wolfHSM Documentation

2026-07-04

Contents

1 wolfHSM Documentation

1.1 Table of Contents

1.2 Overview

wolfHSM is a software framework that provides a portable and open-source client-server abstraction for hardware cryptography, non-volatile memory, and isolated secure processing that maximizes security and performance. It consists of a client-server library architecture, where the wolfHSM server application runs in a trusted environment, and client applications communicate with the server through the wolfHSM client library. wolfHSM is intended to simplify the challenge of moving between hardware with enhanced security features without being tied to any vendor-specific library calls. wolfHSM also dramatically simplifies client HSM applications by allowing direct use of wolfCrypt APIs, with the framework automatically offloading all sensitive cryptographic operations to the wolfHSM server as remote procedure calls with no additional logic required by the client app.

Although initially targeted to automotive-style HSM-enabled microcontrollers, wolfHSM can run on any platform that provides a secure/trusted execution environment for the server. wolfHSM provides an extensible solution to support future capabilities of any platform while still supporting standardized interfaces and protocols such as AUTOSAR SHE.

1.3 Features

- Unified client and server library APIs abstracting common HSM operations
- Transport-agnostic architecture with reference implementations for shared memory buffers, POSIX TCP sockets, wolfSSL TLS on top of TCP, and POSIX inter-process shared memory
- Non-volatile object store with fail-safe atomic updates and fine-grained access control policies
- Per-client and global key stores supporting non-volatile and RAM-backed key slots, configurable usage policies, and encrypted (wrapped) keys
- Tight integration with the wolfCrypt cryptography library
  - Client applications may use the wolfCrypt API directly, with supported algorithms and key storage transparently offloaded to the HSM server
  - Server applications gain access to all wolfCrypt software algorithms, with optional device-specific hardware acceleration where available
  - Support for Post-Quantum Cryptography (PQC) algorithms
- DMA support for shared-memory systems, including address allowlisting and configurable pre- and post-access callbacks for cache synchronization and address remapping
- X.509 certificate chain verification, including support for RFC 5755 attribute and authorization certificates
- Server image manager for authenticating firmware images or arbitrary regions of memory with customizable post-verification actions
- Extensible server behavior via user-defined callbacks for runtime functionality and message handling
- Non-volatile monotonic counters
- AUTOSAR Secure Hardware Extension (SHE) interface support
- Support for user authentication with full role-based access control

1.4 Supported Platforms

1.4.1 Infineon

- TC2xx
- TC3xx
- TC4xx

1.4.2 ST Microelectronics

- Stellar SR6G
- SPC58N Bernina
- STM32H5 Trust Zone

1.4.3 Texas Instruments

- TDA4VH

1.4.4 Renesas

- Renesas RH850

1.4.5 Microchip

- PIC32CZ

1.4.6 AMD/Xilinx

- Zynq UltraScale+

1.4.7 Planned Ports

- Microchip PIC32CN
- NXP S32K3 and S32G
- AMD/Xilinx Versal

2 FAQ

2.1 What is wolfHSM?

wolfHSM is a portable, open-source client-server framework for cryptography, non-volatile memory (NVM), and isolated secure processing. A server application runs in a trusted environment (physical HSM core, trust zone secure world, remote server, etc.) while client applications use a library API that can offload cryptographic and storage operations to the server. The core library exposes a client and server API to help developers stitch together their own HSM applications using a curated set of ready-to-use and secure components.

2.2 Why would I need wolfHSM?

wolfHSM was initially designed for developers targeting automotive-style SoCs that include a dedicated HSM core with secure resources such as protected memory regions and hardware cryptographic accelerators, however can generalize to any application scenario where operations must be delegated to a secure programmable secure environment. It enables applications running in an untrusted environment to securely leverage the HSM without requiring a custom, ground-up implementation, while still allowing flexibility to tailor HSM functionality to specific product requirements.

When building security-sensitive products on these platforms, developers typically face two common approaches:

The first option is to build everything from scratch. This is a daunting undertaking that involves developing a custom application to run on the HSM core, managing non-volatile memory, implementing cryptographic algorithms in software or writing hardware accelerator drivers, and creating a bespoke messaging or RPC framework to communicate between cores. In many cases, silicon vendors provide minimal reference code, limited documentation, and little direct support for HSM software development.

The second option is to rely on vendor-provided, fixed-function HSM binaries. These solutions are typically expensive and expose only a narrow, predefined API. While convenient for simple use cases, they are often inflexible and difficult to adapt. Algorithm support is usually fixed or constrained to available hardware accelerators, large chunks of system memory are reserved for opaque internal use, and feature sets are limited by the vendor's design choices.

wolfHSM provides a practical middle ground. It offers a flexible, open-source framework that implements commonly required HSM functionality while empowering users to build their own HSM applications. The core library is hardware agnostic, extensible, auditable, and modular, exposing essential building blocks that can be composed to meet specific security and system requirements. wolfHSM is built on top of wolfCrypt, leveraging a mature, well-tested cryptographic foundation while adding key management, policy enforcement, transport abstraction, and secure storage capabilities.

Building a custom HSM application sounds scary if you are used to a "black box" HSM solutions, but fear not! We took care of the hard part! Once you realize the freedom and flexibility that full control over the HSM-side provides, you will never want to go back. wolfHSM empowers your team to build a bespoke HSM solution that is right for YOU!

2.3 What is an HSM client and server?

The HSM server is an application that runs in a trusted or secure execution environment, typically on a dedicated HSM core. It is responsible for performing cryptographic operations, managing keys, and accessing non-volatile memory (NVM) on behalf of clients.

An HSM client is an application that consumes these services from outside of the secure environment. Clients communicate with the server using a linked client library that marshals requests over a de-

fined transport and returns responses. This architecture allows applications to use the wolfHSM and wolfCrypt APIs transparently, without requiring direct interaction with or management of the secure execution environment.

2.4 How is wolfHSM different from other vendor solutions?

wolfHSM is a library framework rather than a proprietary, fixed-function firmware. It is designed to enable users to build their own custom HSM applications instead of being constrained to a predefined feature set. The library is vendor-agnostic and open-source, with a clean abstraction layer and pluggable transports. For users who just want an out-of-the-box solution, each wolfHSM port ships with a "reference server" with default functionality that can be directly loaded onto HSM hardware and used by client applications.

wolfHSM also differs from other solutions due to its inherent portability. Client applications can use wolfHSM and wolfCrypt APIs directly, with sensitive operations transparently offloaded to the HSM, avoiding proprietary interfaces and simplifying portability across HSM-enabled platforms. When migrating between platforms, the same high-level application logic can be retained, requiring only the registration of platform-specific transport and non-volatile storage drivers.

2.5 What is the difference between wolfHSM and wolfSSL/wolfCrypt

wolfSSL/wolfCrypt is a standalone library that provides a TLS stack (wolfSSL) alongside an integrated cryptography library (wolfCrypt). wolfHSM is a separate library that is built on top of wolfSSL/wolfCrypt, using it internally as its cryptographic provider. Think about it like a remote procedure call framