLFSSL_SUCCESS) {
    // グループリストの設定に失敗しました
}
```

C.52.2.442 function wolfSSL_preferred_group

```
int wolfSSL_preferred_group(
    WOLFSSL * ssl
)
```

この関数は、TLS v1.3 ハンドシェイクでクライアントが優先的に使用したい鍵交換グループを返します。ハンドシェイク完了後にこの関数を呼び出して、サーバが優先するグループを判定し、この情報を将来の接続で使用して鍵交換用の鍵ペアを事前生成できます。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_UseKeyShare`
- `wolfSSL_CTX_set_groups`
- `wolfSSL_set_groups`
- `wolfSSL_CTX_set1_groups_list`
- `wolfSSL_set1_groups_list`

Return:

- `BAD_FUNC_ARG` ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- `SIDE_ERROR` サーバで呼び出された場合。
- `NOT_READY_ERROR` ハンドシェイク完了前に呼び出された場合。
- `Group identifier` 成功時、グループ識別子。

Example

```
int ret;
int group;
WOLFSSL* ssl;
...
ret = wolfSSL_CTX_set1_groups_list(ssl)
if (ret < 0) {
    // グループの取得に失敗しました
}
group = ret;
```

C.52.2.443 function wolfSSL_CTX_set_groups

```
int wolfSSL_CTX_set_groups(
    WOLFSSL_CTX * ctx,
    int * groups,
    int count
)
```

この関数は、wolfSSL コンテキストで優先順位に従って許可する鍵交換グループ（名前付き楕円曲線および KEM）のリストを設定します。リストは名前付きグループ識別子（下表参照）の配列で、count に配列内の識別子数を指定します。TLS v1.3 接続で使用する鍵交換パラメータを設定するために本関数を呼び出します。配列の順序がそのまま KeyShare の優先順位として用いられます。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX 構造体へのポインタ。
- **groups** 識別子による鍵交換グループのリスト。
- **count** groups 内の識別子数（WOLFSSL_MAX_GROUP_COUNT を超えてはならない）。

See:

- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_preferred_group

Return:

- BAD_FUNC_ARG ポインタパラメータが NULL の場合、count が WOLFSSL_MAX_GROUP_COUNT を超える場合、または基底メソッドが TLS メソッドでない場合。
- WOLFSSL_SUCCESS 成功時。

認識される識別子と必要なビルドオプションを以下に示します。識別子は <wolfssl/ssl.h> 内の無名 enum で定義されています。

NIST / SEC 曲線 (HAVE_ECC が必要) :

識別子	曲線 / グループ
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 曲線 (HAVE_ECC に加えて WOLFSSL_CUSTOM_CURVES と HAVE_ECC_BRAINPOOL が必要。通常は -enable-eccustcurves=all で有効化) :

識別子	曲線	備考
WOLFSSL_ECC_BRAINPOOLP256R1	brainpoolP256r1	TLS 1.2 グループ ID 26
WOLFSSL_ECC_BRAINPOOLP384R1	brainpoolP384r1	TLS 1.2 グループ ID 27
WOLFSSL_ECC_BRAINPOOLP512R1	brainpoolP512r1	TLS 1.2 グループ 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 曲線:

識別子	必要なビルドオプション
WOLFSSL_ECC_X25519	HAVE_CURVE25519
WOLFSSL_ECC_X448	HAVE_CURVE448

SM2 (WOLFSSL_SM2 が必要) :

識別子	グループ
WOLFSSL_ECC_SM2P256V1	sm2p256v1

有限体 DH (RFC 7919) グループ (HAVE_FFDHE および各サイズに対応する HAVE_FFDHE_NNNN マクロが必要) :

識別子	グループ
WOLFSSL_FFDHE_2048	ffdhe2048
WOLFSSL_FFDHE_3072	ffdhe3072
WOLFSSL_FFDHE_4096	ffdhe4096
WOLFSSL_FFDHE_6144	ffdhe6144
WOLFSSL_FFDHE_8192	ffdhe8192

ML-KEM (耐量子) グループ (HAVE_PQC、WOLFSSL_HAVE_MLKEM、!WOLFSSL_NO_ML_KEM が必要) :

識別子
WOLFSSL_ML_KEM_512
WOLFSSL_ML_KEM_768
WOLFSSL_ML_KEM_1024

ML-KEM ハイブリッドグループは、上記に加えて HAVE_ECC、および WOLFSSL_WC_MLKEM または HAVE_LIBOQS、さらに WOLFSSL_PQC_HYBRIDS (“extra” セットには WOLFSSL_EXTRA_PQC_HYBRIDS) が必要です:

識別子	必要なハイブリッドフラグ
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

レガシー Kyber グループ (HAVE_PQC と WOLFSSL_MLKEM_KYBER が必要。ハイブリッドはさらに HAVE_ECC と、WOLFSSL_WC_MLKEM または HAVE_LIBOQS が必要) :

識別子
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

 識別子

 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) {
    // グループリストの設定に失敗しました
}
```

C.52.2.444 function wolfSSL_set_groups

```
int wolfSSL_set_groups(
    WOLFSSL * ssl,
    int * groups,
    int count
)
```

この関数は、wolfSSL セッションで優先順位に従って許可する鍵交換グループ（名前付き楕円曲線および KEM）のリストを設定します。配列の形式および認識される識別子は `wolfSSL_CTX_set_groups()` の説明を参照してください。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。
- **groups** 識別子による鍵交換グループのリスト。
- **count** groups 内の識別子数（WOLFSSL_MAX_GROUP_COUNT を超えてはならない）。

See:

- `wolfSSL_CTX_set_groups`
- `wolfSSL_UseKeyShare`
- `wolfSSL_CTX_set1_groups_list`
- `wolfSSL_set1_groups_list`
- `wolfSSL_preferred_group`

Return:

- BAD_FUNC_ARG ポインタパラメータが NULL の場合、count が WOLFSSL_MAX_GROUP_COUNT を超える場合、識別子のいずれかが認識されない場合、または基底メソッドが TLS メソッドでない場合。
- WOLFSSL_SUCCESS 成功時。

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) {
```

```
// グループリストの設定に失敗しました
}
```

C.52.2.445 function wolfSSL_connect_TLSv13

```
int wolfSSL_connect_TLSv13(
    WOLFSSL * ssl
)
```

この関数はクライアント側で呼び出され、サーバとの TLS v1.3 ハンドシェイクを開始します。この関数が呼び出されるとき、基礎となる通信チャンネルはすでに設定されています。wolfSSL_connect() はブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、基礎となる I/O が wolfSSL_connect() がハンドシェイクを続行するために必要なものを満たせない場合、wolfSSL_connect() は返されます。この場合、wolfSSL_get_error() を呼び出すと SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、基礎となる I/O の準備ができたときに wolfSSL_connect() の呼び出しを繰り返す必要があり、wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキング I/O の場合、wolfSSL_connect() はハンドシェイクが完了するかエラーが発生するまで返されません。wolfSSL は証明書検証に OpenSSL とは異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバを検証することです。つまり、サーバを検証するための CA をロードしない場合、接続エラー「検証できません (-155)」が発生します。サーバの検証が失敗しても SSL_connect が成功するという OpenSSL の動作を模倣し、セキュリティを低下させたい場合は、SSL_new() を呼び出す前に SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0) を呼び出すことでこれを行うことができます。ただし、これは推奨されません。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功時。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出してください。

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.52.2.446 function wolfSSL_accept_TLSv13


```

wolfSSL_accept_TLSv13(
    WOLFSSL * ssl
)

```

この関数はサーバ側で呼び出され、SSL/TLS クライアントが SSL/TLS ハンドシェイクを開始するのを待ちます。この関数が呼び出されるとき、基礎となる通信チャネルはすでに設定されています。wolfSSL_accept() はブロッキング I/O と非ブロッキング I/O の両方で動作します。基礎となる I/O が非ブロッキングの場合、基礎となる I/O が wolfSSL_accept() がハンドシェイクを続行するために必要なものを満たせない場合、wolfSSL_accept() は返されます。この場合、wolfSSL_get_error() を呼び出すと SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出し側プロセスは、データが読み取り可能になったときに wolfSSL_accept の呼び出しを繰り返す必要があります。wolfSSL は中断したところから再開します。非ブロッキングソケットを使用する場合、何もする必要はありませんが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロッキング I/O の場合、wolfSSL_accept() はハンドシェイクが完了するかエラーが発生するまで返されません。TLS v1.3 接続を期待する場合にこの関数を呼び出してください。ただし、古いバージョンの ClientHello メッセージもサポートされています。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_connect_TLSv13
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功時。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出してください。

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.52.2.447 function wolfSSL_CTX_set_max_early_data

```

int wolfSSL_CTX_set_max_early_data(
    WOLFSSL_CTX * ctx,
    unsigned int sz
)

```

この関数は、TLS v1.3 クライアントまたはサーバが wolfSSL コンテキストを使用して交換する意思のある早期データの最大量を設定します。リプレイ攻撃を軽減するために処理する早期データの量を制限するには、この関数を呼び出してください。早期データは、セッションチケットが送信された接続の鍵から派生した鍵によって保護されるため、セッションチケットが再開に使用されるたびに同じになります。この値は再

開用のセッションチケットに含まれます。サーバの値がゼロの場合、セッションチケットを使用してクライアントが早期データを送信しないことを示します。クライアントの値がゼロの場合、クライアントが早期データを送信しないことを示します。早期データのバイト数は、アプリケーションで実用的に可能な限り低く保つことをお勧めします。

Parameters:

- **ctx** `wolfSSL_CTX_new()`で作成された WOLFSSL_CTX 構造体へのポインタ。
- **sz** 受け入れる早期データの量 (バイト単位)。

See:

- `wolfSSL_set_max_early_data`
- `wolfSSL_write_early_data`
- `wolfSSL_read_early_data`

Return:

- BAD_FUNC_ARG ctx が NULL の場合、または TLS v1.3 を使用していない場合。
- 0 成功時。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_set_max_early_data(ctx, 128);
if (ret != WOLFSSL_SUCCESS) {
    // グループリストの設定に失敗しました
}
```

C.52.2.448 function wolfSSL_set_max_early_data

```
int wolfSSL_set_max_early_data(
    WOLFSSL * ssl,
    unsigned int sz
)
```

この関数は、TLS v1.3 クライアントまたはサーバが交換する意思のある早期データの最大量を設定します。リプレイ攻撃を軽減するために処理する早期データの量を制限するには、この関数を呼び出してください。早期データは、セッションチケットが送信された接続の鍵から派生した鍵によって保護されるため、セッションチケットが再開に使用されるたびに同じになります。この値は再開用のセッションチケットに含まれます。サーバの値がゼロの場合、セッションチケットを使用してクライアントが早期データを送信しないことを示します。クライアントの値がゼロの場合、クライアントが早期データを送信しないことを示します。早期データのバイト数は、アプリケーションで実用的に可能な限り低く保つことをお勧めします。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **sz** クライアントから受け入れる早期データの量 (バイト単位)。

See:

- `wolfSSL_CTX_set_max_early_data`
- `wolfSSL_write_early_data`
- `wolfSSL_read_early_data`

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- 0 成功時。 *Example*

```

int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_set_max_early_data(ssl, 128);
if (ret != WOLFSSL_SUCCESS) {
    // グループリストの設定に失敗しました
}

```

C.52.2.449 function wolfSSL_write_early_data

```

int wolfSSL_write_early_data(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int * outSz
)

```

この関数は再開時にサーバーにアーリーデータを書き込みます。wolfSSL_connect() または wolfSSL_connect_TLSv13() の前にこの関数を呼び出してください。この関数はクライアントでのみ使用されません。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **data** サーバーに書き込むアーリーデータを保持するバッファ。
- **sz** 書き込むアーリーデータの量 (バイト単位)。
- **outSz** 書き込まれたアーリーデータの量 (バイト単位)。

See:

- wolfSSL_read_early_data
- wolfSSL_connect
- wolfSSL_connect_TLSv13

Return:

- BAD_FUNC_ARG ポインタパラメータが NULL、sz が 0 未満、または TLSv1.3 を使用していない場合。
- SIDE_ERROR サーバーで呼び出された場合。
- WOLFSSL_FATAL_ERROR 接続が確立されなかった場合。
- 成功した場合、書き込まれたアーリーデータのバイト数。

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)) {
    // すべてのアーリーデータが送信されませんでした
}

```

```

}
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.52.2.450 function wolfSSL_read_early_data

```

int wolfSSL_read_early_data(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int * outSz
)

```

この関数は再開時にクライアントからのアーリーデータを読み取ります。クライアントを受け入れ、ハンドシェイクでアーリーデータを読み取るには、wolfSSL_accept() または wolfSSL_accept_TLsv13() の代わりにこの関数を呼び出します。wolfSSL_is_init_finished() が true を返すまで関数を呼び出す必要があります。アーリーデータは複数のメッセージでクライアントから送信される場合があります。アーリーデータがない場合、ハンドシェイクは通常通り処理されます。この関数はサーバーでのみ使用されます。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **data** クライアントから読み取ったアーリーデータを保持するバッファ。
- **sz** バッファのサイズ (バイト単位)。
- **outSz** 読み取ったアーリーデータのバイト数。

See:

- wolfSSL_write_early_data
- wolfSSL_accept
- wolfSSL_accept_TLsv13

Return:

- BAD_FUNC_ARG ポインタパラメータが NULL、sz が 0 未満、または TLSv1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。
- WOLFSSL_FATAL_ERROR 接続の受け入れが失敗した場合。
- 読み取ったアーリーデータのバイト数 (ゼロの場合もあります)。

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) {

```

```

    // アーリーデータが利用可能
}
} while (!wolfSSL_is_init_finished(ssl));

```

C.52.2.451 function wolfSSL_inject

```

int wolfSSL_inject(
    WOLFSSL * ssl,
    const void * data,
    int sz
)

```

この関数は WOLFSSL オブジェクトにデータを注入するために呼び出されます。これは、データを単一の場所から読み取り、複数の接続に分割する必要がある場合に便利です。呼び出し元は wolfSSL_read() を呼び出して WOLFSSL オブジェクトから平文データを抽出する必要があります。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **data** ssl オブジェクトに注入するデータ。
- **sz** 注入するデータのバイト数。

See: [wolfSSL_read](#)

Return:

- BAD_FUNC_ARG いずれかのポインタパラメータが NULL または sz <= 0 の場合。
- APP_DATA_READY 読み取るべきアプリケーションデータが残っている場合。
- MEMORY_E 割り当てが失敗した場合。
- WOLFSSL_SUCCESS 成功時。

Example

```

byte buf[2000]
sz = recv(fd, buf, sizeof(buf), 0);
if (sz <= 0)
    // エラー
if (wolfSSL_inject(ssl, buf, sz) != WOLFSSL_SUCCESS)
    // エラー
sz = wolfSSL_read(ssl, buf, sizeof(buf));

```

C.52.2.452 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
)

```

この関数は TLS v1.3 接続のための事前共有鍵 (PSK) クライアント側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL_CTX 構造体の client_psk_tls13_cb メンバーを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new() で作成された WOLFSSL_CTX 構造体へのポインタ。
- **cb** TLS 1.3 クライアント用の事前共有鍵 (PSK) コールバック。

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.52.2.453 function `wolfSSL_set_psk_client_tls13_callback`

```
void wolfSSL_set_psk_client_tls13_callback(  
    WOLFSSL * ssl,  
    wc_psk_client_tls13_callback cb  
)
```

この関数は TLS v1.3 接続のための事前共有鍵 (PSK) クライアント側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL 構造体の options フィールドの client_psk_tls13_cb メンバーを設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** TLS 1.3 クライアント用の事前共有鍵 (PSK) コールバック。

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.52.2.454 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  
)
```

この関数は TLS v1.3 接続のための事前共有鍵 (PSK) サーバー側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL_CTX 構造体の server_psk_tls13_cb メンバーを設定します。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された WOLFSSL_CTX 構造体へのポインタ。
- **cb** TLS 1.3 サーバー用の事前共有鍵 (PSK) コールバック。

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.52.2.455 function wolfSSL_set_psk_server_tls13_callback

```
void wolfSSL_set_psk_server_tls13_callback(  
    WOLFSSL * ssl,  
    wc_psk_server_tls13_callback cb  
)
```

この関数は TLS v1.3 接続のための事前共有鍵 (PSK) サーバー側コールバックを設定します。コールバックは PSK アイデンティティを検索し、その鍵とハンドシェイクで使用する暗号の名前を返すために使用されます。この関数は WOLFSSL 構造体の options フィールドの server_psk_tls13_cb メンバーを設定します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** TLS 1.3 サーバー用の事前共有鍵 (PSK) コールバック。

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.52.2.456 function wolfSSL_UseKeyShare

```
int wolfSSL_UseKeyShare(  
    WOLFSSL * ssl,  
    word16 group  
)
```

この関数は鍵ペアの生成を含む、グループから鍵共有エントリを作成します。KeyShare 拡張には鍵交換用に生成されたすべての公開鍵が含まれます。この関数が呼び出されると、指定されたグループのみが含まれます。サーバーに対して優先グループが以前に確立されている場合に、この関数を呼び出します。

Parameters:

- **ssl** [wolfSSL_new\(\)](#) を使用して作成された WOLFSSL 構造体へのポインタ。
- **group** 鍵交換グループ識別子。

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 ssl が NULL の場合。
- MEMORY_E 動的メモリ割り当てが失敗した場合。

- WOLFSSL_SUCCESS 成功した場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_UseKeyShare(ssl, WOLFSSL_ECC_X25519);
if (ret != WOLFSSL_SUCCESS) {
    // 鍵共有の設定に失敗しました
}
```

C.52.2.457 function wolfSSL_NoKeyShares

```
int wolfSSL_NoKeyShares(
    WOLFSSL * ssl
)
```

この関数は ClientHello で鍵共有が送信されないようにするために呼び出されます。これにより、ハンドシェイクで鍵交換が必要な場合、サーバーは HelloRetryRequest で応答することになります。期待される鍵交換グループが不明で、不要な鍵の生成を避けたい場合にこの関数を呼び出します。鍵交換が必要な場合、ハンドシェイクを完了するために追加のラウンドトリップが必要になることに注意してください。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。

See: [wolfSSL_UseKeyShare](#)

Return:

- BAD_FUNC_ARG ssl が NULL の場合。
- SIDE_ERROR サーバーで呼び出された場合。
- WOLFSSL_SUCCESS 成功した場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_NoKeyShares(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // 鍵共有なしの設定に失敗しました
}
```

C.52.2.458 function wolfTLSv1_3_server_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method_ex(
    void * heap
)
```

この関数はアプリケーションがサーバーであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、[wolfSSL_CTX_new\(\)](#) で SSL/TLS コンテキストを作成する際に使用される新しい [WOLFSSL_METHOD](#) 構造体のメモリを割り当て、初期化します。

Parameters:

- **heap** 静的メモリアロケータが動的メモリ割り当て時に使用するバッファへのポインタ。

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:

- 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL_XMALLOC の呼び出し時にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method_ex(NULL);
if (method == NULL) {
    // メソッドの取得ができません
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.459 function wolfTLSv1_3_client_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method_ex(
    void * heap
)
```

この関数はアプリケーションがクライアントであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体のメモリを割り当て、初期化します。

Parameters:

- **heap** 静的メモリアロケータが動的メモリ割り当て時に使用するバッファへのポインタ。

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:

- 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL_XMALLOC の呼び出し時にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常は NULL で、errno が ENOMEM に設定されます)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;    method = wolfTLSv1_3_client_method_ex(NULL);
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.460 function wolfTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method(
    void
)
```

この関数は、アプリケーションがサーバーであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体用のメモリを割り当て、初期化します。

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:

- 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC 呼び出し時にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常、errno が ENOMEM に設定された NULL)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.461 function wolfTLSv1_3_client_method

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method(
    void
)
```

この関数は、アプリケーションがクライアントであり、TLS 1.3 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() で SSL/TLS コンテキストを作成する際に使用される新しい wolfSSL_METHOD 構造体用のメモリを割り当て、初期化します。

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:

- 成功した場合、呼び出しは新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- FAIL XMALLOC 呼び出し時にメモリ割り当てが失敗した場合、基礎となる malloc() 実装の失敗値が返されます (通常、errno が ENOMEM に設定された NULL)。

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_client_method();
if (method == NULL) {
    // メソッドを取得できません
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.462 function wolfTLSv1_3_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_method_ex(
    void * heap
)
```

この関数は、どちら側 (サーバー/クライアント) であるかがまだ決定されていないことを除いて、wolfTLSv1_3_client_method と同様の WOLFSSL_METHOD を返します。

Parameters:

- **heap** 静的メモリアロケータが動的メモリ割り当て中に使用するバッファへのポインタ。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD 正常に作成された場合、WOLFSSL_METHOD ポインタを返します。

- NULL メモリ割り当てエラーまたはメソッドの作成に失敗した場合は Null。

Example

```
WOLFSSL* ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
// ret 値を確認
```

C.52.2.463 function wolfTLSv1_3_method

```
WOLFSSL_METHOD * wolfTLSv1_3_method(
    void
)
```

この関数は、どちら側 (サーバー/クライアント) であるかがまだ決定されていないことを除いて、wolfTLSv1_3_client_method と同様の WOLFSSL_METHOD を返します。

See:

- [wolfSSL_new](#)
- [wolfSSL_free](#)

Return:

- WOLFSSL_METHOD 正常に作成された場合、WOLFSSL_METHOD ポインタを返します。
- NULL メモリ割り当てエラーまたはメソッドの作成に失敗した場合は Null。

Example

```
WOLFSSL* ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_3_method());
// ret 値を確認
```

C.52.2.464 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
)
```

この関数は、テスト専用の固定/静的エフェメラル鍵を設定します。

Parameters:

- **ctx** WOLFSSL_CTX コンテキストポインタ。
- **keyAlgo** WC_PK_TYPE_DH や WC_PK_TYPE_ECDH などの enum wc_PkType。
- **key** 鍵ファイルパス (keySz == 0 の場合) または実際の鍵バッファ (PEM または ASN.1)。
- **keySz** 鍵サイズ ("key" 引数がファイルパスの場合は 0 である必要があります)。
- **format** WOLFSSL_FILETYPE_ASN1 または WOLFSSL_FILETYPE_PEM。

See: [wolfSSL_CTX_get_ephemeral_key](#)

Return: 0 鍵が正常にロードされました。

C.52.2.465 function wolfSSL_set_ephemeral_key

```
int wolfSSL_set_ephemeral_key(
    WOLFSSL * ssl,
```

```

    int keyAlgo,
    const char * key,
    unsigned int keySz,
    int format
)

```

この関数は、テスト専用の固定/静的エフェメラル鍵を設定します。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **keyAlgo** WC_PK_TYPE_DH や WC_PK_TYPE_ECDH などの enum wc_PkType。
- **key** 鍵ファイルパス (keySz == 0 の場合) または実際の鍵バッファ (PEM または ASN.1)。
- **keySz** 鍵サイズ ("key" 引数がファイルパスの場合は 0 である必要があります)。
- **format** WOLFSSL_FILETYPE_ASN1 または WOLFSSL_FILETYPE_PEM。

See: [wolfSSL_get_ephemeral_key](#)

Return: 0 鍵が正常にロードされました。

C.52.2.466 function wolfSSL_CTX_get_ephemeral_key

```

int wolfSSL_CTX_get_ephemeral_key(
    WOLFSSL_CTX * ctx,
    int keyAlgo,
    const unsigned char ** key,
    unsigned int * keySz
)

```

この関数は、ASN.1/DER としてロードされた鍵へのポインタを返します。

Parameters:

- **ctx** WOLFSSL_CTX コンテキストポインタ。
- **keyAlgo** WC_PK_TYPE_DH や WC_PK_TYPE_ECDH などの enum wc_PkType。
- **key** 鍵バッファポインタ。
- **keySz** 鍵サイズポインタ。

See: [wolfSSL_CTX_set_ephemeral_key](#)

Return: 0 鍵が正常に返されました。

C.52.2.467 function wolfSSL_get_ephemeral_key

```

int wolfSSL_get_ephemeral_key(
    WOLFSSL * ssl,
    int keyAlgo,
    const unsigned char ** key,
    unsigned int * keySz
)

```

この関数は、ASN.1/DER としてロードされた鍵へのポインタを返します。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **keyAlgo** WC_PK_TYPE_DH や WC_PK_TYPE_ECDH などの enum wc_PkType。
- **key** 鍵バッファポインタ。
- **keySz** 鍵サイズポインタ。

See: [wolfSSL_set_ephemeral_key](#)

Return: 0 鍵が正常に返されました。

C.52.2.468 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  
)
```

選択したメッセージダイジェスト、パディング、および RSA 鍵でメッセージに署名します。

Parameters:

- **hashAlg** ハッシュ NID。
- **hash** 署名するメッセージ。おそらくこれは署名するメッセージのダイジェストになります。
- **hLen** 署名するメッセージの長さ。
- **sigRet** 出力バッファ。
- **sigLen** 入力時: sigRet バッファの長さ、出力時: sigRet に書き込まれたデータの長さ。
- **rsa** 入力の署名に使用される RSA 鍵。
- **flag** 1: 署名を出力、0: パディングされていない署名と比較すべき値を出力。注意: RSA_PKCS1_PSS_PADDING の場合、Verify 関数の出力をチェックするために wc_RsaPSS_CheckPadding_ex 関数を使用する必要があります。
- **padding** 使用するパディング。現在、署名には RSA_PKCS1_PSS_PADDING と RSA_PKCS1_PADDING のみがサポートされています。

Return: WOLFSSL_SUCCESS 成功時、エラー時は c。

C.52.2.469 function wolfSSL_dtls13_has_pending_msg

```
int wolfSSL_dtls13_has_pending_msg(  
    WOLFSSL * ssl  
)
```

DTLSv1.3 スタックが送信したが、まだ他のピアから確認応答されていないメッセージがあるかどうかをチェックします。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。

Return: 1 保留中のメッセージがある場合、それ以外は 0。

C.52.2.470 function wolfSSL_SESSION_get_max_early_data

```
unsigned int wolfSSL_SESSION_get_max_early_data(  
    const WOLFSSL_SESSION * s  
)
```

セッションから Early Data の最大サイズを取得します。

Parameters:

- **s** WOLFSSL_SESSION インスタンス。

See:

- [wolfSSL_set_max_early_data](#)
- [wolfSSL_write_early_data](#)
- [wolfSSL_read_early_data](#)

Return: セッションが派生した WOLFSSL* で設定された max_early_data の値。

C.52.2.471 function wolfSSL_CRYPTOP_get_ex_new_index

```
int wolfSSL_CRYPTOP_get_ex_new_index(
    int class_index,
    long arg1,
    void * argp,
    WOLFSSL_CRYPTOP_EX_new * new_func,
    WOLFSSL_CRYPTOP_EX_dup * dup_func,
    WOLFSSL_CRYPTOP_EX_free * free_func
)
```

外部データ用の新しいインデックスを取得します。このエントリは以下の API にも適用されます:

Parameters:

- **class_index** 外部データインデックスが適用されるオブジェクトクラスの識別子。wolfSSL では無視されます。
- **arg1** 互換性のために渡されるオプションの long 型引数。wolfSSL では無視されます。
- **argp** 互換性のために渡されるオプションのポインタ型引数。wolfSSL では無視されます。
- **new_func** 外部データコンストラクタコールバックへのポインタ。wolfSSL では無視されます。
- **dup_func** 外部データ複製コールバックへのポインタ。wolfSSL では無視されます。
- **free_func** 外部データデストラクタコールバックへのポインタ。wolfSSL では無視されます。

Return: このオブジェクトクラスの外部データ API で使用される新しいインデックス値。

- wolfSSL_CTX_get_ex_new_index
- wolfSSL_get_ex_new_index
- wolfSSL_SESSION_get_ex_new_index
- wolfSSL_X509_get_ex_new_index

C.52.2.472 function wolfSSL_CTX_set_client_cert_type

```
int wolfSSL_CTX_set_client_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

この関数がクライアント側で呼び出された場合、ピアに送信できる証明書タイプを設定します。サーバー側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が "Raw public key" 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクトポインタ。
- **buf** 証明書タイプが格納されるバッファ。
- **len** buf サイズ (バイト単位)(含まれる証明書タイプの数と同じ)。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 証明書タイプが正常に設定された場合。
- BAD_FUNC_ARG ctx に NULL が渡された場合、証明書タイプとして不正な値が指定された場合、buf サイズが MAX_CLIENT_CERT_TYPE_CNT を超えた場合、または buf 内に重複する値が見つかった場合。

C.52.2.473 function `wolfSSL_CTX_set_server_cert_type`

```
int wolfSSL_CTX_set_server_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

この関数がサーバー側で呼び出された場合、ピアに送信できる証明書タイプを設定します。クライアント側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が “Raw public key” 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクトポインタ。
- **buf** 証明書タイプが格納されるバッファ。
- **len** buf サイズ (バイト単位)(含まれる証明書タイプの数と同じ)。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 証明書タイプが正常に設定された場合。
- BAD_FUNC_ARG ctx に NULL が渡された場合、証明書タイプとして不正な値が指定された場合、buf サイズが MAX_SERVER_CERT_TYPE_CNT を超えた場合、または buf 内に重複する値が見つかった場合。

C.52.2.474 function wolfSSL_set_client_cert_type

```
int wolfSSL_set_client_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

この関数がクライアント側で呼び出された場合、ピアに送信できる証明書タイプを設定します。サーバー側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が “Raw public key” 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **buf** 証明書タイプが格納されるバッファ。
- **len** buf サイズ (バイト単位)(含まれる証明書タイプの数と同じ)。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 証明書タイプが正常に設定された場合。
- BAD_FUNC_ARG ctx に NULL が渡された場合、証明書タイプとして不正な値が指定された場合、buf サイズが MAX_CLIENT_CERT_TYPE_CNT を超えた場合、または buf 内に重複する値が見つかった場合。

C.52.2.475 function wolfSSL_set_server_cert_type

```
int wolfSSL_set_server_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

この関数がサーバー側で呼び出された場合、ピアに送信できる証明書タイプを設定します。クライアント側で呼び出された場合、ピアから受け入れ可能な証明書タイプを設定します。優先順位の高い順にバッファに証明書タイプを格納します。設定をデフォルトにリセットするには、buf に NULL を渡すか、len に 0 を渡します。デフォルトでは、証明書タイプは X509 のみです。両側が “Raw public key” 証明書を送信または受け入れることを意図している場合、WOLFSSL_CERT_TYPE_RPK をバッファに含めて設定する必要があります。

Parameters:

- **ctx** WOLFSSL_CTX オブジェクトポインタ。
- **buf** 証明書タイプが格納されるバッファ。
- **len** buf サイズ (バイト単位)(含まれる証明書タイプの数と同じ)。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 証明書タイプが正常に設定された場合。
- BAD_FUNC_ARG ctx に NULL が渡された場合、証明書タイプとして不正な値が指定された場合、buf サイズが MAX_SERVER_CERT_TYPE_CNT を超えた場合、または buf 内に重複する値が見つかった場合。

C.52.2.476 function wolfSSL_CTX_clear_group_messages

```
int wolfSSL_CTX_clear_group_messages(
    WOLFSSL_CTX * ctx
)
```

指定された WOLFSSL_CTX コンテキストに対してハンドシェイクメッセージグループ化を無効にします。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ。

See:

- [wolfSSL_CTX_set_group_messages](#)
- [wolfSSL_set_group_messages](#)
- [wolfSSL_clear_group_messages](#)

Return:

- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG ctx が NULL の場合。

この関数は、指定されたコンテキストから作成されたすべての SSL オブジェクトに対してハンドシェイクメッセージグループ化をオフにします。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());  
wolfSSL_CTX_clear_group_messages(ctx);
```

C.52.2.477 function wolfSSL_clear_group_messages

```
int wolfSSL_clear_group_messages(  
    WOLFSSL * ssl  
)
```

指定された WOLFSSL オブジェクトに対してハンドシェイクメッセージグループ化を無効にします。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ。

See:

- [wolfSSL_set_group_messages](#)
- [wolfSSL_CTX_set_group_messages](#)
- [wolfSSL_CTX_clear_group_messages](#)

Return:

- WOLFSSL_SUCCESS 成功時。
- BAD_FUNC_ARG ssl が NULL の場合。

この関数は、指定された SSL オブジェクトに対してハンドシェイクメッセージグループ化をオフにします。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);  
wolfSSL_clear_group_messages(ssl);
```

C.52.2.478 function wolfSSL_get_negotiated_client_cert_type

```
int wolfSSL_get_negotiated_client_cert_type(  
    WOLFSSL * ssl,  
    int * tp  
)
```

この関数は、ClientHello と ServerHello で行われたクライアント証明書タイプネゴシエーションの結果を返します。ネゴシエーションが発生しなかった場合、戻り値として WOLFSSL_SUCCESS が返され、証明書タイプとして WOLFSSL_CERT_TYPE_UNKNOWN が返されます。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **tp** 証明書タイプが返されるバッファ。次の 3 つの証明書タイプのいずれかが返されます: WOLFSSL_CERT_TYPE_RPK、WOLFSSL_CERT_TYPE_X509、または 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 ネゴシエートされた証明書タイプを取得できた場合。
- BAD_FUNC_ARG ctx または tp に NULL が渡された場合。

Example

```
int ret;
WOLFSSL* ssl;
int tp;
...

ret = wolfSSL_get_negotiated_client_cert_type(ssl, &tp);
```

C.52.2.479 function wolfSSL_get_negotiated_server_cert_type

```
int wolfSSL_get_negotiated_server_cert_type(
    WOLFSSL * ssl,
    int * tp
)
```

この関数は、ClientHello と ServerHello で行われたサーバ証明書タイプネゴシエーションの結果を返します。ネゴシエーションが発生しなかった場合、戻り値として WOLFSSL_SUCCESS が返され、証明書タイプとして WOLFSSL_CERT_TYPE_UNKNOWN が返されます。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **tp** 証明書タイプが返されるバッファ。次の 3 つの証明書タイプのいずれかが返されます: WOLFSSL_CERT_TYPE_RPK、WOLFSSL_CERT_TYPE_X509、または 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 ネゴシエートされた証明書タイプを取得できた場合。
- BAD_FUNC_ARG ctx または tp に NULL が渡された場合。

C.52.2.480 function wolfSSL_dtls_cid_use

```
int wolfSSL_dtls_cid_use(
    WOLFSSL * ssl
)
```

SSL オブジェクトに対して ConnectionID 拡張の使用を有効にします。RFC 9146 および RFC 9147 を参照してください。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。

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 成功時、それ以外の場合はエラーコード。

C.52.2.481 `function wolfSSL_dtls_cid_is_enabled`

```
int wolfSSL_dtls_cid_is_enabled(  
    WOLFSSL * ssl  
)
```

ハンドシェイク完了後に呼び出された場合、SSL オブジェクトに対して ConnectionID が正常にネゴシエートされたかどうかを確認します。RFC 9146 および RFC 9147 を参照してください。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。

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 ConnectionID が正しくネゴシエートされた場合、それ以外は 0。

C.52.2.482 `function wolfSSL_dtls_cid_set`

```
int wolfSSL_dtls_cid_set(  
    WOLFSSL * ssl,  
    unsigned char * cid,  
    unsigned int size  
)
```

この接続でレコードを送信する際に相手ピアが使用する ConnectionID を設定します。RFC 9146 および RFC 9147 を参照してください。ConnectionID は最大 DTLS_CID_MAX_SIZE(調整可能なコンパイル時定義)である必要があり、255 バイトを超えることはできません。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **cid** 使用する ConnectionID。
- **size** 提供された ConnectionID のサイズ。

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 ConnectionID が正しく設定された場合、それ以外の場合はエラーコード。

C.52.2.483 function wolfSSL_dtls_cid_get_rx_size

```
int wolfSSL_dtls_cid_get_rx_size(  
    WOLFSSL * ssl,  
    unsigned int * size  
)
```

この接続でレコードを送信する際に相手ピアが使用する ConnectionID のサイズを取得します。RFC 9146 および RFC 9147 を参照してください。サイズはパラメータ size に格納されます。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **size** サイズが格納される符号なし int 型へのポインタ。

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 ConnectionID が正しくネゴシエートされた場合、それ以外の場合はエラーコード。

C.52.2.484 function wolfSSL_dtls_cid_get_rx

```
int wolfSSL_dtls_cid_get_rx(  
    WOLFSSL * ssl,  
    unsigned char * buffer,  
    unsigned int bufferSize  
)
```

この接続でレコードを送信する際に相手ピアが使用する ConnectionID を、パラメータ buffer が指すバッファにコピーします。RFC 9146 および RFC 9147 を参照してください。bufferSz でバッファ内の利用可能なスペースを提供する必要があります。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **buffer** ConnectionID がコピーされるバッファ。
- **bufferSz** buffer 内の利用可能なスペース。

See:

- 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

Return: WOLFSSL_SUCCESS ConnectionID が正しくコピーされた場合、それ以外の場合はエラーコード。

C.52.2.485 function wolfSSL_dtls_cid_get0_rx

```
int wolfSSL_dtls_cid_get0_rx(  
    WOLFSSL * ssl,
```

```
    unsigned char ** cid  
)
```

相手ピアが使用する ConnectionID を取得します。RFC 9146 および RFC 9147 を参照してください。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **cid** CID を保持する内部メモリに設定されるポインタ。

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 ConnectionID が cid に正しく設定された場合。

C.52.2.486 function `wolfSSL_dtls_cid_get_tx_size`

```
int wolfSSL_dtls_cid_get_tx_size(  
    WOLFSSL * ssl,  
    unsigned int * size  
)
```

この接続でレコードを送信する際に使用する ConnectionID のサイズを取得します。RFC 9146 および RFC 9147 を参照してください。サイズはパラメータ size に格納されます。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **size** サイズが格納される符号なし int 型へのポインタ。

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 ConnectionID サイズが正しく格納された場合、それ以外の場合はエラーコード。

C.52.2.487 function `wolfSSL_dtls_cid_get_tx`

```
int wolfSSL_dtls_cid_get_tx(  
    WOLFSSL * ssl,  
    unsigned char * buffer,  
    unsigned int bufferSize  
)
```

この接続でレコードを送信する際に使用する ConnectionID を、パラメータ buffer が指すバッファにコピーします。RFC 9146 および RFC 9147 を参照してください。bufferSz で利用可能なサイズを提供する必要があります。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **buffer** ConnectionID がコピーされるバッファ。
- **bufferSz** buffer 内の利用可能なスペース。

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 ConnectionID が正しくコピーされた場合、それ以外の場合はエラーコード。

C.52.2.488 function wolfSSL_dtls_cid_get0_tx

```
int wolfSSL_dtls_cid_get0_tx(  
    WOLFSSL * ssl,  
    unsigned char ** cid  
)
```

この接続でレコードを送信する際に使用する ConnectionID を取得します。RFC 9146 および RFC 9147 を参照してください。

Parameters:

- **ssl** WOLFSSL オブジェクトポインタ。
- **cid** CID を保持する内部メモリに設定されるポインタ。

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 ConnectionID が正しく取得された場合、それ以外の場合はエラーコード。

C.52.2.489 function wolfSSL_dtls_cid_parse

```
const unsigned char * wolfSSL_dtls_cid_parse(  
    const unsigned char * msg,  
    unsigned int msgSz,  
    unsigned int cidSz  
)
```

レコードデータグラム/メッセージから ConnectionID を抽出します。RFC 9146 および RFC 9147 を参照してください。

Parameters:

- **msg** ネットワークから読み取られたデータグラムを保持するバッファ。
- **msgSz** msg のサイズ (バイト単位)。
- **cid** msg バッファ内の CID の開始位置へのポインタ。

- **cidSz** 期待される CID のサイズ。レコード層には CID サイズフィールドがないため、CID のサイズを事前に知っている必要があります。すべての接続に定数 CID を使用することを推奨します。

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.52.2.490 function wolfSSL_CTX_set_client_CA_list

```
void wolfSSL_CTX_set_client_CA_list(
    WOLFSSL_CTX * ctx,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

サーバ側では、この関数は証明書要求でクライアントに送信される CA 名のリストを設定します。これは、サーバがサポートする CA のヒントとして機能します。

Parameters:

- **ctx** wolfSSL コンテキストへのポインタ。
- **names** 設定される名前のリスト。

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](#)

クライアント側では、この関数は効果がありません。

C.52.2.491 function wolfSSL_CTX_get_client_CA_list

```
WOLFSSL_STACK * wolfSSL_CTX_get_client_CA_list(
    const WOLFSSL_CTX * ctx
)
```

これは、wolfSSL_CTX_set_client_CA_list を介して以前に設定されたリストを取得します。リストが設定されていない場合は NULL を返します。

Parameters:

- **ctx** wolfSSL コンテキストへのポインタ。

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: CA 名を含む WOLFSSL_X509_NAME のスタック。

C.52.2.492 function `wolfSSL_set_client_CA_list`

```
void wolfSSL_set_client_CA_list(
    WOLFSSL * ssl,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

`wolfSSL_CTX_set_client_CA_list` と同じですが、セッション固有です。CA リストがコンテキストとセッションの両方に設定されている場合、セッション上のリストが使用されます。

Parameters:

- **ssl** WOLFSSL オブジェクトへのポインタ。
- **names** 設定する名前のリスト。

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.52.2.493 function `wolfSSL_get_client_CA_list`

```
WOLFSSL_STACK * wolfSSL_get_client_CA_list(
    const WOLFSSL * ssl
)
```

サーバ側では、`wolfSSL_set_client_CA_list` を介して以前に設定されたリストを取得します。何も設定されていない場合は、`wolfSSL_CTX_set_client_CA_list` を介して以前に設定されたリストを返します。リストが全く設定されていない場合は、NULL を返します。

Parameters:

- **ssl** WOLFSSL オブジェクトへのポインタ。

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: CA 名を含む WOLFSSL_X509_NAME のスタック。

クライアント側では、サーバから受信したリストを取得します。何も受信していない場合は NULL を返します。wolfSSL_CTX_set_cert_cb を使用して、サーバから証明書要求を受信したときに証明書を動的にロードするコールバックを登録できます。

C.52.2.494 function wolfSSL_CTX_set0_CA_list

```
void wolfSSL_CTX_set0_CA_list(  
    WOLFSSL_CTX * ctx,  
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names  
)
```

この関数は、ピアの認証でサポートされている CA のヒントとしてピアに送信される CA 名のリストを設定します。

Parameters:

- **ctx** wolfSSL コンテキストへのポインタ。
- **names** 設定する名前のリスト。

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](#)

TLS >= 1.3 では、これはクライアントとサーバ間の両方向でサポートされています。サーバ側では、CA 名は CertificateRequest の一部として送信されるため、この関数は *_set_client_CA_list と同等です。クライアント側では、これらは ClientHello の一部として送信されます。

TLS < 1.3 では、クライアントからサーバへの CA 名の送信はサポートされていないため、この関数は wolfSSL_CTX_set_client_CA_list と同等です。

_set_client_CA_list と *_set0_CA_list を介して設定されたリストは内部的に別々であることに注意してください。つまり、*_get_client_CA_list を呼び出しても *_set0_CA_list を介して設定されたリストは取得されず、その逆も同様です。両方が設定されている場合、サーバはクライアントに CA 名を送信する際に *_set0_CA_list を無視します。

C.52.2.495 function wolfSSL_CTX_get0_CA_list

```
WOLFSSL_STACK * wolfSSL_CTX_get0_CA_list(  
    const WOLFSSL_CTX * ctx  
)
```

これは、wolfSSL_CTX_set0_CA_list を介して以前に設定されたリストを取得します。リストが設定されていない場合は NULL を返します。

Parameters:

- **ctx** wolfSSL コンテキストへのポインタ。

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: CA 名を含む WOLFSSL_X509_NAME のスタック。

C.52.2.496 function `wolfSSL_set0_CA_list`

```
void wolfSSL_set0_CA_list(
    WOLFSSL * ssl,
    WOLF_STACK_OF(WOLFSSL_X509_NAME) * names
)
```

`wolfSSL_CTX_set0_CA_list` と同じですが、セッション固有です。CA リストがコンテキストとセッションの両方に設定されている場合、セッション上のリストが使用されます。

Parameters:

- **ssl** WOLFSSL オブジェクトへのポインタ。
- **names** 設定する名前のリスト。

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.52.2.497 function `wolfSSL_get0_CA_list`

```
WOLFSSL_STACK * wolfSSL_get0_CA_list(
    const WOLFSSL * ssl
)
```

これは、`wolfSSL_set0_CA_list` を介して以前に設定されたリストを取得します。何も設定されていない場合は、`wolfSSL_CTX_set0_CA_list` を介して以前に設定されたリストを返します。リストが全く設定されていない場合は、NULL を返します。

Parameters:

- **ssl** WOLFSSL オブジェクトへのポインタ。

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: CA 名を含む WOLFSSL_X509_NAME のスタック。

C.52.2.498 function wolfSSL_get0_peer_CA_list

```
WOLFSSL_STACK * wolfSSL_get0_peer_CA_list(  
    const WOLFSSL * ssl  
)
```

これは、ピアから受信した CA リストを返します。

Parameters:

- **ssl** WOLFSSL オブジェクトへのポインタ。

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: CA 名を含む WOLFSSL_X509_NAME のスタック。

クライアント側では、これはサーバが CertificateRequest で送信したリストであり、この関数は `wolfSSL_get_client_CA_list` と同等です。

サーバ側では、これは TLS >= 1.3 でクライアントが ClientHello メッセージで送信したリストです。TLS < 1.3 では、この関数はサーバ側で常に NULL を返します。

`wolfSSL_CTX_set_cert_cb` を使用して、ピアから CA リストを受信したときに証明書を動的にロードするコールバックを登録できます。

C.52.2.499 function wolfSSL_CTX_set_cert_cb

```
void wolfSSL_CTX_set_cert_cb(  
    WOLFSSL_CTX * ctx,  
    int (*)(WOLFSSL *, void *) cb,  
    void * arg  
)
```

この関数は、証明書が使用される直前に呼び出されるコールバックを設定し、アプリケーションが証明書を検査、設定、またはクリアできるようにします。例えば、ピアから送信された CA リストに反応することができます。

Parameters:

- **ctx** wolfSSL コンテキストへのポインタ。
- **cb** コールバックへの関数ポインタ。
- **arg** コールバックに渡されるポインタ。

See:

- [wolfSSL_get0_peer_CA_list](#)
- [wolfSSL_get_client_CA_list](#)

C.52.2.500 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
)
```

この関数は、クライアントが提供する暗号スイートと署名アルゴリズムの生のリストを返します。リストは、wolfSSL_CTX_set_cert_cb() で設定されたコールバック内でのみ保存され、返されます。これは、利用可能な暗号スイートと署名アルゴリズムに基づいて証明書と鍵を動的にロードできるようにするのに便利です。

Parameters:

- **ssl** リストを抽出する WOLFSSL オブジェクト。
- **suites** クライアント暗号スイートの生のフィルタリングされていないリスト。利用可能なスイートがない場合は NULL になることがあります。
- **suiteSz** suites のサイズ (バイト単位)。
- **hashSigAlgo** クライアント署名アルゴリズムの生のフィルタリングされていないリスト。提供されない場合は NULL になることがあります。
- **hashSigAlgoSz** hashSigAlgo のサイズ (バイト単位)。

See:

- wolfSSL_get_ciphersuite_info
- wolfSSL_get_sigalg_info

Return:

- WOLFSSL_SUCCESS スイートが利用可能な場合。
- WOLFSSL_FAILURE スイートが利用できない場合。

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);

    // 暗号スイートと sigalgs に基づいてロードする証明書を選択
}

WOLFSSL* ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
wolfSSL_CTX_set_cert_cb(ctx, certCB, NULL);
```

C.52.2.501 function wolfSSL_get_ciphersuite_info

```
WOLFSSL_CIPHERSUITE_INFO wolfSSL_get_ciphersuite_info(
    byte first,
    byte second
)
```

これは、生の暗号スイートバイトから直接暗号スイートに関する情報を返します。

Parameters:

- **first** 暗号スイートの最初のバイト。
- **second** 暗号スイートの 2 番目のバイト。

See:

- [wolfSSL_get_client_suites_sigalgs](#)
- [wolfSSL_get_sigalg_info](#)

Return: WOLFSSL_CIPHERSUITE_INFO 暗号スイートで使用する認証のタイプに関する情報を含む構造体。

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.52.2.502 function wolfSSL_get_sigalg_info

```
int wolfSSL_get_sigalg_info(
    byte first,
    byte second,
    int * hashAlgo,
    int * sigAlgo
)
```

これは、生の暗号スイートバイトから直接ハッシュおよび署名アルゴリズムに関する情報を返します。

Parameters:

- **first** ハッシュおよび署名アルゴリズムの最初のバイト。
- **second** ハッシュおよび署名アルゴリズムの 2 番目のバイト。
- **hashAlgo** MAC アルゴリズムの enum wc_HashType。
- **sigAlgo** 認証アルゴリズムの enum Key_Sum。

See:

- [wolfSSL_get_client_suites_sigalgs](#)
- [wolfSSL_get_ciphersuite_info](#)

Return:

- 0 情報が正しく設定された場合。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合、またはバイトが認識される sigalg スイートでない場合。

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.52.3 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_read(WOLFSSL* ssl, void* data, int sz);

int wolfSSL_peek(WOLFSSL* ssl, void* data, int sz);

int wolfSSL_accept(WOLFSSL* ssl);

int wolfDTLS_accept_stateless(WOLFSSL* ssl);

void wolfSSL_CTX_free(WOLFSSL_CTX* ctx);

void wolfSSL_free(WOLFSSL* ssl);

int wolfSSL_shutdown(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_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);

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_memsz(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_get_timeout(WOLFSSL* ssl);

int wolfSSL_dtls_retransmit(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);

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_online(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(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);

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_CTX_set_tlsext_status_arg(WOLFSSL_CTX* ctx, void* arg);

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);

```

```
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_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);

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); /* ビルドに基づくオブジェクトサイズ */
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_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_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_TLShv13(WOLFSSL* ssl);

wolfSSL_accept_TLShv13(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_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);

int wolfSSL_UseKeyShare(WOLFSSL* ssl, word16 group);

int wolfSSL_NoKeyShares(WOLFSSL* ssl);

WOLFSSL_METHOD* wolfTLShv1_3_server_method_ex(void* heap);

WOLFSSL_METHOD* wolfTLShv1_3_client_method_ex(void* heap);

WOLFSSL_METHOD* wolfTLShv1_3_server_method(void);

WOLFSSL_METHOD* wolfTLShv1_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_CRYPT0_get_ex_new_index(int class_index, long arg1, void *argp,
    WOLFSSL_CRYPT0_EX_new* new_func,
    WOLFSSL_CRYPT0_EX_dup* dup_func,
    WOLFSSL_CRYPT0_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_set_group_messages(WOLFSSL_CTX* ctx);

int wolfSSL_CTX_clear_group_messages(WOLFSSL_CTX* ctx);

int wolfSSL_set_group_messages(WOLFSSL* ssl);

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_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);

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);

void wolfSSL_CTX_set_cert_cb(WOLFSSL_CTX* ctx,
    int (*cb)(WOLFSSL *, void *), void *arg);

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);
```

C.53 dox_comments/header_files-ja/tfm.h

C.53.1 Functions

	Name
word32	CheckRunTimeFastMath (void) この関数は、整数の最大サイズに関するランタイム fastmath 設定をチェックします。ユーザーが wolfCrypt ライブラリを独立して使用している場合に重要です。数学が正しく動作するためには、各ライブラリの FP_SIZE が一致している必要があります。このチェックは CheckFastMathSettings() として定義されており、CheckRunTimeFastMath と FP_SIZE を単純に比較し、不一致の場合は 0 を、一致する場合は 1 を返します。

C.53.2 Functions Documentation

C.53.2.1 function CheckRunTimeFastMath

```
word32 CheckRunTimeFastMath(
    void
)
```

この関数は、整数の最大サイズに関するランタイム fastmath 設定をチェックします。ユーザーが wolfCrypt ライブラリを独立して使用している場合に重要です。数学が正しく動作するためには、各ライブラリの FP_SIZE が一致している必要があります。このチェックは CheckFastMathSettings() として定義されており、CheckRunTimeFastMath と FP_SIZE を単純に比較し、不一致の場合は 0 を、一致する場合は 1 を返します。

Parameters:

- **none** パラメータなし。

See: [CheckRunTimeSettings](#)

Return: FP_SIZE 数学ライブラリで使用可能な最大サイズに対応する FP_SIZE を返します。

Example

```
if (CheckFastMathSettings() != 1) {
    return err_sys("Build vs. runtime fastmath FP_MAX_BITS mismatch\n");
}
// これはプリプロセッサによって次のように変換されます:
// if ( (CheckRunTimeFastMath() == FP_SIZE) != 1) {
// そして fastmath 設定がコンパイル時の設定と一致することを確認します
```

C.53.3 Source code

```
word32 CheckRunTimeFastMath(void);
```

C.54 dox_comments/header_files-ja/types.h

C.54.1 Functions

	Name
void *	XMALLOC(size_t n, void * heap, int type) これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: 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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: #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)) これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えま <pre>す: #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))</pre>

	Name
void *	XREALLOC(void * p, size_t n, void * heap, int type) これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: 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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: <pre>#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))</pre> これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えます: <pre>#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))</pre>

	Name
void	XFREE(void * p, void * heap, int type) これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: 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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: #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)) これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えられます: #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) この関数は、コンパイル時のクラス設定をチェックします。ユーザーが wolfCrypt ライブラリを独立して使用している場合に重要です。数学が正しく動作するためには、ライブラリ間で設定が一致している必要があります。このチェックは CheckCtcSettings() として定義されており、CheckRunTimeSettings と CTC_SETTINGS を単純に比較し、不一致の場合は 0 を、一致する場合は 1 を返します。

C.54.2 Functions Documentation

C.54.2.1 function XMALLOC

```
void * XMALLOC(
    size_t n,
    void * heap,
    int type
)
```

これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: 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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: #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)) これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えます: #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** 割り当てるメモリのサイズ
- **h** (カスタム XMALLOC 関数で使用する) 使用するヒープへのポインタ
- **t** ユーザーヒント用のメモリ割り当てタイプ。types.h の列挙型を参照

See:

- [wolfSSL_Malloc](#)
- [wolfSSL_Realloc](#)
- [wolfSSL_Free](#)
- [wolfSSL_SetAllocators](#)

Return:

- pointer 成功時に割り当てられたメモリへのポインタを返します
- NULL 失敗時

Example

```
int* tenInts = XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // スペース割り当てエラー
    return MEMORY_E;
}
```

C.54.2.2 function XREALLOC

```
void * XREALLOC(
    void * p,
    size_t n,
    void * heap,
    int type
)
```

これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: 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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: #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)) これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えます: #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** 再割り当てするアドレスへのポインタ
- **n** 割り当てるメモリのサイズ
- **h** (カスタム XREALLOC 関数で使用) 使用するヒープへのポインタ
- **t** ユーザーヒント用のメモリ割り当てタイプ。types.h の列挙型を参照

See:

- [wolfSSL_Malloc](#)
- [wolfSSL_Realloc](#)
- [wolfSSL_Free](#)
- [wolfSSL_SetAllocators](#)

Return:

- 成功時に割り当てられたメモリへのポインタを返します
- NULL 失敗時

Example

```
int* tenInts = (int*)XMAALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
int* twentyInts = (int*)XREALLOC(tenInts, sizeof(int)*20, NULL,
    DYNAMIC_TYPE_TMP_BUFFER);
```

C.54.2.3 function XFREE

```
void XFREE(
    void * p,
    void * heap,
    int type
)
```

これは実際には関数ではなく、プリプロセッサマクロです。ユーザーが標準の C メモリ関数の代わりに独自の malloc、realloc、free 関数を置き換えることを可能にします。外部メモリ関数を使用するには、XMAALLOC_USER を定義します。これにより、メモリ関数は次の形式の外部関数に置き換えられます: extern void XMAALLOC(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); wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free の代わりに基本的な C メモリ関数を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数は次のように置き換えられます: #define XMAALLOC(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)) これらのオプションのいずれも選択されていない場合、システムはデフォルトで wolfSSL メモリ関数を使用します。ユーザーはコールバックフックを通じてカスタムメモリ関数を設定できます (wolfSSL_Malloc、wolfSSL_Realloc、wolfSSL_Free を参照)。このオプションは、メモリ関数を次のように置き換えます: #define XMAALLOC(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** 解放するアドレスへのポインタ
- **h** (カスタム XFREE 関数で使用) 使用するヒープへのポインタ
- **t** ユーザーヒント用のメモリ割り当てタイプ。types.h の列挙型を参照

See:

- [wolfSSL_Malloc](#)
- [wolfSSL_Realloc](#)
- [wolfSSL_Free](#)
- [wolfSSL_SetAllocators](#)

Return: none 戻り値なし。

Example

```
int* tenInts = XMALLOC(sizeof(int) * 10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // スペース割り当てエラー
    return MEMORY_E;
}
```

C.54.2.4 function CheckRunTimeSettings

```
word32 CheckRunTimeSettings(
    void
)
```

この関数は、コンパイル時のクラス設定をチェックします。ユーザーが wolfCrypt ライブラリを独立して使用している場合に重要です。数学が正しく動作するためには、ライブラリ間で設定が一致している必要があります。このチェックは CheckCtcSettings() として定義されており、CheckRunTimeSettings と CTC_SETTINGS を単純に比較し、不一致の場合は 0 を、一致する場合は 1 を返します。

Parameters:

- **none** パラメータなし。

See: [CheckRunTimeFastMath](#)

Return: settings ランタイム CTC_SETTINGS(コンパイル時設定) を返します

Example

```
if (CheckCtcSettings() != 1) {
    return err_sys("Build vs. runtime math mismatch\n");
}
// これはプリプロセッサによって次のように変換されます:
// if ( (CheckCtcSettings() == CTC_SETTINGS) != 1) {
// そしてコンパイル時のクラス設定が現在の設定と一致するかどうかを比較します
```

C.54.3 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);
```

C.55 dox_comments/header_files-ja/wc_encrypt.h

C.55.1 Functions

	Name
int	wc_AesCbcDecryptWithKey (byte * out, const byte * in, word32 inSz, const byte * key, word32 keySz, const byte * iv) 入力バッファ in から暗号を復号し、結果の平文を AES を使用した暗号ブロック連鎖を使用して出力バッファ out に格納します。この関数は、AES 構造体を初期化する必要はありません。代わりに、キーと iv(初期化ベクトル)を受け取り、これらを使用して AES オブジェクトを初期化し、暗号文を復号します。
int	wc_Des_CbcDecryptWithKey (byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。この関数は wc_Des_CbcDecrypt の代替で、ユーザーが Des 構造体を直接インスタンス化せずにメッセージを復号できるようにします。
int	wc_Des_CbcEncryptWithKey (byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は、入力平文 in を暗号化し、結果の暗号文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。この関数は wc_Des_CbcEncrypt の代替で、ユーザーが Des 構造体を直接インスタンス化せずにメッセージを暗号化できるようにします。
int	wc_Des3_CbcEncryptWithKey (byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は、入力平文 in を暗号化し、結果の暗号文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル DES(3DES) 暗号化を使用します。この関数は wc_Des3_CbcEncrypt の代替で、ユーザーが Des3 構造体を直接インスタンス化せずにメッセージを暗号化できるようにします。
int	wc_Des3_CbcDecryptWithKey (byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。この関数は wc_Des3_CbcDecrypt の代替で、ユーザーが Des3 構造体を直接インスタンス化せずにメッセージを復号できるようにします。

C.55.2 Functions Documentation

C.55.2.1 function wc_AesCbcDecryptWithKey

```
int wc_AesCbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 inSz,
```

```

    const byte * key,
    word32 keySz,
    const byte * iv
)

```

入力バッファ in から暗号を復号し、結果の平文を AES を使用した暗号ブロック連鎖を使用して出力バッファ out に格納します。この関数は、AES 構造体を初期化する必要はありません。代わりに、キーと iv(初期化ベクトル)を受け取り、これらを使用して AES オブジェクトを初期化し、暗号文を復号します。

Parameters:

- **out** 復号されたメッセージの平文を格納する出力バッファへのポインタ
- **in** 復号する暗号文を含む入力バッファへのポインタ
- **inSz** 入力メッセージのサイズ
- **key** 復号用の 16、24、または 32 バイトの秘密鍵
- **keySz** 復号に使用するキーのサイズ

See:

- [wc_AesSetKey](#)
- [wc_AesSetIV](#)
- [wc_AesCbcEncrypt](#)
- [wc_AesCbcDecrypt](#)

Return:

- 0 メッセージの復号に成功した場合
- BAD_ALIGN_E ブロックアライメントエラーが発生した場合に返されます
- BAD_FUNC_ARG AesSetIV 中にキー長が無効または AES オブジェクトが null の場合に返されます
- MEMORY_E WOLFSSL_SMALL_STACK が有効で、XMALLOC が AES オブジェクトのインスタンス化に失敗した場合に返されます。

Example

```

int ret = 0;
byte key[] = { 16、24、または32バイトのキー };
byte iv[] = { 16バイトの iv };
byte cipher[AES_BLOCK_SIZE * n]; //n は正の整数で、cipher を 16 バイトの倍数にする
// cipher を暗号文で埋める
byte plain [AES_BLOCK_SIZE * n];
if ((ret = wc_AesCbcDecryptWithKey(plain, cipher, AES_BLOCK_SIZE, key,
AES_BLOCK_SIZE, iv)) != 0 ) {
// 復号エラー
}

```

C.55.2.2 function wc_Des_CbcDecryptWithKey

```

int wc_Des_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)

```

この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。この関数は wc_Des_CbcDecrypt の代替で、ユーザーが Des 構造体を直接インスタンス化せずにメッセージを復号できるようにします。

Parameters:

- **out** 復号された平文を格納するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号する暗号文の長さ
- **key** 復号に使用する 8 バイトのキーを含むバッファへのポインタ
- **iv** 復号に使用する 8 バイトの iv を含むバッファへのポインタ。iv が提供されない場合、iv はデフォルトで 0 になります

See: [wc_Des_CbcDecrypt](#)

Return:

- 0 指定された暗号文の復号に成功した場合に返されます
- MEMORY_E Des 構造体用のスペース割り当て中にエラーが発生した場合に返されます

3

Example

```
int ret;
byte key[] = { // 8 バイトのキーで初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };

byte cipher[] = { // 暗号文で初期化 };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecryptWithKey(decoded, cipher, sizeof(cipher), key,
iv) != 0 ) {
    // メッセージの復号エラー
}
```

C.55.2.3 function wc_Des_CbcEncryptWithKey

```
int wc_Des_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

この関数は、入力平文 in を暗号化し、結果の暗号文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードの DES 暗号化を使用します。この関数は wc_Des_CbcEncrypt の代替で、ユーザーが Des 構造体を直接インスタンス化せずにメッセージを暗号化できるようにします。

Parameters:

- **out** 最終的に暗号化されたデータ
- **in** 暗号化されるデータ。Des ブロックサイズにパディングされている必要があります。
- **sz** 入力バッファのサイズ。
- **key** 暗号化に使用するキーへのポインタ。
- **iv** 初期化ベクトル

See:

- [wc_Des_CbcDecryptWithKey](#)
- [wc_Des_CbcEncrypt](#)

Return:

- 0 データの暗号化に成功した後に返されます。
- MEMORY_E Des 構造体用のメモリ割り当て中にエラーが発生した場合に返されます。

- <0 暗号化中の任意のエラーで返されます。

3

Example

```
byte key[] = { // 8 バイトのキーで初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };
byte in[] = { // 平文で初期化 };
byte out[sizeof(in)];
if ( wc_Des_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
    // メッセージの暗号化エラー
}
```

C.55.2.4 function wc_Des3_CbcEncryptWithKey

```
int wc_Des3_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

この関数は、入力平文 in を暗号化し、結果の暗号文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル DES(3DES) 暗号化を使用します。この関数は wc_Des3_CbcEncrypt の代替で、ユーザーが Des3 構造体を直接インスタンス化せずにメッセージを暗号化できるようにします。

Parameters:

- **out** 最終的に暗号化されたデータ
- **in** 暗号化されるデータ。Des ブロックサイズにパディングされている必要があります。
- **sz** 入力バッファのサイズ。
- **key** 暗号化に使用するキーへのポインタ。
- **iv** 初期化ベクトル

See:

- [wc_Des3_CbcDecryptWithKey](#)
- [wc_Des_CbcEncryptWithKey](#)
- [wc_Des_CbcDecryptWithKey](#)

Return:

- 0 データの暗号化に成功した後に返されます。
- MEMORY_E Des 構造体用のメモリ割り当て中にエラーが発生した場合に返されます。
- <0 暗号化中の任意のエラーで返されます。

3

Example

```
byte key[] = { // 8 バイトのキーで初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };

byte in[] = { // 平文で初期化 };
byte out[sizeof(in)];

if ( wc_Des3_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
}
```



```
// メッセージの暗号化エラー
}
```

C.55.2.5 function wc_Des3_CbcDecryptWithKey

```
int wc_Des3_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

この関数は、入力暗号文 in を復号し、結果の平文を出力バッファ out に格納します。暗号ブロック連鎖 (CBC) モードのトリプル Des(3DES) 暗号化を使用します。この関数は wc_Des3_CbcDecrypt の代替で、ユーザーが Des3 構造体を直接インスタンス化せずにメッセージを復号できるようにします。

Parameters:

- **out** 復号された平文を格納するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号する暗号文の長さ
- **key** 復号に使用する 24 バイトのキーを含むバッファへのポインタ
- **iv** 復号に使用する 8 バイトの iv を含むバッファへのポインタ。iv が提供されない場合、iv はデフォルトで 0 になります

See: [wc_Des3_CbcDecrypt](#)

Return:

- 0 指定された暗号文の復号に成功した場合に返されます
- MEMORY_E Des 構造体用のスペース割り当て中にエラーが発生した場合に返されます

3

Example

```
int ret;
byte key[] = { // 24 バイトのキーで初期化 };
byte iv[] = { // 8 バイトの iv で初期化 };

byte cipher[] = { // 暗号文で初期化 };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecryptWithKey(decoded, cipher, sizeof(cipher),
key, iv) != 0) {
    // メッセージの復号エラー
}
```

C.55.3 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);

```

C.56 dox_comments/header_files-ja/wc_port.h

C.56.1 Functions

Name
int wolfCrypt_Init (void)wolfCrypt で使用されるリソースを初期化するために使用されます。
int wolfCrypt_Cleanup (void)wolfCrypt で使用されるリソースをクリーンアップするために使用されます。

C.56.2 Functions Documentation

C.56.2.1 function wolfCrypt_Init

```

int wolfCrypt_Init(
    void
)

```

wolfCrypt で使用されるリソースを初期化するために使用されます。

Parameters:

- **none** パラメータなし。

See: [wolfCrypt_Cleanup](#)

Return:

- 0 成功時。
- <0 リソースの初期化に失敗した場合。

Example

```

...
if (wolfCrypt_Init() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Init call");
}

```

C.56.2.2 function wolfCrypt_Cleanup

```

int wolfCrypt_Cleanup(
    void
)

```

wolfCrypt で使用されるリソースをクリーンアップするために使用されます。

Parameters:

- **none** パラメータなし。

See: [wolfCrypt_Init](#)

Return:

- 0 成功時。
- <0 リソースのクリーンアップに失敗した場合。

Example

```
...
if (wolfCrypt_Cleanup() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Cleanup call");
}
```

C.56.3 Source code

```
int wolfCrypt_Init(void);
int wolfCrypt_Cleanup(void);
```

C.57 dox_comments/header_files-ja/wolfio.h

C.57.1 Functions

	Name
int	EmbedReceive (WOLFSSL * ssl, char * buf, int sz, void * ctx) この関数は受信埋め込みコールバックです。
int	EmbedSend (WOLFSSL * ssl, char * buf, int sz, void * ctx) この関数は送信埋め込みコールバックです。
int	EmbedReceiveFrom (WOLFSSL * ssl, char * buf, int sz, void * ctx) この関数は受信埋め込みコールバックです。
int	EmbedSendTo (WOLFSSL * ssl, char * buf, int sz, void * ctx) この関数は送信埋め込みコールバックです。
int	EmbedGenerateCookie (WOLFSSL * ssl, byte * buf, int sz, void * ctx) この関数は DTLS Generate Cookie コールバックです。
void	EmbedOcspRespFree (void * ctx, byte * resp) この関数は応答バッファを解放します。

	Name
void	wolfSSL_CTX_SetIORecv (WOLFSSL_CTX * ctx, CallbackIORecv CBIORcv) この関数は、wolfSSL が入力データを取得するための受信コールバックを登録します。デフォルトでは、wolfSSL はシステムの TCP recv() 関数を使用するコールバックとして EmbedReceive() を使用します。ユーザーは、メモリ、他のネットワークモジュール、または任意の場所から入力を取得する関数を登録できます。関数の動作方法とエラーコードについては、src/io.c の EmbedReceive() 関数をガイドとして参照してください。特に、データの準備ができていないノンブロッキング受信の場合は IO_ERR_WANT_READ を返す必要があります。
void	wolfSSL_SetIOReadCtx (WOLFSSL * ssl, void * ctx) この関数は、SSL セッションの受信コールバック関数のコンテキストを登録します。デフォルトでは、wolfSSL がシステムの TCP ライブラリを使用している場合、wolfSSL は wolfSSL_set_fd() に渡されたファイル記述子をコンテキストとして設定します。独自の受信コールバックを登録した場合、セッションの特定のコンテキストを設定することができます。たとえば、メモリバッファを使用している場合、コンテキストはメモリバッファへのアクセス方法と場所を記述する構造体へのポインタになる可能性があります。
void	wolfSSL_SetIOWriteCtx (WOLFSSL * ssl, void * ctx) この関数は、SSL セッションの送信コールバック関数のコンテキストを登録します。デフォルトでは、wolfSSL がシステムの TCP ライブラリを使用している場合、wolfSSL は wolfSSL_set_fd() に渡されたファイル記述子をコンテキストとして設定します。独自の送信コールバックを登録した場合、セッションの特定のコンテキストを設定することができます。たとえば、メモリバッファを使用している場合、コンテキストはメモリバッファへのアクセス方法と場所を記述する構造体へのポインタになる可能性があります。
void *	wolfSSL_GetIOReadCtx (WOLFSSL * ssl) この関数は WOLFSSL 構造体の IOCB_ReadCtx メンバーを返します。
void *	wolfSSL_GetIOWriteCtx (WOLFSSL * ssl) この関数は WOLFSSL 構造体の IOCB_WriteCtx メンバーを返します。

	Name
void	wolfSSL_SetIOReadFlags (WOLFSSL * ssl, int flags) この関数は、指定された SSL セッションの受信コールバックで使用するフラグを設定します。受信コールバックは、デフォルトの wolfSSL EmbedReceive コールバック、またはユーザーが指定したカスタムコールバックのいずれかです (wolfSSL_CTX_SetIORecv を参照)。デフォルトのフラグ値は、wolfSSL 内部で 0 の値に設定されます。デフォルトの wolfSSL 受信コールバックは、recv() 関数を使用してソケットからデータを受信します。recv() の man ページより：「recv() 関数の flags パラメータは、次の値の 1 つ以上を OR して形成されます：MSG_OOB 帯域外データを処理、MSG_PEEK 受信メッセージを覗き見、MSG_WAITALL 完全なリクエストまたはエラーを待機。MSG_OOB フラグは、通常のデータストリームでは受信されない帯域外データの受信を要求します。一部のプロトコルは、通常のデータキューの先頭に緊急データを配置するため、このフラグはそのようなプロトコルでは使用できません。MSG_PEEK フラグは、受信操作が受信キューの先頭からデータを返すようにしますが、そのデータをキューから削除しません。したがって、後続の受信呼び出しは同じデータを返します。MSG_WAITALL フラグは、完全なリクエストが満たされるまで操作をブロックするよう要求します。ただし、シグナルがキャッチされた場合、エラーまたは切断が発生した場合、または次に受信されるデータが返されたデータとは異なるタイプの場合、呼び出しは要求されたデータよりも少ないデータを返す可能性があります。」

	Name
void	wolfSSL_SetIOWriteFlags (WOLFSSL * ssl, int flags) この関数は、指定された SSL セッションの送信コールバックで使用するフラグを設定します。送信コールバックは、デフォルトの wolfSSL EmbedSend コールバック、またはユーザーが指定したカスタムコールバックのいずれかです (wolfSSL_CTX_SetIOSend を参照)。デフォルトのフラグ値は、wolfSSL 内部で 0 の値に設定されます。デフォルトの wolfSSL 送信コールバックは、send() 関数を使用してソケットからデータを送信します。send() の man ページより：「flags パラメータには、次の 1 つ以上が含まれる場合があります：#define MSG_OOB 0x1 // 帯域外データを処理、#define MSG_DONTROUTE 0x4 // ルーティングをバイパス、直接インターフェースを使用。フラグ MSG_OOB は、この概念をサポートするソケット（例：SOCK_STREAM）で「帯域外」データを送信するために使用されます。基礎となるプロトコルも「帯域外」データをサポートする必要があります。MSG_DONTROUTE は通常、診断またはルーティングプログラムによってのみ使用されます。」
void	wolfSSL_SetIO_NetX (WOLFSSL * ssl, NX_TCP_SOCKET * nxsocket, ULONG waitoption) この関数は、WOLFSSL 構造体内の nxCtx 構造体の nxSocket および nxWait メンバーを設定します。
void	wolfSSL_CTX_SetGenCookie (WOLFSSL_CTX * ctx, CallbackGenCookie cb) この関数は、WOLFSSL_CTX 構造体の CBIOCookie メンバーのコールバックを設定します。CallbackGenCookie 型は関数ポインタで、次のシグネチャを持ちます：int (CallbackGenCookie)(WOLFSSL ssl, unsigned char* buf, int sz, void* ctx);
void *	wolfSSL_GetCookieCtx (WOLFSSL * ssl) この関数は WOLFSSL 構造体の IOCB_CookieCtx メンバーを返します。
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) この関数は、wolfSSL が WOLFSSL_ISOTP でコンパイルされている場合に使用するために、wolfSSL の ISO-TP コンテキストを設定します。
void	wolfSSL_SSLEnableRead (WOLFSSL * ssl) この関数は IO レイヤーからの読み取りを無効にします。

	Name
void	wolfSSL_SSLEnableRead (WOLFSSL * ssl) この関数は IO レイヤーからの読み取りを有効にします。読み取りはデフォルトで有効になっており、wolfSSL_SSLEnableRead() を元に戻すために使用する必要があります。

C.57.2 Functions Documentation

C.57.2.1 function EmbedReceive

```
int EmbedReceive(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

この関数は受信埋め込みコールバックです。

Parameters:

- **ssl** **wolfSSL_new()**を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** バッファの char 型ポインタ表現。
- **sz** バッファのサイズ。
- **ctx** ユーザー登録コンテキストへの void 型ポインタ。デフォルトの場合、ctx はソケット記述子ポインタです。

See:

- **EmbedSend**
- **wolfSSL_CTX_SetIORecv**
- **wolfSSL_SSLEnableRecv**

Return:

- Success この関数は読み取られたバイト数を返します。
- WOLFSSL_CBIO_ERR_WANT_READ 最後のエラーが SOCKET_EWOULDBLOCK または SOCKET_EAGAIN の場合、“Would block” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_TIMEOUT “Socket timeout” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_CONN_RST 最後のエラーが SOCKET_ECONNRESET の場合、“Connection reset” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_ISR 最後のエラーが SOCKET_EINTR の場合、“Socket interrupted” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_WANT_READ 最後のエラーが SOCKET_ECONNREFUSED の場合、“Connection refused” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_CONN_CLOSE 最後のエラーが SOCKET_ECONNABORTED の場合、“Connection aborted” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_GENERAL 最後のエラーが指定されていない場合、“General error” メッセージとともに返されます。

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){
    // バイトが読み取られませんでした。失敗ケース。
}
```

C.57.2.2 function EmbedSend

```
int EmbedSend(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

この関数は送信埋め込みコールバックです。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** バッファを表す char 型ポインタ。
- **sz** バッファのサイズ。
- **ctx** ユーザー登録コンテキストへの void 型ポインタ。

See:

- [EmbedReceive](#)
- [wolfSSL_CTX_SetIOSend](#)
- [wolfSSL_SSLSendIOSend](#)

Return:

- Success この関数は送信されたバイト数を返します。
- WOLFSSL_CBIO_ERR_WANT_WRITE 最後のエラーが SOCKET_EWOULDBLOCK または SOCKET_EAGAIN の場合、“Would block” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_CONN_RST 最後のエラーが SOCKET_ECONNRESET の場合、“Connection reset” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_ISR 最後のエラーが SOCKET_EINTR の場合、“Socket interrupted” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_CONN_CLOSE 最後のエラーが SOCKET_EPIPE の場合、“Socket EPIPE” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_GENERAL 最後のエラーが指定されていない場合、“General error” メッセージとともに返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char* buf;
int sz;
void* ctx;
int dSent = EmbedSend(ssl, buf, sz, ctx);
if(dSent <= 0){
    // バイトが送信されませんでした。失敗ケース。
}
```

C.57.2.3 function EmbedReceiveFrom

```
int EmbedReceiveFrom(
    WOLFSSL * ssl,
    char * buf,
```



```

    int sz,
    void * ctx
)

```

この関数は受信埋め込みコールバックです。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** バッファへの const char 型ポインタ。
- **sz** バッファのサイズを表す int 型。
- **ctx** WOLFSSL_CTX コンテキストへの void 型ポインタ。

See:

- `EmbedSendTo`
- `wolfSSL_CTX_SetIORecv`
- `wolfSSL_SSLSetIORecv`
- `wolfSSL_dtls_get_current_timeout`

Return:

- Success 実行が成功した場合、この関数は読み取られた nb バイト数を返します。
- WOLFSSL_CBIO_ERR_WANT_READ 接続が拒否された場合、または関数で'would block' エラーがスローされた場合に返されます。
- WOLFSSL_CBIO_ERR_TIMEOUT ソケットがタイムアウトした場合に返されます。
- WOLFSSL_CBIO_ERR_CONN_RST 接続がリセットされた場合に返されます。
- WOLFSSL_CBIO_ERR_ISR ソケットが中断された場合に返されます。
- WOLFSSL_CBIO_ERR_GENERAL 一般的なエラーがあった場合に返されます。

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 は書き込まれたバイト数で正の値です
}

```

C.57.2.4 function EmbedSendTo

```

int EmbedSendTo(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)

```

この関数は送信埋め込みコールバックです。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** バッファを表す char 型ポインタ。
- **sz** バッファのサイズ。

- **ctx** ユーザー登録コンテキストへの void 型ポインタ。デフォルトの場合は WOLFSSL_DTLS_CTX 構造体です。

See:

- [EmbedReceiveFrom](#)
- [wolfSSL_CTX_SetIOSend](#)
- [wolfSSL_SSLSetIOSend](#)

Return:

- Success この関数は送信されたバイト数を返します。
- WOLFSSL_CBIO_ERR_WANT_WRITE 最後のエラーが SOCKET_EWOULDBLOCK または SOCKET_EAGAIN エラーの場合、“Would Block” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_CONN_RST 最後のエラーが SOCKET_ECONNRESET の場合、“Connection reset” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_ISR 最後のエラーが SOCKET_EINTR の場合、“Socket interrupted” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_CONN_CLOSE 最後のエラーが WOLFSSL_CBIO_ERR_CONN_CLOSE の場合、“Socket EPIPE” メッセージとともに返されます。
- WOLFSSL_CBIO_ERR_GENERAL 最後のエラーが指定されていない場合、“General error” メッセージとともに返されます。

Example

```
WOLFSSL* ssl;
...
char* buf;
int sz;
void* ctx;

int sEmbed = EmbedSendto(ssl, buf, sz, ctx);
if(sEmbed <= 0){
    // バイトが送信されませんでした。失敗ケース。
}
```

C.57.2.5 function EmbedGenerateCookie

```
int EmbedGenerateCookie(
    WOLFSSL * ssl,
    byte * buf,
    int sz,
    void * ctx
)
```

この関数は DTLS Generate Cookie コールバックです。

Parameters:

- **ssl** [wolfSSL_new\(\)](#)を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** バッファを表す byte 型ポインタ。XMEMCPY() の宛先です。
- **sz** バッファのサイズ。
- **ctx** ユーザー登録コンテキストへの void 型ポインタ。

See: [wolfSSL_CTX_SetGenCookie](#)

Return:

- Success この関数はバッファにコピーされたバイト数を返します。
- GEN_COOKIE_E EmbedGenerateCookie で getpeername が失敗した場合に返されます。

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 の成功コードブロック
}

```

C.57.2.6 function EmbedOcspRespFree

```

void EmbedOcspRespFree(
    void * ctx,
    byte * resp
)

```

この関数は応答バッファを解放します。

Parameters:

- **ctx** ヒープヒントへの void 型ポインタ。
- **resp** 応答を表す byte 型ポインタ。

See:

- [wolfSSL_CertManagerSetOCSP_Cb](#)
- [wolfSSL_CertManagerEnableOCSPStapling](#)
- [wolfSSL_CertManagerEnableOCSP](#)

Return: none 戻り値なし。

Example

```

void* ctx;
byte* resp; // 応答バッファ
...
EmbedOcspRespFree(ctx, resp);

```

C.57.2.7 function wolfSSL_CTX_SetIORecv

```

void wolfSSL_CTX_SetIORecv(
    WOLFSSL_CTX * ctx,
    CallbackIORecv CBIORcv
)

```

この関数は、wolfSSL が入力データを取得するための受信コールバックを登録します。デフォルトでは、wolfSSL はシステムの TCP recv() 関数を使用するコールバックとして EmbedReceive() を使用します。ユーザーは、メモリ、他のネットワークモジュール、または任意の場所から入力を取得する関数を登録できます。関数の動作方法とエラーコードについては、src/io.c の EmbedReceive() 関数をガイドとして参照してください。特に、データの準備ができていないノンブロッキング受信の場合は IO_ERR_WANT_READ を返す必要があります。

Parameters:

- **ctx** [wolfSSL_CTX_new\(\)](#)で作成された SSL コンテキストへのポインタ。

- **callback** wolfSSL コンテキスト ctx の受信コールバックとして登録される関数。この関数のシグネチャは、上記の Synopsis セクションに示されているものに従う必要があります。

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx

Return: none 戻り値なし。

Example

```
WOLFSSL_CTX* ctx = 0;
// 受信コールバックのプロトタイプ
int MyEmbedReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx);
// カスタム受信コールバックを wolfSSL に登録
wolfSSL_CTX_SetIORecv(ctx, MyEmbedReceive);
int MyEmbedReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx)
{
    // カスタム EmbedReceive 関数
}
```

C.57.2.8 function wolfSSL_SetIOReadCtx

```
void wolfSSL_SetIOReadCtx(
    WOLFSSL* ssl,
    void* ctx
)
```

この関数は、SSL セッションの受信コールバック関数のコンテキストを登録します。デフォルトでは、wolfSSL がシステムの TCP ライブラリを使用している場合、wolfSSL は wolfSSL_set_fd() に渡されたファイル記述子をコンテキストとして設定します。独自の受信コールバックを登録した場合、セッションの特定のコンテキストを設定することができます。たとえば、メモリバッファを使用している場合、コンテキストはメモリバッファへのアクセス方法と場所を記述する構造体へのポインタになる可能性があります。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **ctx** SSL セッション (ssl) の受信コールバック関数に登録されるコンテキストへのポインタ。

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOWriteCtx

Return: none 戻り値なし。

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
// 例として、受信 CTX としてソケット fd を手動で設定
wolfSSL_SetIOReadCtx(ssl, &sockfd);
...
```

C.57.2.9 function wolfSSL_SetIOWriteCtx

```
void wolfSSL_SetIOWriteCtx(  
    WOLFSSL * ssl,  
    void * ctx  
)
```

この関数は、SSL セッションの送信コールバック関数のコンテキストを登録します。デフォルトでは、wolfSSL がシステムの TCP ライブラリを使用している場合、wolfSSL は wolfSSL_set_fd() に渡されたファイル記述子をコンテキストとして設定します。独自の送信コールバックを登録した場合、セッションの特定のコンテキストを設定することができます。たとえば、メモリバッファを使用している場合、コンテキストはメモリバッファへのアクセス方法と場所を記述する構造体へのポインタになる可能性があります。

Parameters:

- **ssl** wolfSSL_new() で作成された SSL セッションへのポインタ。
- **wctx** SSL セッション (ssl) の送信コールバック関数に登録されるコンテキストへのポインタ。

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx

Return: none 戻り値なし。

Example

```
int sockfd;  
WOLFSSL* ssl = 0;  
...  
// 例として、送信 CTX としてソケット fd を手動で設定  
wolfSSL_SetIOWriteCtx(ssl, &sockfd);  
...
```

C.57.2.10 function wolfSSL_GetIOReadCtx

```
void * wolfSSL_GetIOReadCtx(  
    WOLFSSL * ssl  
)
```

この関数は WOLFSSL 構造体の IOCB_ReadCtx メンバーを返します。

Parameters:

- **ssl** wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_GetIOWriteCtx
- wolfSSL_SetIOReadFlags
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer この関数は WOLFSSL 構造体の IOCB_ReadCtx メンバーへの void 型ポインタを返します。
- NULL WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
void* ioRead;
...
ioRead = wolfSSL_GetIOReadCtx(ssl);
if(ioRead == NULL){
    // 失敗ケース。ssl オブジェクトが NULL でした。
}
```

C.57.2.11 function wolfSSL_GetIOWriteCtx

```
void * wolfSSL_GetIOWriteCtx(
    WOLFSSL * ssl
)
```

この関数は WOLFSSL 構造体の IOCB_WriteCtx メンバーを返します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_GetIOReadCtx`
- `wolfSSL_SetIOWriteCtx`
- `wolfSSL_SetIOReadCtx`
- `wolfSSL_CTX_SetIOSend`

Return:

- pointer この関数は WOLFSSL 構造体の IOCB_WriteCtx メンバーへの void 型ポインタを返します。
- NULL WOLFSSL 構造体が NULL の場合に返されます。

Example

```
WOLFSSL* ssl;
void* ioWrite;
...
ioWrite = wolfSSL_GetIOWriteCtx(ssl);
if(ioWrite == NULL){
    // 関数が NULL を返しました。
}
```

C.57.2.12 function wolfSSL_SetIOReadFlags

```
void wolfSSL_SetIOReadFlags(
    WOLFSSL * ssl,
    int flags
)
```

この関数は、指定された SSL セッションの受信コールバックで使用するフラグを設定します。受信コールバックは、デフォルトの `wolfSSL_EMBED_RECEIVE` コールバック、またはユーザーが指定したカスタムコールバックのいずれかです（`wolfSSL_CTX_SetIORecv` を参照）。デフォルトのフラグ値は、`wolfSSL` 内部で 0 の値に設定されます。デフォルトの `wolfSSL` 受信コールバックは、`recv()` 関数を使用してソケットからデータを受信します。`recv()` の man ページより：「`recv()` 関数の `flags` パラメータは、次の値の 1 つ以上を OR して形成されます：MSG_OOB 帯域外データを処理、MSG_PEEK 受信メッセージを覗き見、MSG_WAITALL 完全なリクエストまたはエラーを待機。MSG_OOB フラグは、通常の変換データストリームでは受信されない帯域外データの受信を要求します。一部のプロトコルは、通常の変換データキューの先頭に緊急データを配置するため、このフラグはそのようなプロトコルでは使用できません。MSG_PEEK フラグは、受信操作が受信キューの先頭からデータを返すようにしますが、そのデータをキューから削除しません。したがって、後続

の受信呼び出しは同じデータを返します。MSG_WAITALL フラグは、完全なリクエストが満たされるまで操作をブロックするよう要求します。ただし、シグナルがキャッチされた場合、エラーまたは切断が発生した場合、または次に受信されるデータが返されたデータとは異なるタイプの場合、呼び出しは要求されたデータよりも少ないデータを返す可能性があります。」

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL セッションへのポインタ。
- **flags** 指定された SSL セッション (ssl) の I/O 読み取りフラグの値。

See:

- `wolfSSL_CTX_SetIORecv`
- `wolfSSL_CTX_SetIOSend`
- `wolfSSL_SetIOReadCtx`

Return: none 戻り値なし。

Example

```
WOLFSSL* ssl = 0;
...
// recv フラグを手動で 0 に設定
wolfSSL_SetIOReadFlags(ssl, 0);
...
```

C.57.2.13 function `wolfSSL_SetIOWriteFlags`

```
void wolfSSL_SetIOWriteFlags(
    WOLFSSL * ssl,
    int flags
)
```

この関数は、指定された SSL セッションの送信コールバックで使用するフラグを設定します。送信コールバックは、デフォルトの `wolfSSL_EMBED_SEND` コールバック、またはユーザーが指定したカスタムコールバックのいずれかです (`wolfSSL_CTX_SetIOSend` を参照)。デフォルトのフラグ値は、`wolfSSL` 内部で 0 の値に設定されます。デフォルトの `wolfSSL` 送信コールバックは、`send()` 関数を使用してソケットからデータを送信します。`send()` の man ページより: 「flags パラメータには、次の 1 つ以上が含まれる場合があります: `#define MSG_OOB 0x1` // 帯域外データを処理、`#define MSG_DONTROUTE 0x4` // ルーティングをバイパス、直接インターフェースを使用。フラグ `MSG_OOB` は、この概念をサポートするソケット (例: `SOCK_STREAM`) で「帯域外」データを送信するために使用されます。基礎となるプロトコルも「帯域外」データをサポートする必要があります。`MSG_DONTROUTE` は通常、診断またはルーティングプログラムによってのみ使用されます。」

Parameters:

- **ssl** `wolfSSL_new()`で作成された SSL セッションへのポインタ。
- **flags** 指定された SSL セッション (ssl) の I/O 送信フラグの値。

See:

- `wolfSSL_CTX_SetIORecv`
- `wolfSSL_CTX_SetIOSend`
- `wolfSSL_SetIOReadCtx`

Return: none 戻り値なし。

Example

```
WOLFSSL* ssl = 0;
...
// 送信フラグを手動で 0 に設定
```

```
wolfSSL_SetIOWriteFlags(ssl, 0);
...
```

C.57.2.14 function wolfSSL_SetIO_NetX

```
void wolfSSL_SetIO_NetX(
    WOLFSSL * ssl,
    NX_TCP_SOCKET * nxsocket,
    ULONG waitoption
)
```

この関数は、WOLFSSL 構造体内の nxCtx 構造体の nxSocket および nxWait メンバーを設定します。

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **nxSocket** nxCTX 構造体の nxSocket メンバーに設定される NX_TCP_SOCKET 型へのポインタ。
- **waitOption** nxCtx 構造体の nxWait メンバーに設定される ULONG 型。

See:

- set_fd
- NetX_Send
- NetX_Receive

Return: none 戻り値なし。

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 {
    // 適切なパラメータを渡す必要があります。
}
```

C.57.2.15 function wolfSSL_CTX_SetGenCookie

```
void wolfSSL_CTX_SetGenCookie(
    WOLFSSL_CTX * ctx,
    CallbackGenCookie cb
)
```

この関数は、WOLFSSL_CTX 構造体の CBIOWriteCookie メンバーのコールバックを設定します。CallbackGenCookie 型は関数ポインタで、次のシグネチャを持ちます：int (CallbackGenCookie)(WOLFSSL ssl, unsigned char* buf, int sz, void* ctx);

Parameters:

- **ssl** `wolfSSL_new()`を使用して作成された WOLFSSL 構造体へのポインタ。
- **cb** CallbackGenCookie のシグネチャを持つ CallbackGenCookie 型の関数ポインタ。

See: CallbackGenCookie

Return: none 戻り値なし。

Example


```

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
int SetGenCookieCB(WOLFSSL* ssl, unsigned char* buf, int sz, void* ctx){
// コールバック関数本体
}
...
wolfSSL_CTX_SetGenCookie(ssl->ctx, SetGenCookieCB);

```

C.57.2.16 function wolfSSL_GetCookieCtx

```

void * wolfSSL_GetCookieCtx(
    WOLFSSL * ssl
)

```

この関数は WOLFSSL 構造体の IOCB_CookieCtx メンバーを返します。

Parameters:

- **ssl** `wolfSSL_new()` を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- `wolfSSL_SetCookieCtx`
- `wolfSSL_CTX_SetGenCookie`

Return:

- pointer 関数は IOCB_CookieCtx に保存されている void 型ポインタ値を返します。
- NULL WOLFSSL 構造体が NULL の場合。

Example

```

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
void* cookie;
...
cookie = wolfSSL_GetCookieCtx(ssl);
if(cookie != NULL){
// cookie を取得しました
}

```

C.57.2.17 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
)

```

この関数は、wolfSSL が WOLFSSL_ISOTP でコンパイルされている場合に使用するために、wolfSSL の ISO-TP コンテキストを設定します。

Parameters:

- **ssl** wolfSSL コンテキスト。
- **ctx** この関数が初期化するユーザー作成の ISOTP コンテキスト。
- **recv_fn** ユーザーの CAN バス受信コールバック。
- **send_fn** ユーザーの CAN バス送信コールバック。
- **delay_fn** ユーザーのマイクロ秒粒度遅延関数。
- **receive_delay** 各 CAN バスパケットを遅延させる設定マイクロ秒数。
- **receive_buffer** データを受信するためのユーザー提供バッファ、ISOTP_DEFAULT_BUFFER_SIZE バイトに割り当てることを推奨。
- **receive_buffer_size** - receive_buffer のサイズ。
- **arg** recv_fn と send_fn に送信される任意のポインタ。

Return: 0 成功時、WOLFSSL_CBIO_ERR_GENERAL 失敗時。

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.57.2.18 function wolfSSL_SSLSDisableRead

```
void wolfSSL_SSLSDisableRead(
    WOLFSSL * ssl
)
```

この関数は IO レイヤーからの読み取りを無効にします。

Parameters:

- **ssl** wolfSSL コンテキスト。

See:

- [wolfSSL_CTX_SetIORecv](#)
- [wolfSSL_SSLSSetIORecv](#)
- [wolfSSL_SSEnableRead](#)

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLSDisableRead(ssl);
```

C.57.2.19 function wolfSSL_SSEnableRead

```
void wolfSSL_SSEnableRead(
    WOLFSSL * ssl
)
```

この関数は IO レイヤーからの読み取りを有効にします。読み取りはデフォルトで有効になっており、wolfSSL_SSLSDisableRead() を元に戻すために使用する必要があります。

Parameters:

- **ssl** wolfSSL コンテキスト。

See:

- [wolfSSL_CTX_SetIORecv](#)
- [wolfSSL_SSLSetIORecv](#)
- [wolfSSL_SSEnableRead](#)

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLSDisableRead(ssl);
...
wolfSSL_SSEnableRead(ssl);
```

C.57.3 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 CBIORcv);

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_CTX_SetGenCookie(WOLFSSL_CTX* ctx, CallbackGenCookie cb);

void* wolfSSL_GetCookieCtx(WOLFSSL* ssl);

int wolfSSL_SetIO_ISOTP(WOLFSSL *ssl, isotp_wolfssl_ctx *ctx,
                        can_rcv_fn rcv_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_SSLEnableRead(WOLFSSL *ssl);
```

```
void wolfSSL_SSLDisableRead(WOLFSSL *ssl);
```

D SSL/TLS の概要

D.1 アーキテクチャ

組み込み向け SSL/TLS ライブラリである wolfSSL(旧称: Cyassl) は、SSL 3.0、TLS 1.0、TLS 1.1、TLS 1.2、TLS 1.3 プロトコルを実装しています。TLS 1.3 は、標準化されている最も安全な最新バージョンです。SSL 2.0 は数年間不安定であることから、サポートしていません。

wolfSSL の TLS プロトコルは、RFC 5246 で定義されている通りに実装されています。TLS には、メッセージ層とハンドシェイク層の 2 つのレコード層プロトコルが存在します。ハンドシェイクメッセージは、共通の暗号スイートのネゴシエーション、シークレットの作成、および安全な接続の有効化に使用されます。メッセージレイヤーはハンドシェイクレイヤーをカプセル化すると同時に、アラート処理とアプリケーションデータ転送もサポートします。

SSL プロトコルが既存のプロトコルにどのように適合するかを、図 1 に示します。SSL は、OSI モデルのトランスポート層とアプリケーション層の間に位置し、多くのプロトコル (TCP/IP、Bluetooth など) がトランスポートメディアとして機能します。HTTP、FTP、SMTP などのアプリケーションプロトコルは、SSL の上に階層化されています。

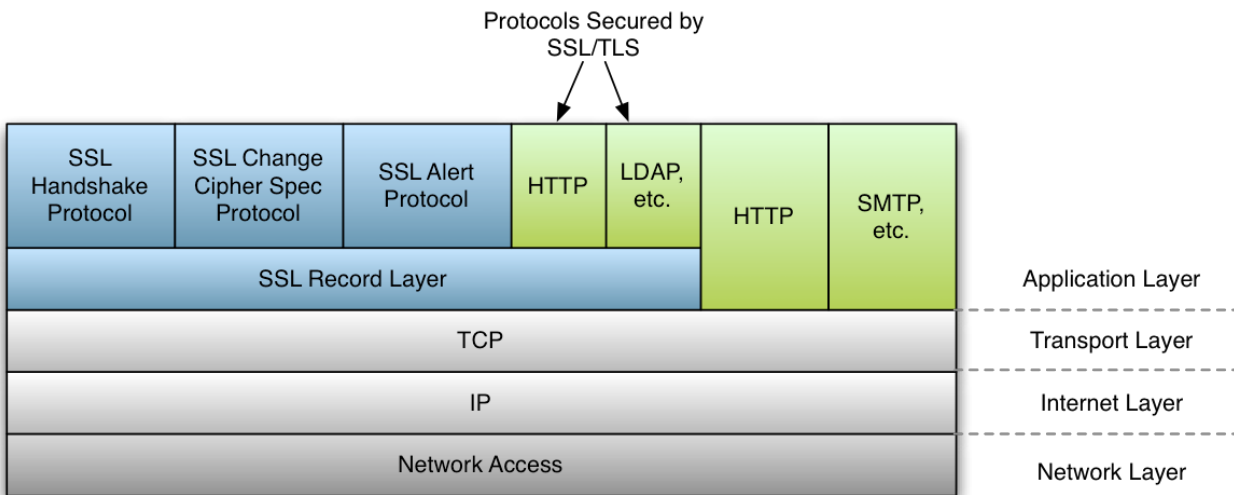


Figure 5: 概要図

D.2 SSL ハンドシェイク

SSL ハンドシェイクプロセスの簡略を以下の図 2 に示します。なお、SSL クライアントとサーバーの構成オプションによってはいくつかのステップは実行されません。

D.3 SSL/TLS における、バージョンごとの違い

SSL (Secure Sockets Layer) と TLS (Transport Security Layer) はどちらも、ネットワーク上で安全な通信を提供する暗号化プロトコルです。これら 2 つのプロトコルは、今日、Web ブラウジングから E メール、インスタントメッセージング、VoIP に至るまで、さまざまなアプリケーションで広く使用されています。SSL と TLS、そしてそれぞれに含まれる各バージョン間にはいくつかの違いがあります。

以下に、SSL および TLS プロトコルの各バージョンについての解説と主な相違点を示します。各プロトコルの具体的な詳細については、記載されている RFC 仕様を参照してください。

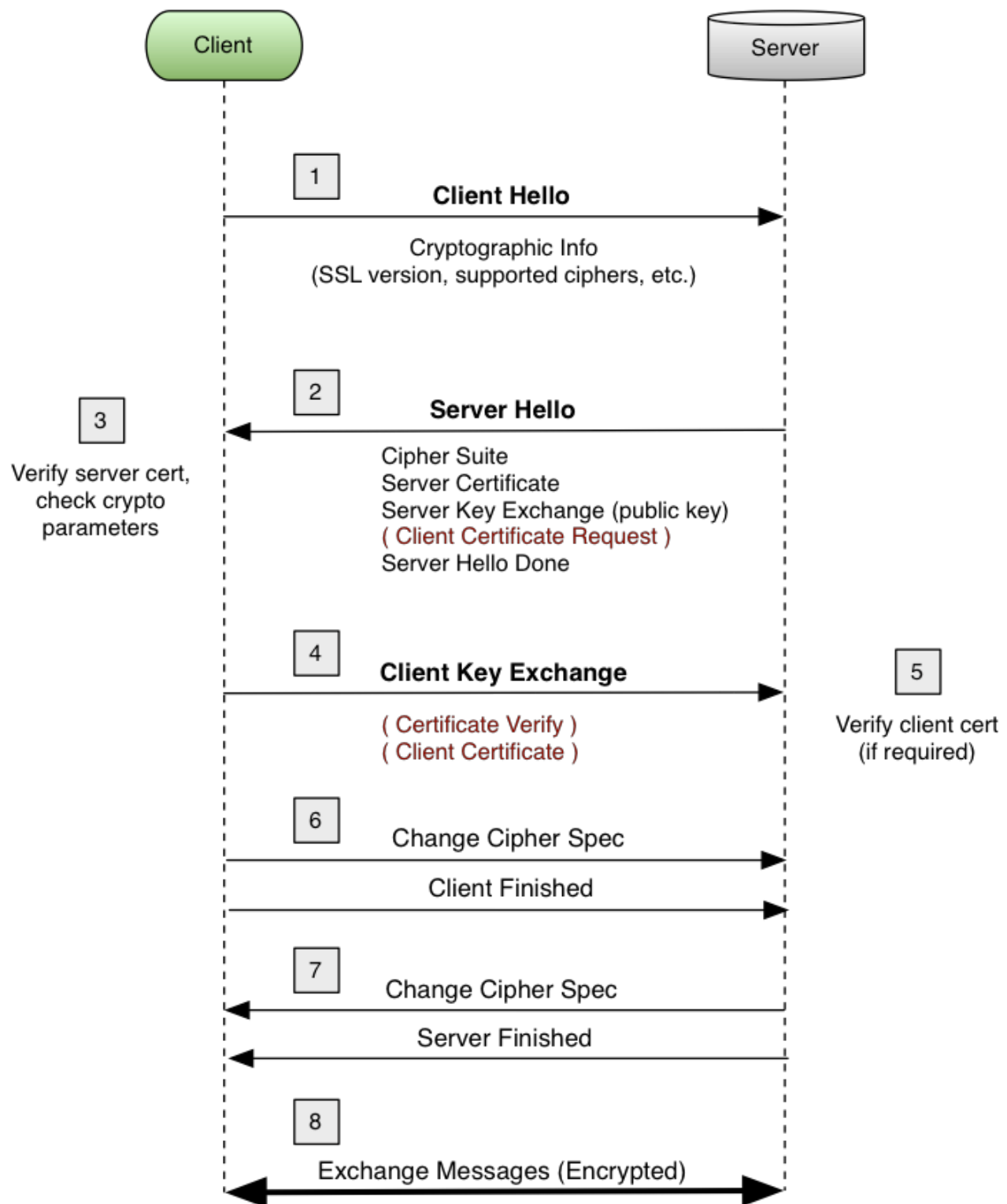


Figure 6: ハンドシェイク

D.3.1 SSL 3.0

このプロトコルは Netscape によって開発された SSL 1.0 から始まり、1996 年にリリースされました。バージョン 1.0 はリリースされておらず、バージョン 2.0 には多くのセキュリティ上の欠陥があり、SSL 3.0 のリリースにつながりました。SSL 2.0 に対する SSL 3.0 のいくつかの主要な改善点は次のとおりです。

- データ転送をメッセージレイヤーから分離
- Export Cipher を使用している場合でも、128 ビットのキーイングマテリアルを使用
- クライアントとサーバーが証明書のチェーンを送信し、組織は 2 つ以上の証明書の証明書階層を使用可能に
- 一般化された鍵交換プロトコルを実装し、Diffie-Hellman と Fortezza の鍵交換と非 RSA 証明書を許可
- レコードの圧縮と解凍を可能に
- 2.0 クライアントが検出されたときに SSL 2.0 にフォールバック

D.3.2 TLS 1.0

このプロトコルは 1999 年 1 月に [RFC 2246](#) で最初に定義されました。これは SSL 3.0 からのアップグレードであり劇的な違いはありませんが、SSL 3.0 と TLS 1.0 は相互運用できない程度の変更を含んでいます。SSL 3.0 と TLS 1.0 の間の大きな違いとして、以下が挙げられます。

- 鍵導出関数の変更
- MAC の変更 - SSL 3.0 は初期 HMAC の変更を使用し、TLS 1.0 は HMAC を使用します。
- 完了 (Finished) メッセージの変更
- アラートの増加
- DSS/DH サポートの要求

D.3.3 TLS 1.1

このプロトコルは、2006 年 4 月に TLS 1.0 の後継として [RFC 4346](#) で定義されました。主な変更点は次のとおりです。

- 暗黙の初期化ベクトル (IV) は、暗号ブロック連鎖 (CBC) 攻撃から保護するために明示的な IV に置き換えられました。
- パディングエラーの取り扱いは、CBC 攻撃から保護するために `decryption_failed` アラートではなく `bad_record_mac` アラートを使用するよう変更されました。
- IANA レジストリは、プロトコルパラメーター用に定義されました。
- 早期終了によってセッションが再開できなくなることがなくなりました。

D.3.4 TLS 1.2

このプロトコルは、2008 年 8 月に [RFC 5246](#) で定義されました。TLS 1.1 をベースとして、いくつかの改善が行われました。主な相違点は次のとおりです。

- 擬似ランダム関数 (PRF) の MD5/SHA-1 の組み合わせを、暗号スイート指定 PRF に置き換えました。
- デジタル署名要素の MD5/SHA-1 の組み合わせは、単一のハッシュに置き換えられました。署名された要素には、使用されるハッシュアルゴリズムを明示的に指定するフィールドが含まれます。
- クライアントとサーバーが受け入れるハッシュおよび署名アルゴリズムの組み合わせが整理されました。
- 追加のデータモードを使用した、認証された暗号化のためのサポートが追加されました。
- TLS 拡張機能の定義と AES 暗号スイートがマージされました。
- EncryptedPremasterSecret バージョン番号が厳しくチェックされるようになりました。
- 多くの要件が厳しくなりました。
- Verify_data の長さは暗号スイートに依存します
- Bleichenbacher/Dlima 攻撃防御の説明が簡潔になりました。

D.3.5 TLS 1.3

このプロトコルは、2018 年 8 月に[RFC 8446](#)で定義されました。主に、セキュリティ性能とスピードが向上しています。主な違いは次のとおりです。

- サポートされている対称アルゴリズムのリストは、すべての従来のアルゴリズムから整理されました。残りのアルゴリズムはすべて、認証タグ付き暗号 (AEAD) アルゴリズムを使用します。
- ゼロ RTT (0-RTT) モードが追加され、一部のセキュリティ属性を犠牲にすることで、一部のアプリケーションデータのための接続時のラウンドトリップが削減されました。
- ServerHello の後のすべてのハンドシェイクメッセージが暗号化されるようになりました。
- HMAC ベースの抽出および拡張鍵導出機能 (HKDF) がプリミティブとして使用されているため、鍵導出機能が再設計されました。
- ハンドシェイクステートマシンが再構築され、一貫性が向上し、余分なメッセージが削除されました。
- ECC は基本仕様になり、新しい署名アルゴリズムが含まれるようになりました。各曲線の単一のポイント形式を支持して、ポイント形式のネゴシエーションを削除しました。
- 圧縮、カスタム DHE グループ、および DSA が削除されました、RSA パディングは PSS を使用するようになりました。
- TLS 1.2 バージョンネゴシエーション検証メカニズムは廃止され、拡張機能のバージョンリストが採用されました。
- サーバー側の状態の有無にかかわらず、セッションの再開と、TLS の以前のバージョンの PSK ベースの暗号スイートは、単一の新しい PSK 交換に置き換えられました。

E RFC・仕様・リファレンス

E.1 プロトコル

- SSL v3.0 - [IETF Draft](#)
- TLS v1.0 - [RFC2246](#)
- TLS v1.1 - [RFC4346](#)
- TLS v1.2 - [RFC5246](#)
- TLS v1.3 - [RFC8446](#)
- DTLS - [RFC4347 仕様文書](#)
- IPv4 の概要 - [Wikipedia](#)
- IPv6 の概要 - [Wikipedia](#)

E.2 ストリーム暗号

- 概要 - [Wikipedia](#)
- RC4 / ARC4 - [IETF Draft Wikipedia](#)

E.3 ブロック暗号

- 概要 - [Wikipedia](#)
- AES - [NIST Publication Wikipedia](#)
- AES-GCM - [NIST Specification](#)
- AES-NI - [Intel Software Network](#)
- DES/3DES - [NIST Publication Wikipedia](#)

E.4 ハッシュ

- SHA - [NIST FIPS180-1 Publication](#) [NIST FIPS180-2 Publication](#) [Wikipedia](#)
- MD4 - [RFC1320](#)
- MD5 - [RFC1321](#)
- RIPEMD-160 - [Specification document](#)

E.5 公開鍵暗号

- 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 その他

- PKCS#5, PBKDF1, PBKDF2 - [RFC2898](#)
- PKCS#8 - [RFC5208](#)
- PKCS#12 - [Wikipedia](#)

F エラーコード

F.1 wolfSSL エラーコード

wolfSSL のエラーコードは、wolfssl/ssl.h に記載しています。以下のエラーの詳細な説明については、OpenSSL のマニュアルページ `SSL_get_error` (man `SSL_get_error`) をご参照ください。

エラーコード列挙型	エラーコード
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

wolfSSL の追加のエラーコードは、wolfssl/error-ssl.h に記載しています。

エラーコード列挙型	エラーコード	エラーの説明
INPUT_CASE_ERROR	-301	プロセス入力状態エラー
PREFIX_ERROR	-302	鍵ラウンドへの不正なインデックス
MEMORY_ERROR	-303	メモリ確保失敗
VERIFY_FINISHED_ERROR	-304	Finished メッセージ検証に失敗
VERIFY_MAC_ERROR	-305	MAC 検証に失敗
PARSE_ERROR	-306	ヘッダーの解析エラー
UNKNOWN_HANDSHAKE_TYPE	-307	不正なハンドシェイクタイプ
SOCKET_ERROR_E	-308	ソケットがエラー状態
SOCKET_NODATA	-309	予想されるデータが存在しない
INCOMPLETE_DATA	-310	タスクを完了するのに十分なデータがない
UNKNOWN_RECORD_TYPE	-311	レコードヘッダーの不明なタイプ
DECRYPT_ERROR	-312	復号中のエラー
FATAL_ERROR	-313	致命的なエラーアラートを受信
ENCRYPT_ERROR	-314	暗号化中のエラー
FREAD_ERROR	-315	fread でエラー
NO_PEER_KEY	-316	ピアの鍵が存在しない
NO_PRIVATE_KEY	-317	秘密鍵が存在しない
RSA_PRIVATE_ERROR	-318	RSA 秘密鍵操作中のエラー
NO_DH_PARAMS	-319	サーバーが DH パラメータを送信してこない
BUILD_MSG_ERROR	-320	メッセージの作成に失敗
BAD_HELLO	-321	ClientHello メッセージが不正
DOMAIN_NAME_MISMATCH	-322	ピアのサブジェクト名の不一致
WANT_READ	-323	データ再読み取りが必要
NOT_READY_ERROR	-324	ハンドシェイクレイヤーが Ready でない
VERSION_ERROR	-326	レコードレイヤーのバージョンエラー
WANT_WRITE	-327	データの再送が必要
BUFFER_ERROR	-328	不正な形式のバッファ入力
VERIFY_CERT_ERROR	-329	証明書検証エラー
VERIFY_SIGN_ERROR	-330	署名検証エラー
CLIENT_ID_ERROR	-331	PSK クライアント ID エラー
SERVER_HINT_ERROR	-332	PSK サーバーヒントエラー

エラーコード列挙型	エラーコード	エラーの説明
PSK_KEY_ERROR	-333	PSK 鍵エラー
GETTIME_ERROR	-337	gettimeofday が失敗
GETTIMER_ERROR	-338	gettimer が失敗
SIGACT_ERROR	-339	sigaction が失敗
SETTIMER_ERROR	-340	settimer が失敗
LENGTH_ERROR	-341	レコードレイヤーの長さが不正
PEER_KEY_ERROR	-342	ピア鍵のデコードに失敗
ZERO_RETURN	-343	ピアが close notify を送信
SIDE_ERROR	-344	クライアント/サーバータイプが不正
NO_PEER_CERT	-345	ピアが鍵を送信しなかった
ECC_CURVETYPE_ERROR	-350	不正な ECC 曲線タイプ
ECC_CURVE_ERROR	-351	不正な ECC 曲線
ECC_PEERKEY_ERROR	-352	不正なピア ECC 鍵
ECC_MAKEKEY_ERROR	-353	不正な ECC 鍵の作成
ECC_EXPORT_ERROR	-354	不正な ECC エクスポート鍵
ECC_SHARED_ERROR	-355	不正な ECC 共有秘密
NOT_CA_ERROR	-357	CA 証明書ではない
BAD_CERT_MANAGER_ERROR	-359	不正な証明書マネージャー
OCSP_CERT_REVOKED	-360	OCSP 証明書が失効
CRL_CERT_REVOKED	-361	CRL 証明書が失効
CRL_MISSING	-362	CRL が読み込まれていない
MONITOR_SETUP_E	-363	CRL モニターのセットアップエラー
THREAD_CREATE_E	-364	スレッド作成エラー
OCSP_NEED_URL	-365	OCSP は検索用の URL が必要
OCSP_CERT_UNKNOWN	-366	OCSP レスポンダーが不明
OCSP_LOOKUP_FAIL	-367	OCSP 検索が失敗
MAX_CHAIN_ERROR	-368	最大チェーン深度を超えた
COOKIE_ERROR	-369	DTLS クッキーエラー
SEQUENCE_ERROR	-370	DTLS シーケンスエラー
SUITES_ERROR	-371	スイートポインタエラー
OUT_OF_ORDER_E	-373	順序外のメッセージ
BAD_KEA_TYPE_E	-374	不正な KEA タイプが見つかった
SANITY_CIPHER_E	-375	暗号のサニティチェックエラー
RECV_OVERFLOW_E	-376	受信コールバックが要求以上のデータを返却した
GEN_COOKIE_E	-377	クッキー生成エラー
NO_PEER_VERIFY	-378	検証すべきピア証明書が存在しない
FWRITE_ERROR	-379	fwrite エラー
CACHE_MATCH_ERROR	-380	キャッシュ HRD 不一致
UNKNOWN_SNI_HOST_NAME_E	-381	認識されないホスト名
UNKNOWN_MAX_FRAG_LEN_E	-382	認識されない最大フラグメント
KEYUSE_SIGNATURE_E	-383	KeyUse digSignature エラー
KEYUSE_ENCIPHER_E	-385	KeyUse KeyEncipher エラー
EXTKEYUSE_AUTH_E	-386	ExtKeyUse server・client_auth
SEND_OOB_READ_E	-387	送信コールバックが境界外読み取り
SECURE_RENEGOTIATION_E	-388	無効な再ネゴシエーション情報
SESSION_TICKET_LEN_E	-389	セッションチケットが大きすぎる
SESSION_TICKET_EXPECT_E	-390	セッションチケットがない
SCR_DIFFERENT_CERT_E	-391	SCR 異なる証明書エラー
SESSION_SECRET_CB_E	-392	セッション秘密 CB 関数の失敗
NO_CHANGE_CIPHER_E	-393	Cipher 変更前に終了
SANITY_MSG_E	-394	メッセージ順序のサニティチェックエラー
DUPLICATE_MST_E	-395	重複メッセージエラー

エラーコード列挙型	エラーコード	エラーの説明
SNI_UNSUPPORTED	-396	SSL 3.0 は SNI をサポートしていない
SOCKET_PEER_CLOSED_E	-397	トランスポート層がクローズされた
BAD_TICKET_KEY_CB_SZ	-398	不正なセッションチケット鍵 cb サイズ
BAD_TICKET_MSG_SZ	-399	不正なセッションチケット msg サイズ
BAD_TICKET_ENCRYPT	-400	不正なユーザーチケット暗号化
DH_KEY_SIZE_E	-401	DH 鍵が小さすぎる
SNI_ABSENT_ERROR	-402	SNI リクエストなし
RSA_SIGN_FAULT	-403	RSA 署名障害
HANDSHAKE_SIZE_ERROR	-404	ハンドシェイクメッセージが大きすぎる
UNKNOWN_ALPN_PROTOCOL_NAME_E	-405	認識されないプロトコル名
BAD_CERTIFICATE_STATUS_ERROR	-406	不正な証明書ステータスメッセージ
OCSP_INVALID_STATUS	-407	無効な OCSP ステータス
OCSP_WANT_READ	-408	OCSP コールバックレスポンス
RSA_KEY_SIZE_E	-409	RSA 鍵が小さすぎる
ECC_KEY_SIZE_E	-410	ECC 鍵が小さすぎる
DTLS_EXPORT_VER_E	-411	エクスポートバージョンエラー
INPUT_SIZE_E	-412	入力サイズが大きすぎる
CTX_INIT_MUTEX_E	-413	コンテキストミューテックス初期化エラー
EXT_MASTER_SECRET_NEEDED_E	-414	再開するには EMS を有効にする必要がある
DTLS_POOL_SZ_E	-415	DTLS プールサイズを超えた
DECODE_E	-416	ハンドシェイクメッセージのデコードエラー
HTTP_TIMEOUT	-417	OCSP または CRL リクエストの HTTP タイムアウト
WRITE_DUP_READ_E	-418	書き込み複製の書き込み側が読み込めない
WRITE_DUP_WRITE_E	-419	書き込み複製の読み込み側が書き込めない
INVALID_CERT_CTX_E	-420	TLS 証明書コンテキストが一致しない
BAD_KEY_SHARE_DATA	-421	鍵共有データが無効
MISSING_HANDSHAKE_DATA	-422	ハンドシェイクメッセージにデータがない
BAD_BINDER	-423	バインダーが一致しない
EXT_NOT_ALLOWED	-424	拡張子がメッセージで許可されていない
INVALID_PARAMETER	-425	セキュリティパラメータが無効
MCAST_HIGHWATER_CB_E	-426	マルチキャストハイウォーター cb エラー
ALERT_COUNT_E	-427	アラートカウントを超えた
EXT_MISSING	-428	必要な拡張機能が見つからない
UNSUPPORTED_EXTENSION	-429	TLSX はクライアントから要求されていない
PRF_MISSING	-430	PRF がコンパイルされていない
DTLS_RETX_OVER_TX	-431	DTLS フライトを再送信
DH_PARAMS_NOT_FFDHE_E	-432	サーバーからの DH パラメータが FFDHE でない
TCA_INVALID_ID_TYPE	-433	TLSX TCA ID タイプが無効
TCA_ABSENT_ERROR	-434	TLSX TCA ID レスポンスなし

ネゴシエーションパラメータエラー

エラーコード列挙型	エラーコード	エラーの説明
UNSUPPORTED_SUITE	-500	サポートされていない暗号スイート
MATCH_SUITE_ERROR	-501	暗号スイートの不一致
COMPRESSION_ERROR	-502	圧縮の不一致
KEY_SHARE_ERROR	-503	鍵共有の不一致
POST_HAND_AUTH_ERROR	-504	クライアントがポストハンド認証を行わない
HRR_COOKIE_ERROR	-505	HRR メッセージクッキーの不一致

F.2 wolfCrypt エラーコード

wolfCrypt のエラーコードは、wolfssl/wolfcrypt/error.h に記載しています。

エラーコード列挙型	エラーコード	エラーの説明
OPEN_RAN_E	-101	ランダムデバイスを開く際にエラー
READ_RAN_E	-102	ランダムデバイスの読み取りエラー
WINCRYPT_E	-103	Windows crypt 初期化エラー
CRYPTGEN_E	-104	Windows crypt 暗号化エラー
RAN_BLOCK_E	-105	ランダムデバイスの読み取りがブロックされる
BAD_MUTEX_E	-106	ミューテックス操作で失敗
MP_INIT_E	-110	mp_init エラー
MP_READ_E	-111	mp_read エラー
MP_EXPTMOD_E	-112	mp_exptmod エラー
MP_TO_E	-113	mp_to_xxx エラー、変換できない
MP_SUB_E	-114	mp_sub エラー、引き算できない
MP_ADD_E	-115	mp_add エラー、足し算できない
MP_MUL_E	-116	mp_mul エラー、掛け算できない
MP_MULMOD_E	-117	mp_mulmod エラー、mod 掛け算できない
MP_MOD_E	-118	mp_mod エラー、mod できない
MP_INVMOD_E	-119	mp_invmod エラー、invmod できない
MP_CMP_E	-120	mp_cmp エラー
MP_ZERO_E	-121	予期しない mp zero の結果を取得
MEMORY_E	-125	メモリ不足エラー
RSA_WRONG_TYPE_E	-130	RSA 関数に対する不正なブロックタイプ
RSA_BUFFER_E	-131	RSA バッファエラー、出力が小さすぎるか入力が大きすぎる
BUFFER_E	-132	出力バッファが小さすぎるか入力が大きすぎる
ALGO_ID_E	-133	アルゴリズム ID 設定エラー
PUBLIC_KEY_E	-134	公開鍵設定エラー
DATE_E	-135	日付有効性エラー
SUBJECT_E	-136	サブジェクト名設定エラー
ISSUER_E	-137	発行者名設定エラー
CA_TRUE_E	-138	CA 基本制約を true に設定するエラー
EXTENSIONS_E	-139	拡張設定エラー
ASN_PARSE_E	-140	ASN 解析エラー、無効な入力
ASN_VERSION_E	-141	ASN バージョンエラー、無効な番号
ASN_GETINT_E	-142	ASN 大きな整数取得エラー、無効なデータ
ASN_RSA_KEY_E	-143	ASN 鍵初期化エラー、無効な入力
ASN_OBJECT_ID_E	-144	ASN オブジェクト ID エラー、無効な ID
ASN_TAG_NULL_E	-145	ASN タグエラー、null でない
ASN_EXPECT_0_E	-146	ASN 期待値エラー、ゼロでない
ASN_BITSTR_E	-147	ASN ビット文字列エラー、不正な ID
ASN_UNKNOWN_OID_E	-148	ASN OID エラー、不明な合計 ID
ASN_DATE_SZ_E	-149	ASN 日付エラー、不正なサイズ
ASN_BEFORE_DATE_E	-150	ASN 日付エラー、現在の日付が前
ASN_AFTER_DATE_E	-151	ASN 日付エラー、現在の日付が後
ASN_SIG_OID_E	-152	ASN 署名エラー、OID が一致しない
ASN_TIME_E	-153	ASN 時間エラー、不明な時間タイプ
ASN_INPUT_E	-154	ASN 入力エラー、データ不足
ASN_SIG_CONFIRM_E	-155	ASN 署名エラー、確認失敗
ASN_SIG_HASH_E	-156	ASN 署名エラー、サポートされていないハッシュタイプ
ASN_SIG_KEY_E	-157	ASN 署名エラー、サポートされていない鍵タイプ
ASN_DH_KEY_E	-158	ASN 鍵初期化エラー、無効な入力

エラーコード列挙型	エラーコード	エラーの説明
ASN_CRIT_EXT_E	-160	ASN サポートされていない重要な拡張
ECC_BAD_ARG_E	-170	ECC 入力引数の型が間違っている
ASN_ECC_KEY_E	-171	ASN ECC 不正な入力
ECC_CURVE_OID_E	-172	サポートされていない ECC OID 曲線タイプ
BAD_FUNC_ARG	-173	不正な関数引数が提供された
NOT_COMPILED_IN	-174	機能がコンパイルされていない
UNICODE_SIZE_E	-175	Unicode パスワードが大きすぎる
NO_PASSWORD	-176	ユーザーからのパスワード提供なし
ALT_NAME_E	-177	Alt Name のサイズ問題、大きすぎる
AES_GCM_AUTH_E	-180	AES-GCM 認証チェック失敗
AES_CCM_AUTH_E	-181	AES-CCM 認証チェック失敗
CAVIUM_INIT_E	-182	Cavium 初期化タイプエラー
COMPRESS_INIT_E	-183	圧縮初期化エラー
COMPRESS_E	-184	圧縮エラー
DECOMPRESS_INIT_E	-185	解凍初期化エラー
DECOMPRESS_E	-186	解凍エラー
BAD_ALIGN_E	-187	操作の不正なアライメント、割り当てなし
ASN_NO_SIGNER_E	-188	ASN 署名エラー、証明書を検証する CA の署名者なし
ASN_CRL_CONFIRM_E	-189	ASN CRL 失敗を確認する署名者なし
ASN_CRL_NO_SIGNER_E	-190	ASN CRL 失敗を確認する署名者なし
ASN_OCSP_CONFIRM_E	-191	ASN OCSP 署名確認失敗
BAD_ENC_STATE_E	-192	不正な ECC 暗号化状態操作
BAD_PADDING_E	-193	不正なパディング、メッセージ長が不正
REQ_ATTRIBUTE_E	-194	証明書リクエスト属性設定エラー
PKCS7_OID_E	-195	PKCS#7、OID の不一致エラー
PKCS7_RECIP_E	-196	PKCS#7、受信者エラー
FIPS_NOT_ALLOWED_E	-197	FIPS は許可されていません
ASN_NAME_INVALID_E	-198	ASN 名前制約エラー
RNG_FAILURE_E	-199	RNG 失敗、再初期化
HMAC_MIN_KEYLEN_E	-200	FIPS モード HMAC 最小鍵長エラー
RSA_PAD_E	-201	RSA パディングエラー
LENGTH_ONLY_E	-202	出力長のみを返す
IN_CORE_FIPS_E	-203	コア整合性チェック失敗
AES_KAT_FIPS_E	-204	AES KAT 失敗
DES3_KAT_FIPS_E	-205	DES3 KAT 失敗
HMAC_KAT_FIPS_E	-206	HMAC KAT 失敗
RSA_KAT_FIPS_E	-207	RSA KAT 失敗
DRBG_KAT_FIPS_E	-208	HASH DRBG KAT 失敗
DRBG_CONT_FIPS_E	-209	HASH DRBG 継続テスト失敗
AESGCM_KAT_FIPS_E	-210	AESGCM KAT 失敗
THREAD_STORE_KEY_E	-211	スレッドローカルストレージキー作成失敗
THREAD_STORE_SET_E	-212	スレッドローカルストレージキー設定失敗
MAC_CMP_FAILED_E	-213	MAC 比較失敗
IS_POINT_E	-214	ECC 曲線上の点判定失敗
ECC_INF_E	-215	ECC 点無限大エラー
ECC_PRIV_KEY_E	-216	ECC 秘密鍵が無効エラー
SRP_CALL_ORDER_E	-217	SRP 関数が間違った順序で呼び出された
SRP_VERIFY_E	-218	SRP 証明検証失敗
SRP_BAD_KEY_E	-219	SRP 不正な一時値
ASN_NO_SKID	-220	ASN サブジェクト鍵識別子なし
ASN_NO_AKID	-221	ASN 権限鍵識別子なし
ASN_NO_KEYUSAGE	-223	ASN 鍵用途なし

エラーコード列挙型	エラーコード	エラーの説明
SKID_E	-224	サブジェクト鍵識別子設定エラー
AKID_E	-225	権限鍵識別子設定エラー
KEYUSAGE_E	-226	不正な鍵用途値
CERTPOLICIES_E	-227	証明書ポリシー設定エラー
WC_INIT_E	-228	wolfCrypt 初期化失敗
SIG_VERIFY_E	-229	wolfCrypt 署名検証エラー
BAD_PKCS7_SIGNEEDS_CHECKCOND_E	-230	不正な条件変数操作
SIG_TYPE_E	-231	署名タイプが有効でない/利用不可
HASH_TYPE_E	-232	ハッシュタイプが有効でない/利用不可
WC_KEY_SIZE_E	-234	鍵サイズエラー、小さすぎるか大きすぎる
ASN_COUNTRY_SIZE_E	-235	ASN 証明書生成、無効な国コードサイズ
MISSING_RNG_E	-236	RNG が必要だが提供されていない
ASN_PATHLEN_SIZE_E	-237	ASN CA パス長が大きすぎるエラー
ASN_PATHLEN_INV_E	-238	ASN CA パス長反転エラー
BAD_KEYWRAP_ALG_E	-239	キーラップのアルゴリズムエラー
BAD_KEYWRAP_IV_E	-240	復号化された AES キーラップ IV が不正
WC_CLEANUP_E	-241	wolfCrypt クリーンアップに失敗
ECC_CDH_KAT_FIPS_E	-242	ECC CDH 既知解テスト失敗
DH_CHECK_PUB_E	-243	DH 公開鍵チェックエラー
BAD_PATH_ERROR	-244	opendir のパスが不正
ASYNC_OP_E	-245	非同期操作エラー
ECC_PRIVATEONLY_E	-246	秘密専用 ECC 鍵の無効な使用
EXTKEYUSAGE_E	-247	不正な拡張鍵用途値
WC_HW_E	-248	ハードウェア暗号使用のエラー
WC_HW_WAIT_E	-249	ハードウェアがリソース待ち
PSS_SALTLEN_E	-250	PSS の salt の長さがハッシュに対して長すぎる
PRIME_GEN_E	-251	素数発見に失敗
BER_INDEF_E	-252	不定長 BER をデコードできない
RSA_OUT_OF_RANGE_E	-253	復号する暗号文が範囲外
RSAPSS_PAT_FIPS_E	-254	RSA-PSS PAT 失敗
ECDSA_PAT_FIPS_E	-255	ECDSA PAT 失敗
DH_KAT_FIPS_E	-256	DH KAT 失敗
AESCCM_KAT_FIPS_E	-257	AESCCM KAT 失敗
SHA3_KAT_FIPS_E	-258	SHA-3 KAT 失敗
ECDHE_KAT_FIPS_E	-259	ECDHE KAT 失敗
AES_GCM_OVERFLOW_E	-260	AES-GCM 呼び出しカウンターオーバーフロー
AES_CCM_OVERFLOW_E	-261	AES-CCM 呼び出しカウンターオーバーフロー
RSA_KEY_PAIR_E	-262	RSA 鍵ペアワイズ整合性チェック失敗
DH_CHECK_PRIV_E	-263	DH 秘密鍵チェックエラー
WC_AFALG_SOCK_E	-264	AF_ALG ソケットエラー
WC_DEVCRYPTO_E	-265	/dev/crypto エラー
ZLIB_INIT_ERROR	-266	Zlib 初期化エラー
ZLIB_COMPRESS_ERROR	-267	Zlib 圧縮エラー
ZLIB_DECOMPRESS_ERROR	-268	Zlib 解凍エラー
PKCS7_NO_SIGNER_E	-269	PKCS7 署名データメッセージに署名者なし
WC_PKCS7_WANT_READ_E	-270	PKCS7 ストリーム操作でより多くの入力が必要
CRYPTOCB_UNAVAILABLE	-271	暗号コールバック利用不可
PKCS7_SIGNEEDS_CHECK	-272	署名は呼び出し元による検証が必要
ASN_SELF_SIGNED_E	-275	ASN 自己署名証明書エラー
MIN_CODE_E	-300	エラー -101 - -299

F.3 一般的なエラーコードとその解決策

アプリケーションで wolfSSL を使用する際に起こしやすい、いくつかのエラーについてご説明します。

F.3.1 ASN_NO_SIGNER_E (-188)

このエラーは、証明書を使用する際に、署名した CA 証明書が読み込まれていない場合に発生します。これは、wolfSSL のサンプルサーバーやクライアントを、別のクライアントやサーバーに対して使用する場合には見られることがあります。

例えば、wolfSSL のサンプルクライアントを使用して Google に接続する場合、次のように実行したくなるかもしれません。

```
./examples/client/client -g -h www.google.com -p 443
```

しかしこれは、Google の CA 証明書が「-A」コマンドラインオプションで読み込まれていないため、エラー-188 で失敗します。

F.3.2 WANT_READ (-323)

WANT_READ エラーは、ノンブロッキングソケットを使用する場合によく発生します。ノンブロッキングソケットを使用する場合において、実際にはエラーではありませんが、エラーとして出力されます。I/O コールバックからデータを受信する呼び出しが、現在受信可能なデータがないためブロックされる場合、I/O コールバックは WANT_READ を返します。呼び出し元は待機し、後で再度受信を試みてください。これは通常、[wolfSSL_read\(\)](#)、[wolfSSL_negotiate\(\)](#)、[wolfSSL_accept\(\)](#)、および[wolfSSL_connect\(\)](#)の呼び出しから見られます。サンプルクライアントとサーバーは、デバッグ出力が有効である場合に WANT_READ を出力します。

G wolfSSL 対応暗号スイート・鍵交換グループ・署名アルゴリズム一覧

G.1 A. 暗号スイート

G.1.1 A-1. TLS 1.3

IANA 名	Wire ID	デフォルト	configure オプション
TLS_AES_128_GCM_SHA256	0x1301	有効	--enable-tls13 + --enable-aesgcm
TLS_AES_256_GCM_SHA384	0x1302	有効	--enable-tls13 + --enable-aesgcm + --enable-sha384
TLS_CHACHA20_POLY1305_SHA256	0x1303	有効	--enable-tls13 + --enable-chacha + --enable-poly1305
TLS_AES_128_CCM_SHA256	0x1304	無効	--enable-aesccm
TLS_AES_128_CCM_8_SHA256	0x1305	無効	--enable-aesccm
TLS_SM4_GCM_SHA256	0x1306	無効	--enable-sm4-gcm + --enable-sm3
TLS_SM4_CCM_SHA256	0x1307	無効	--enable-sm4-ccm + --enable-sm3
TLS_SHA256_NULL_SHA256	0x1308	無効	--enable-nullcipher
TLS_SHA384_NULL_SHA384	0x1309	無効	--enable-nullcipher + --enable-sha384

G.1.2 A-2. TLS 1.2 ECDHE-ECDSA

IANA 名	Wire ID	デフォルト	configure オプション
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	0xc017	有効	—
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	0xc018	有効	—
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256	0xc019	有効	—
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	0xc01a	有効	—
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	0xc01b	有効	—
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	0xc01c	有効	—
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	0xc01d	有効	—
TLS_ECDHE_ECDSA_WITH_AES_128_CCM	0xc01e	無効	--enable-sha384
TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8	0xc01f	無効	--enable-aesccm
TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8	0xc020	無効	--enable-aesccm
TLS_ECDHE_ECDSA_WITH_DES_EDE_CBC_SHA	0xc021	無効	--enable-des3
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	0xc022	無効	--enable-arc4
TLS_ECDHE_ECDSA_WITH_NULL_SHA	0xc023	無効	--enable-nullcipher
TLS_ECDHE_ECDSA_WITH_SECP256K1	0xc024	無効	--enable-aria
TLS_ECDHE_ECDSA_WITH_SECP384R1	0xc025	無効	--enable-aria
TLS_ECDHE_ECDSA_WITH_SM4_CBC_SM3	0xc026	無効	--enable-sm4-cbc + --enable-sm3 + --enable-sm2
TLS_ECDHE_ECDSA_WITH_SM4_GCM_SM3	0xc027	無効	--enable-sm4-gcm + --enable-sm3 + --enable-sm2
TLS_ECDHE_ECDSA_WITH_SM4_CCM_SM3	0xc028	無効	--enable-sm4-ccm + --enable-sm3 + --enable-sm2

G.1.3 A-3. TLS 1.2 ECDHE-RSA

IANA 名	Wire ID	デフォルト	configure オプション
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	0xC0,0x2F	有効	—
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	0xC0,0x30	有効	—
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256	0xCC,0xA8	有効	—
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	0xC0,0x13	有効	—
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	0xC0,0x14	有効	—
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	0xC0,0x27	有効	—
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	0xC0,0x28	有効	--enable-sha384
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	0xC0,0x12	無効	--enable-des3
TLS_ECDHE_RSA_WITH_RC4_128_SHA	0xC0,0x11	無効	--enable-arc4

G.1.4 A-4. TLS 1.2 DHE-RSA

IANA 名	Wire ID	デフォルト	configure オプション
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	0xC0,0x2F	無効	--enable-dh
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	0xC0,0x30	無効	--enable-dh
TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256	0xCC,0xA8	無効	--enable-dh + --enable-chacha
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	0xC0,0x13	無効	--enable-dh
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	0xC0,0x14	無効	--enable-dh
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	0xC0,0x27	無効	--enable-dh
TLS_DHE_RSA_WITH_AES_256_CBC_SHA384	0xC0,0x28	無効	--enable-dh
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	0xC0,0x12	無効	--enable-dh + --enable-des3
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	0xC0,0x0B	無効	--enable-dh + --enable-camellia
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	0xC0,0x0C	無効	--enable-dh + --enable-camellia
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256	0xC0,0x29	無効	--enable-dh + --enable-camellia
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256	0xC0,0x2A	無効	--enable-dh + --enable-camellia

G.1.5 A-5. TLS 1.2 RSA (静的鍵交換)

IANA 名	Wire ID	デフォルト	configure オプション
TLS_RSA_WITH_AES_128_GCM_SHA256	0x00,0x9C	有効	—
TLS_RSA_WITH_AES_256_GCM_SHA384	0x00,0x9D	有効	—
TLS_RSA_WITH_AES_128_CBC_SHA	0x00,0x2F	有効	—
TLS_RSA_WITH_AES_256_CBC_SHA	0x00,0x35	有効	—
TLS_RSA_WITH_AES_128_CBC_SHA256	0x00,0x3C	有効	—
TLS_RSA_WITH_AES_256_CBC_SHA256	0x00,0x3D	有効	—
TLS_RSA_WITH_AES_128_CCM_8	0xC0,0xA0	無効	--enable-aesccm
TLS_RSA_WITH_AES_256_CCM_8	0xC0,0xA1	無効	--enable-aesccm
SSL_RSA_WITH_3DES_EDE_CBC_SHA	0x00,0x0A	無効	--enable-des3
SSL_RSA_WITH_RC4_128_SHA	0x00,0x05	無効	--enable-arc4
SSL_RSA_WITH_RC4_128_MD5	0x00,0x04	無効	--enable-arc4
TLS_RSA_WITH_NULL_MD5	0x00,0x01	無効	--enable-nullcipher
TLS_RSA_WITH_NULL_SHA	0x00,0x02	無効	--enable-nullcipher

IANA 名	Wire ID	デフォルト	configure オプション
TLS_RSA_WITH_NULL_SHA256	0x00,0x3B	無効	--enable-nullcipher
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	0x00,0x41	無効	--enable-camellia
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	0x00,0x84	無効	--enable-camellia
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256	0x00,0xBA	無効	--enable-camellia
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256	0x00,0xC0	無効	--enable-camellia

G.1.6 A-6. TLS 1.2 静的 ECDH

IANA 名	Wire ID	デフォルト	configure オプション
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	0x00,0x3D	無効	--enable-staticECDH
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	0x00,0x3E	無効	--enable-staticECDH
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA	0x00,0x40	無効	--enable-staticECDH
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA	0x00,0x41	無効	--enable-staticECDH
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256	0x00,0x42	無効	--enable-staticECDH
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384	0x00,0x43	無効	--enable-staticECDH
TLS_ECDH_RSA_WITH_ARC4_128_SHA	0x00,0x44	無効	--enable-staticECDH + --enable-arc4
TLS_ECDH_RSA_WITH_ARC4_256_SHA	0x00,0x45	無効	--enable-staticECDH + --enable-arc4
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256	0x00,0x3F	無効	--enable-staticECDH
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384	0x00,0x40	無効	--enable-staticECDH
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA	0x00,0x41	無効	--enable-staticECDH
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	0x00,0x42	無効	--enable-staticECDH
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	0x00,0x43	無効	--enable-staticECDH
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	0x00,0x44	無効	--enable-staticECDH
TLS_ECDH_ECDSA_WITH_ARC4_128_SHA	0x00,0x45	無効	--enable-staticECDH + --enable-arc4
TLS_ECDH_ECDSA_WITH_ARC4_256_SHA	0x00,0x46	無効	--enable-staticECDH + --enable-arc4

G.1.7 A-7. TLS 1.2 PSK

IANA 名	Wire ID	デフォルト	configure オプション
TLS_PSK_WITH_AES_128_GCM_SHA256	0x00,0x47	無効	--enable-psk
TLS_PSK_WITH_AES_256_GCM_SHA384	0x00,0x48	無効	--enable-psk
TLS_PSK_WITH_AES_128_CBC_SHA256	0x00,0x49	無効	--enable-psk
TLS_PSK_WITH_AES_256_CBC_SHA384	0x00,0x4A	無効	--enable-psk
TLS_PSK_WITH_AES_128_CBC_SHA	0x00,0x4B	無効	--enable-psk
TLS_PSK_WITH_AES_256_CBC_SHA	0x00,0x4C	無効	--enable-psk
TLS_PSK_WITH_AES_128_GCM	0x00,0x4D	無効	--enable-psk + --enable-aesccm
TLS_PSK_WITH_AES_256_GCM	0x00,0x4E	無効	--enable-psk + --enable-aesccm
TLS_PSK_WITH_AES_128_GCM_8	0x00,0x4F	無効	--enable-psk + --enable-aesccm
TLS_PSK_WITH_AES_256_GCM_8	0x00,0x50	無効	--enable-psk + --enable-aesccm
TLS_PSK_WITH_CHACHA20_POLY1305_SHA256	0x00,0x51	無効	--enable-psk + --enable-chacha
TLS_PSK_WITH_NULL_SHA256	0x00,0x52	無効	--enable-psk + --enable-nullcipher
TLS_PSK_WITH_NULL_SHA384	0x00,0x53	無効	--enable-psk + --enable-nullcipher

IANA 名	Wire ID	デフォルト	configure オプション
TLS_PSK_WITH_NULL_SHA256	0x00,0x2C	無効	--enable-psk + --enable-nullcipher

G.1.8 A-8. TLS 1.2 DHE-PSK

IANA 名	Wire ID	デフォルト	configure オプション
TLS_DHE_PSK_WITH_AES_128_GCM_SHA256	0x00,0x3A	無効	--enable-psk + --enable-dh
TLS_DHE_PSK_WITH_AES_256_GCM_SHA384	0x00,0x3B	無効	--enable-psk + --enable-dh
TLS_DHE_PSK_WITH_AES_128_CBC_SHA256	0x00,0x3C	無効	--enable-psk + --enable-dh
TLS_DHE_PSK_WITH_AES_256_CBC_SHA384	0x00,0x3D	無効	--enable-psk + --enable-dh
TLS_DHE_PSK_WITH_AES_128_CCM	0x00,0x3E	無効	--enable-psk + --enable-dh + --enable-aesccm
TLS_DHE_PSK_WITH_AES_256_CCM	0x00,0x3F	無効	--enable-psk + --enable-dh + --enable-aesccm
TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256	0x00,0x40	無効	--enable-psk + --enable-dh + --enable-chacha
TLS_DHE_PSK_WITH_NULL_SHA256	0x00,0x41	無効	--enable-psk + --enable-dh + --enable-nullcipher
TLS_DHE_PSK_WITH_NULL_SHA384	0x00,0x42	無効	--enable-psk + --enable-dh + --enable-nullcipher

G.1.9 A-9. TLS 1.2 ECDHE-PSK

IANA 名	Wire ID	デフォルト	configure オプション
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256	0x00,0x43	無効	--enable-psk + --enable-supportedcurves
TLS_ECDHE_PSK_WITH_AES_256_GCM_SHA384	0x00,0x44	無効	--enable-psk + --enable-supportedcurves
TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256	0x00,0x45	無効	--enable-psk + --enable-chacha
TLS_ECDHE_PSK_WITH_NULL_SHA256	0x00,0x46	無効	--enable-psk + --enable-nullcipher

G.1.10 A-10. TLS 1.2 DH_anon

IANA 名	Wire ID	デフォルト	configure オプション
TLS_DH_anon_WITH_AES_128_CBC_SHA	0x00,0x34	無効	--enable-anon
TLS_DH_anon_WITH_AES_256_GCM_SHA384	0x00,0xA7	無効	--enable-anon

G.2 B. 鍵交換グループ**G.2.1 B-1. NIST 楕円曲線**

グループ名	ID	デフォルト	configure オプション
secp256r1 (P-256)	23	有効	--enable-ecc
secp384r1 (P-384)	24	有効	--enable-ecc
secp521r1 (P-521)	25	有効	--enable-ecc
secp192r1 (P-192)	19	無効	--enable-ecc (TLS 1.2 のみ)
secp224r1 (P-224)	21	無効	--enable-ecc (TLS 1.2 のみ)

G.2.2 B-2. Koblitz 曲線

グループ名	ID	デフォルト	configure オプション
secp160k1	15	無効	--enable-ecccustcurves
secp192k1	18	無効	--enable-ecccustcurves
secp224k1	20	無効	--enable-ecccustcurves
secp256k1	22	無効	--enable-ecccustcurves

G.2.3 B-3. Brainpool 曲線

グループ名	ID	TLS 版	デフォルト	configure オプション
brainpoolP256r1	26	1.2	無効	--enable-ecccustcurves + --enable-brainpool
brainpoolP384r1	27	1.2	無効	--enable-ecccustcurves + --enable-brainpool
brainpoolP512r1	28	1.2	無効	--enable-ecccustcurves + --enable-brainpool
brainpoolP256r1tls13	31	1.3	無効	--enable-ecccustcurves + --enable-brainpool
brainpoolP384r1tls13	32	1.3	無効	--enable-ecccustcurves + --enable-brainpool
brainpoolP512r1tls13	33	1.3	無効	--enable-ecccustcurves + --enable-brainpool

G.2.4 B-4. X25519 / X448

グループ名	ID	デフォルト	configure オプション
x25519	29	無効	--enable-curve25519
x448	30	無効	--enable-curve448

G.2.5 B-5. FFDHE

グループ名	ID	デフォルト	configure オプション
ffdhe2048	256	条件付き有効	--enable-dh (TLS 1.3+DH 有効時に自動設定)
ffdhe3072	257	無効	-DHAVE_FFDHE_3072
ffdhe4096	258	無効	-DHAVE_FFDHE_4096
ffdhe6144	259	無効	-DHAVE_FFDHE_6144
ffdhe8192	260	無効	-DHAVE_FFDHE_8192

G.2.6 B-6. SM2

グループ名	ID	デフォルト	configure オプション
sm2p256v1	41	無効	--enable-sm2

G.2.7 B-7. ML-KEM スタンドアロン

グループ名	ID	デフォルト	configure オプション
ML-KEM-512	512	無効	--enable-mlkem + --enable-tls-mlkem-standalone
ML-KEM-768	513	無効	--enable-mlkem + --enable-tls-mlkem-standalone
ML-KEM-1024	514	無効	--enable-mlkem + --enable-tls-mlkem-standalone

G.2.8 B-8. ML-KEM ハイブリッド (標準)

グループ名	ID	デフォルト	configure オプション
SecP256r1MLKEM768	4587	条件付き有効	--enable-mlkem + --enable-pqc-hybrids
X25519MLKEM768	4588	無効	--enable-mlkem + --enable-pqc-hybrids + --enable-curve25519
SecP384r1MLKEM1024	4589	条件付き有効	--enable-mlkem + --enable-pqc-hybrids

G.2.9 B-9. ML-KEM ハイブリッド (実験的)

グループ名	ID	デフォルト	configure オプション
SecP256r1MLKEM512	12107	無効	--enable-extra-pqc-hybrids + --enable-experimental
SecP384r1MLKEM768	12108	無効	--enable-extra-pqc-hybrids + --enable-experimental
SecP521r1MLKEM1024	12109	無効	--enable-extra-pqc-hybrids + --enable-experimental
X25519MLKEM512	12214	無効	同上 + --enable-curve25519
X448MLKEM768	12215	無効	同上 + --enable-curve448

G.2.10 B-10. Kyber Round 3 レガシー

グループ名	ID	デフォルト	configure オプション
KYBER_LEVEL1	570	無効	--enable-mlkem=original
KYBER_LEVEL3	572	無効	--enable-mlkem=original
KYBER_LEVEL5	573	無効	--enable-mlkem=original
P256_KYBER_LEVEL1	12090	無効	--enable-mlkem=original
P384_KYBER_LEVEL3	12092	無効	--enable-mlkem=original
P521_KYBER_LEVEL5	12093	無効	--enable-mlkem=original
X25519_KYBER_LEVEL1	12089	無効	同上 + --enable-curve25519
X25519_KYBER_LEVEL3	25497	無効	同上 + --enable-curve25519
X448_KYBER_LEVEL3	12176	無効	同上 + --enable-curve448

グループ名	ID	デフォルト	configure オプション
P256_KYBER_LEVEL3	25498	無効	--enable-mlkem=original

G.3 C. 署名アルゴリズム

G.3.1 C-1. ECDSA (NIST 曲線)

IANA 名	Wire Value	デフォルト	configure オプション
ecdsa_secp256r1_sha256	0x04,0x03	有効	--enable-ecc
ecdsa_secp384r1_sha384	0x05,0x03	有効	--enable-ecc + --enable-sha384
ecdsa_secp521r1_sha512	0x06,0x03	有効	--enable-ecc + --enable-sha512
ecdsa_sha1	0x02,0x03	条件付き	!NO_OLD_TLS または WOLFSSL_ALLOW_TLS_SHA1

G.3.2 C-2. ECDSA (Brainpool, TLS 1.3 専用)

IANA 名	Wire Value	デフォルト	configure オプション
ecdsa_brainpoolp256r1_sha256	0x04,0x13	無効	--enable-eccustcurves + --enable-brainpool
ecdsa_brainpoolp384r1_sha384	0x05,0x13	無効	同上 + --enable-sha384
ecdsa_brainpoolp512r1_sha512	0x06,0x13	無効	同上 + --enable-sha512

G.3.3 C-3. EdDSA

IANA 名	Wire Value	デフォルト	configure オプション
ed25519	0x08,0x07	無効	--enable-ed25519
ed448	0x08,0x08	無効	--enable-ed448

G.3.4 C-4. RSA-PSS RSAE

IANA 名	Wire Value	デフォルト	configure オプション
rsa_pss_rsae_sha256	0x08,0x04	有効	--enable-rsapss (TLS 1.3 時自動)
rsa_pss_rsae_sha384	0x08,0x05	有効	--enable-rsapss + --enable-sha384
rsa_pss_rsae_sha512	0x08,0x06	有効	--enable-rsapss + --enable-sha512

G.3.5 C-5. RSA-PSS PSS (TLS 1.3 専用)

IANA 名	Wire Value	デフォルト	configure オプション
rsa_pss_pss_sha256	0x08,0x09	有効	--enable-tls13 + --enable-rsapss
rsa_pss_pss_sha384	0x08,0x0A	有効	同上 + --enable-sha384
rsa_pss_pss_sha512	0x08,0x0B	有効	同上 + --enable-sha512

G.3.6 C-6. RSA PKCS#1 v1.5

IANA 名	Wire Value	デフォルト	configure オプション
rsa_pkcs1_sha256	0x04,0x01	有効	—
rsa_pkcs1_sha384	0x05,0x01	有効	--enable-sha384
rsa_pkcs1_sha512	0x06,0x01	有効	--enable-sha512
rsa_pkcs1_sha224	0x03,0x01	条件付き	--enable-sha224 (x86_64/aarch64 でデフォルト有効)
rsa_pkcs1_sha1	0x02,0x01	条件付き	!NO_OLD_TLS または WOLFSSL_ALLOW_TLS_SHA1

G.3.7 C-7. SM2

IANA 名	Wire Value	デフォルト	configure オプション
sm2sig_sm3	0x07,0x08	無効	--enable-sm2 + --enable-sm3

G.3.8 C-8. ML-DSA / Dilithium

IANA 名	Wire Value	デフォルト	configure オプション
mldsa44 (ML-DSA-44)	0x09,0x04	無効	--enable-mldsa
mldsa65 (ML-DSA-65)	0x09,0x05	無効	--enable-mldsa
mldsa87 (ML-DSA-87)	0x09,0x06	無効	--enable-mldsa

G.4 D. configure オプション □ #define 対応表

configure オプション	#define マクロ	備考
--enable-tls13	WOLFSSL_TLS13	デフォルト有効
--enable-ecc	HAVE_ECC	デフォルト有効
--enable-aesgcm	HAVE_AESGCM	デフォルト有効
--enable-chacha	HAVE_CHACHA	デフォルト有効
--enable-poly1305	HAVE_POLY1305	デフォルト有効
--enable-sha384	WOLFSSL_SHA384	デフォルト有効
--enable-sha512	WOLFSSL_SHA512	デフォルト有効
--enable-supportedcurves	HAVE_SUPPORTED_CURVES + HAVE_TLS_EXTENSIONS	デフォルト有効
--enable-aesccm	HAVE_AESCCM	
--enable-dh	HAVE_DH	
--enable-psk	HAVE_PSK	
--enable-rsapss	WC_RSA_PSS	TLS 1.3 時自動

configure オプション	#define マクロ	備考
--enable-curve25519	HAVE_CURVE25519	
--enable-curve448	HAVE_CURVE448	
--enable-ed25519	HAVE_ED25519	
--enable-ed448	HAVE_ED448	
--enable-eccustcurves	WOLFSSL_CUSTOM_CURVES	
--enable-brainpool	HAVE_ECC_BRAINPOOL	eccustcurves 必須
--enable-des3	NO_DES3 を除去	
--enable-arc4	NO_RC4 を除去	
--enable-camellia	HAVE_CAMELLIA	
--enable-aria	HAVE_ARIA	
--enable-nullcipher	HAVE_NULL_CIPHER	非推奨
--enable-anon	HAVE_ANON	非推奨
--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	mlkem 有効 時デフォルト 有効
--enable-mldsa	HAVE_DILITHIUM	
--enable-staticECDH	WOLFSSL_STATIC_ECDH	PFS 無し, 非 推奨

H wolfSSL ハードウェア最適化サポート対応表

H.1 A. アーキテクチャ別 ISA/SIMD 最適化

アーキテクチャ	アルゴリズム	user_settings.h マクロ	configure オプション	備考
Intel x86-64	AES (全モード)	WOLFSSL_AESNI	--enable-aesni	AES-NI 命令
Intel x86-64	SHA-256, SHA-512	USE_INTEL_SPEEDY + HAVE_INTEL_AVX1	自動検出	AVX1 アセンブリ
Intel x86-64	SHA-512, SHA-3	USE_INTEL_SPEEDY + HAVE_INTEL_AVX2	自動検出	AVX2 アセンブリ
Intel x86-64	ChaCha20, Poly1305	USE_INTEL_SPEEDY	自動検出	
Intel x86-64	X25519 (Curve25519), Ed25519	USE_INTEL_SPEEDY	自動検出	fe_x25519_asm.S
Intel x86-64 ARM AArch64	RSA/ECC/DH (SP) AES, SHA-256/512, SHA-3, ChaCha20, Poly1305	WOLFSSL_SP_X86_64 WOLFSSL_ARMASM	自動検出 --enable-armasm	sp_x86_64_asm.S ARMv8 CE (Crypto Extension)
ARM AArch64	X25519 (Curve25519), Ed25519	WOLFSSL_ARMASM	--enable-armasm	armv8- curve25519.S
ARM AArch64 ARM 32-bit (ARMv8-32)	RSA/ECC/DH (SP) AES, SHA-256/512, SHA-3, ChaCha20, Poly1305	WOLFSSL_SP_ARM64 WOLFSSL_ARMASM	自動検出 --enable-armasm	sp_arm64.c
ARM 32-bit	X25519 (Curve25519), Ed25519	WOLFSSL_ARMASM	--enable-armasm	armv8-32- curve25519.S
ARM 32-bit ARM Thumb2	RSA/ECC/DH (SP) X25519 (Curve25519), Ed25519	WOLFSSL_SP_ARM32 WOLFSSL_ARMASM + WOLF- SSL_ARMASM_THUMB2	自動検出 --enable-armasm	sp_arm32.c thumb2- curve25519.S
ARM Thumb ARM Cortex-M RISC-V 64-bit	RSA/ECC/DH (SP) RSA/ECC/DH (SP) AES, SHA-256/512, SHA-3, ChaCha20, Poly1305	WOLFSSL_SP_ARM_THUMB WOLFSSL_SP_ARM_CortexM_ASM WOLFSSL_RISCV_ASM	自動検出 自動検出 enable-riscv	sp_armthumb.c sp_cortexm.c riscv/riscv-64-*.c
PowerPC 32-bit	SHA-256	WOLFSSL_PPC32_ASM	自動検出	ppc32/ppc32- sha256-asm.S

H.2 B. SP 数学ライブラリ（公開鍵暗号）

アーキテクチャ	対応アルゴリズム	user_settings.h マクロ	configure オプション
x86-64 アセンブリ	RSA 2048/3072/4096, ECC P-256/P-384/P-521, DH	WOLFSSL_SP_X86_64_ASM	自動検出
ARM AArch64 アセンブリ	RSA, ECC, DH	WOLFSSL_SP_ARM64_ASM	自動検出
ARM 32-bit アセンブリ	RSA, ECC, DH	WOLFSSL_SP_ARM32_ASM	自動検出
ARM Thumb アセンブリ	RSA, ECC, DH	WOLFSSL_SP_ARM_THUMB_ASM	自動検出
Cortex-M アセンブリ	RSA, ECC, DH	WOLFSSL_SP_ARM_CORTX_ASM	自動検出
汎用 C (32-bit / 64-bit)	RSA, ECC, DH	WOLFSSL_SP_MATH	--enable-sp

H.2.1 sp_int.c インラインアセンブリ高速化

sp_int.c には、アーキテクチャ専用 SP ファイル（例：sp_x86_64.c）が存在しない場合や、RSA/ECC 以外の鍵長に対応するため、多倍長整数の乗算・加算向けの小規模インラインアセンブリが含まれています。以下のマクロで制御されます（自動検出、または user_settings.h で明示設定可）：

アーキテクチャ	user_settings.h マクロ
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 鍵サイズ設定

```
#define WOLFSSL_SP_MATH
#define WOLFSSL_SP_X86_64_ASM /* x86-64 */
#define WOLFSSL_SP_ARM64_ASM /* AArch64 */
#define WOLFSSL_SP_ARM_CORTX_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. マイコン/SoC チップ別ハードウェアアクセラレータ

ベンダー / チップ	対応アルゴリズム	user_settings.h マクロ	configure オプシ ョン	ソース
STMicroelectronics				
STM32F2/F4/F7/H7	AES- (CRYP) ECB/CBC/CTR/GCM	STM32_CRYPT0	HAL 自動検出	port/st/stm32.c
STM32L4/L5/WB55	AES, SHA (部分)	STM32_CRYPT0	HAL 自動検出	port/st/stm32.c
STM32U5	鍵ラップ付き AES	WOLFSSL_STM32U5	自動検出	port/st/stm32.c
(DHUK)				
STM32MP1/MP13/MP25	AES, SHA, RSA, ECC	WOLFSSL_STM32_CRYPT0	自動検出	port/st/stm32.c
NXP				
i.MX RT (DCP)	AES- ECB/CBC/CTR/GCM, SHA	WOLFSSL_IMXRT_DCP	自動検出	port/nxp/dcp_port.c
i.MX 6/8 (CAAM)	AES, SHA, RSA, ECC, DH, HMAC	WOLFSSL_IMX6_CAAM / WOLF- SSL_SECO_CAAM	enable-caam	port/caam/*
SE050 セキュアエ レメント	AES, SHA, RSA, ECC, X25519, ECDH	WOLFSSL_SE050 + WOLF- SSL_SE050_CRYPT	自動検出	port/nxp/se050_port.c
Espressif				
ESP32 / ESP32-S2 / ESP32-S3	AES, SHA- 1/256/384/512, RSA, ECC	WOLFSSL_ESP32_CRYPT enable- esp32	自動検出	port/Espressif/esp32_*.c
ESP32-C3 / C6 / H2	AES, SHA- 256/384/512 (部 分)	WOLFSSL_ESP32_CRYPT enable- esp32	自動検出	port/Espressif/esp32_*.c
Renesas				
RX64M/RX71M/RX72M	SHA-1, SHA-256	WOLFSSL_RENESAS_SHA1	自動検出	port/Renesas/renesas_rx64_*.c
RX65N (SCE)	AES, SHA, RSA, ECC, RNG	WOLFSSL_RENESAS_SHA256	自動検出	port/Renesas/renesas_*.c
RZ/A1/A2 (TSIP)	AES, SHA, RSA, ECC, RNG	WOLFSSL_RENESAS_TSIP	自動検出	port/Renesas/renesas_tsip_*.c
RA4M4/RA6M4 (RSIP)	AES, SHA, RSA, ECC, RNG	WOLFSSL_RENESAS_RSIP	自動検出	port/Renesas/renesas_rsip*.c
RA FSP Secure Manager	AES, SHA, RSA, ECC, HMAC	WOLFSSL_RENESAS_FSP	自動検出	port/Renesas/renesas_fspsm_*.c
Cypress				
PSoC 6	AES, SHA, ECC, RNG	WOLFSSL_PSO6_CRYPT0	自動検出	port/cypress/psoc6_crypto.c
Silicon Labs				
EFR32 / EFM32 (SE)	AES, SHA, ECC, RNG	WOLFSSL_SILABS_SHA256	自動検出	port/silabs/silabs_*.c
Maxim				
MAX3266X	AES-ECB/CBC, RSA	WOLFSSL_MAX3266X_CRYPT0	自動検出	port/maxim/max3266x.c
MAXQ10XX	AES, SHA, RSA, ECC, RNG	WOLFSSL_MAXQ10XX_CRYPT0 maxq10xx=PART	自動検出	port/maxim/maxq10xx.c
Microchip				
PIC32MZ	AES, SHA, RNG	WOLFSSL_PIC32MZ_CRYPT0	自動検出	port/pic32/
ATECC508A / 608A	AES, SHA, ECC (ECDH/ECDSA)	WOLFSSL_ATECC	自動検出	port/atmel/atmel.c
Xilinx				

ベンダー / チップ	対応アルゴリズム	user_settings.h マクロ	configure オプション	ソース
Zynq / Zynq-7000	AES, SHA, RSA, ECC	WOLFSSL_XILINX_CRYPTABLE- xilinx	--enable- xilinx	port/xilinx/xil-*.c
Versal	AES-GCM, SHA-3, RNG	WOLFSSL_XILINX_CRYPTABLE- xilinx	--enable- xilinx	port/xilinx/xil- versal-*.c
ARM CryptoCell				
CC- 300/312/700/712/713	AES, SHA, RSA, ECC, ChaCha, Poly1305	WOLFSSL_CRYPTOCCELL- cryptocell	--enable- cryptocell	port/arm/cryptocell*.c
IoT-Safe (SIM)	ECC, SHA, RNG	WOLFSSL_IOTSAFE	--enable- iotsafe	port/iotsafe/

H.4 D. 外部アクセラレータ・コプロセッサ

製品	対応アルゴリズム	user_settings.h マクロ	configure オプション
Intel QuickAssist (QAT)	AES, RSA, DH, ECC, SHA	HAVE_INTEL_QA	--with- intelqa=PATH
Intel QAT (同期)	同上	ENABLED_INTEL_QA_SYNC	--with-intelqa- sync=PATH
Cavium NITROX	AES, RSA, DH, ECC, SHA	HAVE_CAVIUM	--with- cavium=PATH
Cavium NITROX V	同上	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 カーネル経由ハードウェアオフロード

インターフェース	対応アルゴリズム	user_settings.h マクロ	configure オプション
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 (カーネルバ ックエンド依存)	WOLFSSL_AFALG	--enable-afalg
PSA (Platform Security Architecture)	ベンダー実装依存	WOLFSSL_HAVE_PSA	--enable-psa

I ポスト量子暗号の実験

wolfSSL チームは以前、実験的なポスト量子暗号アルゴリズムを wolfSSL ライブラリに統合しました。これは、Open Quantum Safe チームの liboqs と統合することで実現されました。現在、wolfCrypt は LMS、XMSS、ML-DSA、ML-KEM を実装しています。コードサイズの削減と保守の容易さのため、wolfSSL チームは liboqs との統合を削除しました。

この付録は、(D)TLS 1.3 のコンテキストでポスト量子暗号について学び、実験したい人を対象としています。ポスト量子アルゴリズムが重要である理由、量子の脅威に対応して私たちが行ったこと、これらの新しいアルゴリズムの実験を開始する方法について説明します。

注意: 一部のポスト量子アルゴリズムはまだ完全に標準化されていません。一部の OID とコードポイントは一時的なものであり、将来変更される可能性があります。完全に標準化されるまでは、下位互換性を期待すべきではありません。

I.1 ポスト量子暗号をわかりやすく紹介

I.1.1 なぜポスト量子暗号？

今日では、量子コンピューターの開発にますます多くのリソースが割かれるようになりました。そのため、クラウド量子コンピューティングリソースの商業化がすでに始まっています。現時点では未だ実用レベルの暗号が解かれた報告はありません。しかし、「あらかじめデータを収集、蓄積し、後に時間をかけて解読を進めていく」といった脅威モデルが存在します。すなわち、暗号の復号に特化した量子コンピューターが出現するよりも早いうちに準備が必要です。

NIST は量子コンピュータに対して脆弱になる公開鍵暗号アルゴリズムを置き換えるために設計された、新しいアルゴリズムの標準化を主導しています。この文章の執筆時点において、NIST はすでに ML-DSA、ML-KEM、SLH-DSA を標準化しました。

ML-KEM (Module Lattice Key Encapsulation Mechanism) は、NIST が標準化した格子ベースの耐量子アルゴリズムです。Kyber をベースとして標準化されました。2 者間で鍵カプセル化メカニズムを使用して、安全でないチャネル上で共有鍵を確立します。これにより、従来型および量子コンピュータを用いた攻撃者の両方から保護できます。

ML-DSA (Module Lattice Digital Signature Algorithm) は、同じく NIST が標準化した格子ベースの耐量子デジタル署名方式です。Kyber をベースとして標準化されました。これにより、送信者はメッセージの送信元と完全性を証明する検証可能な署名を生成できます。

ML-KEM と ML-DSA はどちらも、暗号の復号に特化した量子コンピュータに耐性を持つように設計された公開鍵アルゴリズムです。これらは NIST の耐量子暗号標準 (FIPS 203 および FIPS 204) の一部です。量子コンピューティング時代に備えるために、ほとんどのケースでハイブリッド形式にてすぐに導入できます。

現在、各標準化団体は OID とコードポイントを示すさまざまなドラフト文書を有しています。NIST はこれらのアルゴリズムを CMVP 規制フレームワークの傘下に置き、実装に対する FIPS-140-3 検証を可能にする取り組みを進めています。

I.1.2 私たちは自分自身をどのように守るのですか？

大まかに言えば、すべての TLS 1.3 接続において、認証・完全性・機密性は各接続を保護する重要な目標です。認証は、ECDSA などの署名スキームによって維持されます。機密性と完全性は、ECDHE などのキー確立アルゴリズムによって維持され、確立されたキーと AES などの対称暗号化アルゴリズムを使用して通信ストリームを暗号化します。したがって、TLS 1.3 プロトコルのセキュリティは、次の 3 種類の暗号化アルゴリズムに分解できます。

- 認証アルゴリズム
- 鍵確立アルゴリズム
- 対称暗号化アルゴリズム

量子コンピュータが従来の暗号に及ぼす脅威には、2つの形態があります。グローバーのアルゴリズムは、最新の対称暗号アルゴリズムのセキュリティを半分に低下させます。ショアのアルゴリズムは、最新の認証および鍵確立アルゴリズムのセキュリティを完全に破壊します。したがって、暗号処理に特化した量子コンピュータが存在する場合でも、十分に安全と考えられる AES-256 対称暗号を使用して通信を保護し続けることができます。そして、従来の認証および鍵確立アルゴリズムをポスト量子アルゴリズムに置き換えることができます。TLS 1.3 ハンドシェイク中、暗号スイートは接続の間使用される対称暗号を指定することに注意してください。CNSA（商用国家安全保障アルゴリズムスイート）1.0 と 2.0 はともに AES_256_GCM_SHA384 暗号スイートの使用を規定しています。鍵確立と認証については、ポスト量子 KEM（鍵カプセル化メカニズム）と署名スキームがあります。

これらは従来のアルゴリズムとは異なる種類の数学を使用します。量子コンピュータに対する耐性を特に考慮して設計されたものです。NIST がネットワークプロトコルでの使用のために標準化した認証アルゴリズムと KEM は、格子ベースのアルゴリズムです。

- ML-DSA（Dilithium）署名スキーム
- ML-KEM（KYBER）KEM

注意: SABER KEM と NTRU KEM は標準化に進まなかったため、非推奨となり削除されました。

注意: KYBER KEM 90s バリエントは NIST が標準化を検討していないため、非推奨となり削除されました。

注意: Dilithium 署名スキームの AES バリエントは NIST が標準化を検討していないため、非推奨となり削除されました。

注意: liboqs 統合が削除された際、FALCON と SPHINCS+ 署名スキームも削除されました。今後、独自の実装を提供する予定です。

格子ベースの暗号化の説明はこの文書の範囲を超えますが、これらのアルゴリズムに関する詳細情報は NIST 提出物で確認できます。

<https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions>

残念ながら、この章を記している時点においてこれらのアルゴリズムが量子コンピューターからの攻撃に耐えられるかどうかは、まだ分かっていません。従来型コンピューターにおける攻撃耐性も同様です。可能性はますます低くなっていますが、誰かが格子ベースの暗号化を破る可能性があります。ただし、暗号化は常にこのような経緯で機能してきた歴史があります。アルゴリズムは使い始めたときは優れていますが、弱点や脆弱性が発見され、テクノロジーは順次改善されます。ポスト量子アルゴリズムは比較的新しいため、コミュニティからもう少し注目を集める必要があるかもしれません。

解決策の 1 つは、これらの新しいアルゴリズムを完全には信頼しないことです。今のところ、ポスト量子 KEM を、信頼している従来のアルゴリズムとハイブリッド化することで、このリスクを回避することができます。FIPS 準拠を第一に考えると、NIST 標準曲線を使用した ECC は良い選択肢になります。このため、ポスト量子 KEM を統合するだけでなく、NIST が承認した曲線上の ECDSA とハイブリッド化しました。詳しくは以下のハイブリッドグループのリストを参照してください。

I.2 wolfSSL のでポスト量子暗号を始めましょう

ここではまっさらな Linux 環境をもとに、量子耐性を有する安全な TLS 1.3 接続を実行できるようにするまでの手順を示します。

I.2.1 ビルド手順

wolfSSL リポジトリの [INSTALL ファイル](#) をご参照ください。

項目 15 には、構成とビルドの方法に関する説明があります。

ポスト量子暗号鍵と署名を使用して X.509 証明書を生成するには、パッチを適用した OQS OpenSSL Provider フォークが必要です。手順は <https://github.com/wolfSSL/osp/tree/master/oqs/README.md> にあります。事前に生成された証明書もそこで入手できます。

I.2.2 量子安全な TLS 接続を確立する

次のようにして、サーバーとクライアントを別々のターミナルで実行します。

```
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
```

これで、対称暗号化に AES-256、認証に ML-DSA 署名スキーム、鍵確立に ECDHE と ML-KEM をハイブリッド化した完全な量子安全な TLS 1.3 接続を実現しました。

その他のポスト量子の例に関する詳細情報は <https://github.com/wolfSSL/wolfssl-examples/blob/master/pq/README.md> で確認できます。

I.3 暗号アーティファクトのサイズ

すべてのサイズはバイト単位です。

I.3.1 ポスト量子署名スキームのアーティファクトサイズ

PQC バリエーション名	公開鍵サイズ	秘密鍵サイズ	最大署名サイズ
ML_DSA_44	1312	2560	2420
ML_DSA_65	1952	4032	3309
ML_DSA_87	2592	4896	4627

I.3.2 ポスト量子 KEM のアーティファクトサイズ

PQC バリエーション名	公開鍵サイズ	秘密鍵サイズ	暗号文サイズ	共有秘密サイズ
ML_KEM_512	800	1632	768	32
ML_KEM_768	1184	2400	1088	32
ML_KEM_1024	1568	3168	1568	32

I.4 統計データ

以下の統計とベンチマークは、8 コアの Ubuntu 22.04.5 LTS を実行している第 11 世代 Intel Core i7-1185G7@3-GHz で取得されました。

```
./configure --enable-kyber \
  --enable-dilithium \
  --disable-psk \
  --disable-shared \
  --enable-intelasm \
  --enable-aesni \
  --enable-sp-math-all \
```



```
--enable-sp-asm \
CFLAGS="-Os"
```

注：主に耐量子アルゴリズムをベンチマークしていますが、比較目的のために従来のアルゴリズムも残しています。

I.4.1 実行時バイナリサイズ

tls_bench サンプルアプリケーションのバイナリファイルは、ビルド後にストリップすると 2498432 バイト (約 2.4M) です。--enable-kyber --enable-dilithium なしでは、ビルド後にストリップすると 2290912 バイト (約 2.2M) です。これは 207520 バイト (約 200K) の差です。

I.4.2 TLS 1.3 データ送信サイズ

サンプルサーバーとクライアントを実行し、送信されるすべての情報を Wireshark で記録することによって取得した値を以下に示します。これには、相互認証による TLS 1.3 ハンドシェイク、“hello wolfssl!”、“I hear you fa shizzle!” メッセージが含まれます。すべてのパケットの tcp.len を合計しました。

暗号スイート	認証	鍵確立	合計バイト
TLS_AES_256_GCM_SHA384	RSA 2048 ビット	ECC SECP256R1	5455
TLS_AES_256_GCM_SHA384	RSA 2048 ビット	ML_KEM_512	6633
TLS_AES_256_GCM_SHA384	RSA 2048 ビット	ML_KEM_768	7337
TLS_AES_256_GCM_SHA384	RSA 2048 ビット	ML_KEM_1024	8201
TLS_AES_256_GCM_SHA384	RSA 2048 ビット	P256_ML_KEM_512	6763
TLS_AES_256_GCM_SHA384	RSA 2048 ビット	P384_ML_KEM_768	7531
TLS_AES_256_GCM_SHA384	RSA 2048 ビット	P521_ML_KEM_1024	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.4.3 ヒープとスタックの使用量

注意：これは古い情報です。これらは wolfSSL がこれらのアルゴリズムの liboqs 実装を使用していた時に取得されたものです。歴史的な目的のために残されています。

これらの統計は、次の構成フラグを追加して取得しました。

```
--enable-trackmemory --enable-stacksize
```

クライアントのサーバー認証なしのサーバー署名とクライアント検証、鍵交換用の TLS13-AES256-GCM-SHA384 暗号スイートおよび ECC SECP256R1 におけるメモリ使用量を以下に示します。

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
```

KEM グループのメモリ使用量を示します。サーバーのクライアント認証には TLS13-AES256-GCM-SHA384 と RSA-2048 を使用し、クライアントのサーバー認証は行いません。

```
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.4.4 ベンチマーク

I.4.4.1 wolfCrypt によるベンチマーク 以下のベンチマークは次の設定フラグを使用して取得されました。

```
./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"
```

注意: 1つのコアのみが使用されています。

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.4.4.2 wolfSSL によるベンチマーク 以下のベンチマークは次の設定フラグを使用して取得されました。

```
./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"
```

注意: これらのベンチマークには2つのコアのみが使用されています。

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 P256_ML_KEM_512
:

```
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 P256_ML_KEM_512
:

```
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 P384_ML_KEM_768
:

```
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 P384_ML_KEM_768
:

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
P521_ML_KEM_1024:

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
P521_ML_KEM_1024:

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 P256_ML_KEM_512
:

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 P256_ML_KEM_512
:

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 P384_ML_KEM_768
:

```

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 P384_ML_KEM_768
:

```

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
P521_ML_KEM_1024:

```

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
P521_ML_KEM_1024:

```

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.5 ドキュメンテーション

技術文書や既知の回答テストなどのその他のリソースは、NIST PQC Web サイトにあります。

<https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions>

アルゴリズム固有のベンチマーク情報については、OQS プロジェクトの Web サイトに掲載されています。

<https://openquantumsafe.org/benchmarking/>

I.6 ポスト量子ステートフルハッシュベース署名

このセクションでは、最近 wolfSSL がサポートを開始した LMS/HSS などのポスト量子ステートフルハッシュベース署名 (HBS) スキームについて記します。

I.6.1 動機づけ

ステートフル HBS スキームは、いくつかの理由から注目を集めています。

ステートフル HBS スキームの主な目的は、量子セキュリティの強化です。前述したように、ショアのアルゴリズムにより、量子コンピューターは大きな整数を効率的に因数分解し、離散対数を計算することができます。これによって、RSA や ECC などの公開鍵暗号スキームを完全に破ることができます。

対照的に、ステートフル HBS スキームは、その基礎となるハッシュ関数とマークルツリー (通常 SHA256 で実装) のセキュリティに基づいており、暗号に関連する量子コンピューターの登場によって破られることは予想されていません。これらの理由から、ステートフル HBS スキームは NIST SP 800-208 および NSA の CNSA 2.0 スイートで推奨されています。詳細については、次の 2 つのリンクをご参照ください。

- <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

さらに CNSA 2.0 のタイムラインでは、2030 年までにポスト量子ステートフル HBS スキームのみを使用する必要があり、採用は「直ちに」開始する必要があると規定されています。LMS の採用は、CNSA 2.0 スイートのタイムラインで最も早い要件です。

ただし、ステートフル HBS スキームの性質上、その使用と状態の追跡には細心の注意を払う必要があります。ステートフル HBS システムでは、秘密鍵は実際にはワンタイム署名 (OTS) キーの有限セットであり、再利用されることはありません。同じ OTS キーを使用して 2 つの異なるメッセージを署名すると、攻撃者が署名を偽造できる可能性があり、スキーム全体のセキュリティが崩壊します。したがって、ステートフル HBS スキームは、パブリックインターネットなどの一般的な使用には適していません。

LMS/HSS などのステートフル HBS スキームは、特に長い運用寿命が期待され、暗号に関連する量子コンピューターに対して耐性が求められる組み込みシステムや制約付きシステムでのオフラインファームウェア認証と署名検証に特に役立ちます。

I.6.2 LMS/HSS 署名

wolfSSL は、wolfCrypt 組み込み暗号エンジンに LMS/HSS ハッシュベースの署名スキームのサポートを追加しています。これは、以前の liboqs 統合と同様に、hash-sigsLMS/HSS ライブラリ (<https://github.com/cisco/hash-sigs>) との実験的な統合によって実現されます。

Leighton-Micali Signatures(LMS) とそのマルチツリーのバリエーションである Hierarchical Signature System(HSS) は、ポスト量子、ステートフルハッシュベース署名スキームです。公開鍵と秘密鍵が小さく、署名と検証が速いことで知られています。署名のサイズは大きくなりますが、Winternitz パラメーターを介して調整できます。詳細については、RFC8554 の次の 2 つのリンクを参照してください：

- LMS: <https://datatracker.ietf.org/doc/html/rfc8554>
- HSS: <https://datatracker.ietf.org/doc/html/rfc8554#section-6>

前述したように、LMS/HSS 署名システムは有限数のワンタイム署名 (OTS) 鍵で構成されているため、有限数の署名しか安全に生成できません。ただし、署名の数と署名のサイズは、次に説明する一連の定義済みパラメーターを介して調整できます。

I.6.2.1 サポートしているパラメータ

LMS/HSS 署名は 3 つのパラメータによって定義されます。

- levels: マークルツリーのレベル数
- height: 個々のマークルツリーの高さ
- Winternitz: ウィンターニッツチェーンで使用されるハッシュのビット数。署名サイズの時空間トレードオフとして使用されます。

wolfSSL は、[RFC 8554](#)で定義されているすべての LMS/HSS パラメータをサポートしています。

- levels = {1..8}
- height = {5, 10, 15, 20, 25}
- Winternitz = {1, 2, 4, 8}

利用可能な署名の数は次の通りです。 - $N = 2^{levels * height}$

便宜上、一部のパラメータセットは enum wc_LmsParm で事前定義されています。その値を以下の表に示します。

パラメータセット	説明
WC_LMS_PARM_NONE	設定されていません。デフォルトを使用します (WC_LMS_PARM_L1_H15_W2)
WC_LMS_PARM_L1_H15_W2	level:1, height:15, Winternitz:2
WC_LMS_PARM_L1_H15_W4	level:1, height:15, Winternitz:4
WC_LMS_PARM_L2_H10_W2	level:2, height:10, Winternitz:2
WC_LMS_PARM_L2_H10_W4	level:2, height:10, Winternitz:4
WC_LMS_PARM_L2_H10_W8	level:2, height:10, Winternitz:8
WC_LMS_PARM_L3_H5_W2	level:3, height:5, Winternitz:2
WC_LMS_PARM_L3_H5_W4	level:3, height:5, Winternitz:4
WC_LMS_PARM_L3_H5_W8	level:3, height:5, Winternitz:8
WC_LMS_PARM_L3_H10_W4	level:3, height:10, Winternitz:4
WC_LMS_PARM_L4_H5_W8	level:4, height:5, Winternitz:8

ここで設定したパラメータに対する、署名のサイズと署名の数を示します。

パラメータセット	署名サイズ	署名数
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

表から分かるように、署名のサイズは主にレベルと Winternitz 値、および比較的影響は小さいですが高さによって決まります。

- レベル値を大きくすると、署名サイズは大幅に増加します。
- 高さの値を大きくすると、署名のサイズは増加します。
- Winternitz 値を大きくすると、署名のサイズは小さくなりますが、鍵の生成と署名/検証にかかる時間が長くなります。

鍵の生成時間は、第 1 レベルのツリーの高さによって大きく決まります。使用可能な署名の数が同じであっても、レベル 3、高さ 5 のツリーは、初期鍵生成時にレベル 1、高さ 15 のツリーよりもはるかに高速です。

I.6.2.2 LMS/HSS ビルド方法 wolfSSL リポジトリの[INSTALL ファイル](#)をご参照ください。項目 17 には、wolfSSL と hash-sigs LMS/HSS ライブラリを設定および構築する方法についての手順が記載されています。

I.6.2.3 ベンチマークデータ 次のベンチマークデータは、8 コア Intel i7-8700 CPU@3.20GHz、Fedora 38(6.2.9-300.fc38.x86_64) で取得されました。マルチスレッドの例では 4 スレッドと 4 コアが使用されましたが、シングルスレッドの例では 1 コアのみが使用されました。

INSTALL ファイルの項目 17 で説明されているように、hash-sigs ライブラリは 2 つの静的ライブラリを提供します。

- hss_lib.a: シングルスレッド
- hss_lib_thread.a: マルチスレッド

マルチスレッドバージョンではワーカースレッドが生成され、鍵生成などの CPU を集中的に使用するタスクが高速化されます。これにより、主にすべてのパラメータ値に対する鍵の生成と署名が高速化され、より大きなレベル値の検証も多少高速化されます。

なお、以下のベンチマークは次の構成オプションを有効化して取得しました。

```
./configure \
  --enable-static \
  --disable-shared \
  --enable-lms=yes \
  --with-liblms=<path to hash sigs install>
```

マルチスレッドベンチマーク

以下は、集中的なタスクを並列化するために 4 スレッドを使用し、4 コアを使用したマルチスレッド hss_lib_thread.a と共にビルドして取得したベンチマークデータです。

```
./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

```

シングルスレッドベンチマーク

以下は、シングルスレッドの `hss_lib.a` と共にビルドして取得したベンチマークデータです。これは単一のコアのみを使用します。

```
$ ./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.6.3 XMSS/XMSS^{MT} 署名

wolfSSL は、XMSS/XMSS^{MT} ステートフルハッシュベース署名のサポートを追加しています。LMS と同様に、これは [RFC 8391](https://github.com/XMSS/xmss-reference.git) の xmss-reference リポジトリ (<https://github.com/XMSS/xmss-reference.git>) との実験的な統合によって実現されています。

xmss-reference は、xmss_core_fast および xmss_core 実装をサポートしています。xmss_core_fast 実装は、トレードオフとしてより大きな秘密鍵サイズでパフォーマンスを優先するように設計されています。

当社の統合では xmss_core_fast を使用していますが、パッチが適用されているため、代わりに wolfCrypt SHA256 実装を使用できます。

パッチは、[wolfssl-examples リポジトリ](#)の

`pq/stateful_hash_sig/0001-Patch-to-support-wolfSSL-xmss-reference-integration.patch`

にて公開しています。

全体的に、XMSS/XMSS^{MT} は LMS/HSS に似ています。より詳細な比較については、「[LMS vs XMSS: 2 つのハッシュベース署名標準の比較](#)」をご参照ください。

XMSS^{MT} は XMSS のマルチツリー一般化であり、HSS with LMS に似ていますが、XMSS/XMSS^{MT} では Winternitz 値が $w=16$ に固定されている点が異なります。公開鍵は XMSS/XMSS^{MT} では若干大きくなり (XMSS/XMSS^{MT} では 68 バイト、LMS/HSS では 60 バイト)、署名は若干小さくなります。

I.6.3.1 サポートしているパラメータ wolfSSL は、[NIST SP 800-208](#) の表 10 および 11 の SHA256 XMSS/XMSS^{MT} パラメータセットをサポートしています。

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

上記の表で、 n は HASH 関数のバイト数、 w は Winternitz 値、 h はツリーシステムの合計高さ、 d はツリーのレベルを示します。

鍵生成時間は第 1 レベルのツリーの高さ (または h/d) によって大きく左右されますが、署名の長さは主に d (ハイパーツリーレベルの数) とともに増加します。

LMS/HSS と同様に、使用可能な署名の数は 2^{**h} に比例して増加します。ここで h はツリーシステムの合計高さです。

I.6.3.2 ベンチマークデータ 以下では、Intel x86_64 および aarch64 のいくつかの XMSS/XMSS^{MT} パラメータセットのベンチマークデータを示します。これらのシステムでの SHA256 パフォーマンスも参考として記載されています。必要なハッシュチェーンの数が多いため、XMSS/XMSS^{MT} の CPU 作業の大部分が計算されるためです。さらに、xmss-reference へのパッチは wolfCrypt の SHA256 実装を置き換えるため、同じ ASM の高速化のメリットが得られます。

前述のように、XMSS 統合では xmss-reference の `xmss_core_fast` 実装を使用しています。この実装は、秘密鍵のサイズが大きいというトレードオフで、より高速なパフォーマンスを実現します。

x86_64

以下の x86_64 ベンチマークデータは、Fedora 38 (6.2.9-300.fc38.x86_64) が動作する Intel i7-8700 CPU @ 3.20GHz(8 コア) で取得しました。この CPU には `avx avx2` フラグがあり、`--enable-intelasm` を有効化することでハッシュ操作を高速化できます。

`--enable-intelasm` を使用する場合

```
$. /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
```

```
--enable-intelasmを使用しない場合
```

```
$. /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

以下の aarch64 データは、CPU フラグ sha1 sha2 sha3 sha512 を使用して Apple M1 上で実行されている Ubuntu(5.15.0-71-generic) で取得されました。--enable-armasm を使用してビルドすると、特に SHA ハッシュ操作が大幅に高速化されます。

--enable-armasm を使用する場合

```
$ ./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

```

--enable-armasm を使用しない場合

```
$ ./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.6.4 開発者ノート

- 「今収穫し、後で復号する」脅威モデルを阻止しようとしていて、相互運用性の一部を犠牲にしても構わない場合は、Supported Groups 拡張機能で従来のアルゴリズムのサポートを広告したくないことと思います。選択したアルゴリズムで `wolfSSL_UseKeyShare()` と `wolfSSL_set_groups()` を必ず呼び出してください。`wolfSSL_UseKeyShare()` だけ呼び出すだけでは不十分です。なぜなら、それは量子的に脆弱なアルゴリズムのサポートを広告することになるからです。ピアがポスト量子アルゴリズムをサポートしていない場合、HelloRetryRequest を送信し、その結果として、従来のアルゴリズムとの接続が確立されます。

J wolfSSL 移植ガイド

J.1 目的

このガイドは、軽量 SSL/TLS ライブラリ wolfSSL を新しい組み込みプラットフォーム、オペレーティングシステム、トランスポートメディア (TCP/IP、Bluetooth など) に移植するエンジニア向けのリファレンスを提供します。wolfSSL を移植する際に通常変更が必要となる wolfSSL コードベースの領域を説明しています。本稿はあくまで「ガイド」で、技術の進化に伴い変更が必要になることもあります。不足していると感じる箇所があれば、気兼ねなくお知らせください。

J.2 対象読者

このガイドは、wolfSSL および wolfCrypt を、現在サポートされていない新しいプラットフォームまたは環境に移植するエンジニアを対象としています。

J.3 序章

組み込みプラットフォームで wolfSSL を実行するには、いくつかの手順を繰り返す必要があります。これらの手順の一部は、[2.4 章](#) で概説しています。

wolfSSL ドキュメント第 2 章に示した手順に加えて、特定のプラットフォームに対応するために移植または変更が必要なコード領域があります。wolfSSL はこれらの領域の多くを抽象化して、新しいプラットフォームへできるだけ簡単に移植できるようにしています。

`./wolfssl/wolfcrypt/settings.h` ファイルには、さまざまなオペレーティングシステム、TCP/IP スタック、およびチップセット (例: MBED、FREESCALE_MQX、MICROCHIP_PIC32、MICRIUM、EBSNET など) に固有の定義がいくつかあります。wolfSSL をコンパイルして新しいプラットフォームに移植するときに、`#defines` を配置する主な場所は 2 つあります。

1. オペレーティングシステムまたは TCP/IP スタックに対応するための新たなマクロ定義は、通常、wolfSSL のポーティングが完了すると、`settings.h` ファイルに追加しています。これにより、機能のオン/オフを簡単に切り替えたり、そのビルドの「デフォルト」となるビルド設定をカスタマイズしたりできます。wolfSSL を新しいプラットフォームに移植する際にも、このファイルに新しいカスタム定義を追加することでお使いいただけます。新しいプラットフォームへの移植が完了した際、もし差し支えなければ、wolfSSL の [Git リポジトリ](#) に Pull Request を送信していただけると嬉しく思います。これにより、より多くの環境で wolfSSL を使用しやすくなります。
2. wolfSSL 自体に変更を加えたくない場合、または追加のプリプロセッサ定義を使用して wolfSSL ビルドをカスタマイズしたい場合、wolfSSL はカスタムヘッダーファイル `user_settings.h` の使用を推奨します。wolfSSL ソースファイルをコンパイルするときに `WOLFSSL_USER_SETTINGS` が定義されている場合、wolfSSL はカスタムヘッダーファイル `user_settings.h` を自動的にインクルードします。このヘッダーはユーザーが作成し、インクルードパスに配置する必要があります。これにより、ユーザーは wolfSSL ビルド用に 1 つのファイルのみを管理すればよく、wolfSSL の新しいバージョンへの更新がはるかに簡単になります。

wolfSSL では、直接のメール (info@wolfssl.jp) または [GitHub Pull Request](#) を通じてパッチとコード変更をご送信いただくことを推奨しています。

J.4 wolfSSL の移植

J.4.1 データ型

Q: どんな場合にこの章の内容が役立ちますか?

A: プラットフォームに適したデータ型のサイズを設定することは常に重要です。

wolfSSL は、64 ビット型を利用できることで速度面でメリットを得ています。プラットフォームの `sizeof(long)` と `sizeof(long long)` の結果と一致するように `SIZEOF_LONG` と `SIZEOF_LONG_LONG` を定義します。これは、`settings.h` または `user_settings.h` のカスタム定義に追加できます。たとえば、`MY_NEW_PLATFORM` のサンプル定義の下の `settings.h` では次のように示します。

```
#ifdef MY_NEW_PLATFORM
    #define SIZEOF_LONG 4
    #define SIZEOF_LONG_LONG 8
    ...
#endif
```

wolfSSL と wolfCrypt では、`word32` と `word16` という 2 つの追加データ型を使用します。これらのデフォルトの型マッピングは次のとおりです。

```
#ifndef WOLFSSL_TYPES
#ifndef byte
    typedef unsigned char byte;
#endif
    typedef unsigned short word16;
    typedef unsigned int word32;
    typedef byte word24[3];
#endif
```

`word32` はコンパイラの 32 ビット型にマッピングされ、`word16` はコンパイラの 16 ビット型にマッピングされます。これらのデフォルトのマッピングがプラットフォームに適していない場合は、`settings.h` または `user_settings.h` で `WOLFSSL_TYPES` を定義し、`word32` および `word16` に独自のカスタム typedef を割り当てる必要があります。

wolfSSL の `fastmath` ライブラリは、`fp_digit` および `fp_word` 型を使用します。デフォルトでは、これらはビルド構成に応じて `<wolfssl/wolfcrypt/tfm.h>` にマッピングされます。

`fp_word` は `fp_digit` の 2 倍のサイズである必要があります。デフォルトのケースがプラットフォームに当てはまらない場合は、`settings.h` または `user_settings.h` で `WOLFSSL_BIGINT_TYPES` を定義し、`fp_word` および `fp_digit` に独自のカスタム typedef を割り当てる必要があります。

wolfSSL は、一部の操作において使用可能な場合は 64 ビット型を使用します。ビルド時には、`SIZEOF_LONG` および `SIZEOF_LONG_LONG` の設定に基づいて、`word64` の正しいデータ型を検出して設定しようとしません。真の 64 ビット型を持たない一部のプラットフォームでは、2 つの 32 ビット型を合わせて使用するため、パフォーマンスが低下する可能性があります。コンパイル時に `NO_64BIT` を定義することで、64 ビット型を使用しないこともできます。

J.4.2 エンディアン

Q: どんな場合にこの章の内容が役立ちますか?

A: あなたのプラットフォームがビッグエンディアンの場合です。

お使いのプラットフォームはビッグエンディアンとリトルエンディアン、どちらでしょうか。wolfSSL はデフォルトでリトルエンディアンを使用しています。システムがビッグエンディアンの場合は、wolfSSL をビルドする際に `BIG_ENDIAN_ORDER` を定義します。例えば、`settings.h` で次のように設定できます。

```
#ifdef MY_NEW_PLATFORM
    ...
    #define BIG_ENDIAN_ORDER
    ...
#endif
```

J.4.3 writev

Q: どんな場合にこの章の内容が役立ちますか？

A: <sys/uio.h> を利用できない場合です。

デフォルトでは、wolfSSL API は、writev() セマンティクスをシミュレートする wolfSSL_writev() をアプリケーションに提供します。<sys/uio.h> ヘッダーが利用できないシステムでは、NO_WRITEV を定義してこの機能を除外します。

J.4.4 ネットワーク I/O

Q: どんな場合にこの章の内容が役立ちますか？

A: BSD スタイルのソケット API を利用できない、あるいはカスタムトランスポート層または TCP/IP スタックを使用している、または静的バッファのみを使用したい場合です。

wolfSSL はデフォルトで BSD スタイルのソケットインターフェイスを使用します。トランスポート層が BSD ソケットインターフェイスを提供する場合、カスタムヘッダーが必要でない限り、wolfSSL はそのまま統合する必要があります。

wolfSSL はカスタム I/O 抽象化レイヤーを提供し、ユーザーは wolfSSL の I/O 機能をシステムに合わせて調整できます。詳細については、[5.1.2 節](#) をご参照ください。

具体的には WOLFSSL_USER_IO を定義し、wolfSSL デフォルトの EmbedSend() と EmbedReceive() をテンプレートとして使用し、独自の I/O コールバック関数を記述します。これら 2 つの関数は ./src/io.c にあります。

wolfSSL は、入力と出力に動的バッファを使用します。このバッファのデフォルトは 0 バイトです。バッファよりも大きいサイズの入力レコードを受信した場合、動的バッファが一時的にリクエストの処理に使用され、その後解放されます。

動的メモリを必要としない 16kB の大きな静的バッファを使用する場合は、LARGE_STATIC_BUFFERS を定義することでこのオプションを使用できます。

動的バッファが使用され、ユーザーがバッファサイズよりも大きい wolfSSL_write() を要求した場合、最大 MAX_RECORD_SIZE までの動的ブロックを使用してデータを送信します。RECORD_SIZE で定義されている現在のバッファサイズの最大のチャンクでのみデータを送信したいユーザーは、STATIC_CHUNKS_ONLY を定義することでこれを実現できます。この定義を使用する場合、RECORD_SIZE はデフォルトで 128 バイトになります。

J.4.5 ファイルシステム

Q: どんな場合にこの章の内容が役立ちますか？

A: ファイルシステムを利用できない、あるいは標準のファイルシステム機能を利用できない、または独自のファイルシステムを使用している場合です。

wolfSSL は、TLS セッションまたはコンテキストに鍵と証明書をロードするためにファイルシステムを使用します。wolfSSL では、これらをメモリバッファからロードすることもできます。メモリバッファのみを使用する場合、ファイルシステムは必要ありません。

ライブラリをビルドするときに NO_FILESYSTEM を定義することで、wolfSSL によるファイルシステムの使用を無効にできます。つまり、証明書と鍵はファイルではなくメモリバッファからロードする必要があります。settings.h で、以下のように設定できます。

```
#ifdef MY_NEW_PLATFORM
...
#define NO_FILESYSTEM
```

```
...
#endif
```

テスト用の鍵と証明書バッファは、./wolfssl/certs_test.h ヘッダーファイルにあります。これらは、./certs ディレクトリにある対応する証明書や鍵と一致します。

certs_test.h ヘッダーファイルは、必要に応じて ./gencertbuf.pl スクリプトを使用して更新できます。gencertbuf.pl 内には、fileList_1024 と fileList_2048 の 2 つの配列があります。鍵サイズに応じて、追加の証明書またはキーをそれぞれの配列に追加できます。なお、DER 形式である必要があります。上記の配列は、証明書・鍵ファイルの場所を目的のバッファ名にマップします。gencertbuf.pl を変更した後、wolfSSL ルートディレクトリから実行すると、./wolfssl/certs_test.h の証明書・鍵のバッファが更新されます。

```
./gencertbuf.pl
```

デフォルト以外のファイルシステムを使用する場合、ファイルシステム抽象化レイヤーは ./wolfssl/-wolfcrypt/wc_port.h にあります。ここには、EBSNET、FREESCALE_MQX、MICRIUM などのさまざまなプラットフォームのファイルシステムポートがあります。必要に応じて、プラットフォームのカスタム定義を追加できます。これにより、XFILE、XFOPEN、XFSEEK などを使用してファイルシステム関数を定義できます。たとえば、Micrium の μ C/OS (MICRIUM) の wc_port.h のファイルシステムレイヤーは次のとおりです。

```
#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 スレッド化

Q: どんな場合にこの章の内容が役立ちますか？

A: マルチスレッド環境で wolfSSL を使用したい、またはシングルスレッドモードでコンパイルしたい場合です。

wolfSSL をシングルスレッド環境でのみ使用する場合、wolfSSL をコンパイルするときに SINGLE_THREADED を定義することで wolfSSL ミューテックスレイヤーを無効にできます。これにより、wolfSSL ミューテックスレイヤーを移植する必要がなくなります。

wolfSSL をマルチスレッド環境で使用する場合、wolfSSL ミューテックスレイヤーを新しい環境に移植する必要があります。ミューテックスレイヤーは ./wolfssl/wolfcrypt/wc_port.h および ./wolfcrypt/src/wc_port.c にあります。新しいシステムでは、wc_port.h に wolfSSL_Mutex を定義し、wc_port.c にミューテックス関数 (wc_InitMutex、wc_FreeMutex、wc_LockMutex、wc_UnLockMutex) を定義する必要があります。wc_port.h と wc_port.c を検索すると、既存のプラットフォームポートレイヤー (EBSNET、FREESCALE_MQX など) の例を参照できます。

J.4.7 ランダムシード

Q: どんな場合にこの章の内容が役立ちますか？

A: /dev/random や /dev/urandom を利用できない、あるいはハードウェア RNG に統合する必要がある場合です。

デフォルトでは、wolfSSL は `/dev/urandom` または `/dev/random` を使用して RNG シードを生成します。wolfSSL をビルドするときに `NO_DEV_RANDOM` 定義を使用して、デフォルトの `GenerateSeed()` 関数を無効にすることができます。これが定義されている場合は、ターゲットプラットフォームに固有の `./wolfcrypt/src/random.c` にカスタム `GenerateSeed()` 関数を記述する必要があります。これにより、ハードウェアベースのランダムエントロピーソースが利用可能な場合は、wolfSSL の PRNG にシードを設定できます。

`GenerateSeed()` の記述方法の例については、`./wolfcrypt/src/random.c` にある wolfSSL の既存の `GenerateSeed()` 実装を参照してください。

J.4.8 メモリー

Q: どんな場合にこの章の内容が役立ちますか？

A: 標準のメモリ関数が利用できない、あるいはオプションの数学ライブラリ間のメモリ使用量の違いに関心がある場合です。

wolfSSL 本体はデフォルトで `malloc()` と `free()` の両方を使用します。旧来使用されてきた第 1 世代の整数演算ライブラリ (Normal Math) を使用する場合、wolfCrypt は `realloc()` も使用します。

現在、整数演算ライブラリとして最初期に開発された第 1 世代の Normal Math ライブラリ、パブリックドメインの TFM(Tom's Fast Math) をベースに開発した第 2 世代の Fast Math ライブラリ、SP(Single Precision) 最適化を適用した第 3 世代の SP Math ライブラリが存在します。このうち、第 2 世代以降の Fast Math, SP Math ライブラリでは、暗号操作において動的メモリを使用しません。

メモリ使用量とスピードの観点から、私たちは SP Math ライブラリの使用を推奨しています。wolfSSL 5.4.0 以降ではデフォルトで採用しており、第 1 世代の Normal Math ライブラリは廃止予定です。詳細は以下のページをご覧ください。

- [wolfSSL の新たな Multi-Precision 演算ライブラリ](#)
- [wolfSSL Math Library Comparison Matrix](#)
- [レガシー数学ライブラリを廃止します](#)

wolfSSL の TLS レイヤーは依然としていくらかの動的メモリを使用するため、`malloc()` と `free()` は依然として必要です。

通常の `malloc()`、`free()`、および場合によっては `realloc()` 関数が使用できない場合は、`XMALLOC_USER` を定義し、ターゲット環境に固有の `./wolfssl/wolfcrypt/types.h` でカスタムメモリ関数フックを提供します。

`XMALLOC_USER` の使用の詳細については、[5.1.1.1 節](#) をご参照ください。

J.4.9 時間

Q: どんな場合にこの章の内容が役立ちますか？

A: 標準の時間関数 (`time()`、`gmtime()`) を利用できない、あるいはカスタムのクロックティック関数を指定する必要がある場合です。

デフォルトでは、wolfSSL は `./wolfcrypt/src/asn.c` で指定されているように、`time()`、`gmtime()`、および `ValidateDate()` を使用します。これらは、`XTIME`、`XGMTIME`、および `XVALIDATE_DATE` に抽象化されています。標準の時間関数と `time.h` を利用できない場合は、ユーザーは `USER_TIME` を定義できます。USER_TIME を定義した後、ユーザーは独自の `XTIME`、`XGMTIME`、および `XVALIDATE_DATE` 関数を定義できます。

wolfSSL は、クロックティック関数にデフォルトで `time(0)` を使用します。これは、`LowResTimer()` 関数内の `./src/internal.c` にあります。

USER_TICKS を定義すると、time(0) が必要ない場合にユーザーが独自のクロックティック関数を定義できるようになります。カスタム関数には秒単位の精度が必要ですが、エポックと相関させる必要はありません。参考として、./src/internal.c の LowResTimer() 関数を参照してください。

J.4.10 C 標準ライブラリ

Q: どんな場合にこの章の内容が役立ちますか？

A: C 標準ライブラリを使用できない、あるいは独自のライブラリがある場合です。

wolfSSL は、開発者に高いレベルの移植性と柔軟性を提供するために、C 標準ライブラリなしで構築できるようにしています。その場合、ユーザーは C 標準の関数の代わりに使用したい関数をマップする必要があります。

第 7 章では、メモリ関数について説明しました。メモリ関数の抽象化に加えて、wolfSSL は文字列関数と数学関数も抽象化します。特定の関数は通常、X<FUNC> 形式の定義に抽象化されます。<FUNC> は抽象化される関数の名前です。

詳細については、5.1 章をご覧ください。

J.4.11 ロギング

Q: どんな場合にこの章の内容が役立ちますか？

A: デバッグメッセージを有効にしたいが、stderr を使用できない場合です。

デフォルトでは、wolfSSL は stderr を介してデバッグ出力を提供します。デバッグメッセージを有効にするには、wolfSSL を DEBUG_WOLFSSL を定義してコンパイルし、アプリケーションコードから wolfSSL_Debugging_ON() を呼び出す必要があります。同様に、アプリケーションから wolfSSL_Debugging_OFF() を呼び出すことで、wolfSSL デバッグメッセージをオフにすることもできます。

stderr を使用できない、あるいは別の出力ストリームや別の形式でデバッグメッセージを出力したい環境には、独自のコールバック関数を使用できるようにしています。

詳細については、8.1 章をご覧ください。

J.4.12 公開鍵操作

Q: どんな場合にこの章の内容が役立ちますか？

A: wolfSSL で独自の公開鍵実装を使用したい場合です。

wolfSSL では、SSL/TLS レイヤーが公開鍵操作を行う必要があるときに呼び出される、独自の公開鍵コールバックをユーザーが作成できます。ユーザーはオプションで 6 つの機能を定義できます。

1. ECC 署名 コールバック
2. ECC 検証 コールバック
3. RSA 署名 コールバック
4. RSA 検証 コールバック
5. RSA 暗号化コールバック
6. RSA 復号 コールバック

詳細については、6.4 章をご覧ください。

J.4.13 アトミックレコードレイヤーの処理

Q: どんな場合にこの章の内容が役立ちますか？

A: レコードレイヤーの処理、特に MAC/暗号化および復号/検証操作を独自に実行したい場合です。

デフォルトでは、wolfSSL は暗号ライブラリ wolfCrypt を使用し、ユーザーに代わってレコードレイヤーの処理を行います。wolfSSL は、SSL/TLS 接続中に MAC/暗号化および復号/検証機能をより細かく制御したいユーザーのために、アトミックレコード処理コールバックを提供します。

ユーザーは 2 つの関数を定義できます。

1. MAC/暗号化コールバック関数
2. 復号/検証コールバック関数

詳細については、6.3 章をご覧ください。

J.4.14 機能

Q: どんな場合にこの章の内容が役立ちますか?

A: 特定の機能を無効にしたい場合です。

wolfSSL をビルドする際、マクロ定義を使用して特定の機能を無効化できます。使用可能なマクロ定義については、2 章をご覧ください。

J.5 次のステップ

J.5.1 wolfCrypt テストアプリケーション

wolfSSL をターゲットプラットフォームで適切にビルドできるようになったら、次のステップとして wolfCrypt テストアプリケーションを移植するのがよいでしょう。このアプリケーションをターゲットシステムで実行すると、NIST テストベクトルを使用して、すべての暗号アルゴリズムが正しく動作しているかどうか検証されます。

このステップを省略して SSL/TLS 接続に進むと、基盤となる暗号操作の失敗によって発生する問題のデバッグが難しくなる可能性があります。

wolfCrypt テストアプリケーションは `./wolfcrypt/test/test.c` にあります。組み込みアプリケーションに独自の `main()` 関数がある場合は、`./wolfcrypt/test/test.c` をコンパイルするときに `NO_MAIN_DRIVER` を定義する必要があります。これにより、アプリケーションの `main()` が各暗号/アルゴリズム テストを個別に呼び出すことができます。

組み込みデバイスに wolfCrypt テストアプリケーション全体を実行するのに十分なリソースがない場合は、個々のテストを `test.c` から切り離して個別にコンパイルできます。`test.c` から分離された暗号テストを抽出するときは、特定のテストケースに必要な正しいヘッダーファイルがビルドに含まれていることを確認してください。

J.6 サポート

一般的なサポートの質問は、メール、サポートフォーラム (英語)、または wolfSSL の Zendesk チケットトラッキングシステムを介して wolfSSL に直接送信できます。

Web サイト: <https://www.wolfssl.com>

サポート用メールアドレス: support@wolfssl.com

Zendesk: <https://wolfssl.zendesk.com>

フォーラム: <https://www.wolfssl.com/forums>

wolfSSL は、ユーザーと顧客が wolfSSL を新しい環境に移植するのを支援するために、いくつかのサポートパッケージとコンサルティングサービスを提供しています。

サポートパッケージ: <https://wolfssl.jp/license/support-packages/>

お問い合わせ: info@wolfssl.jp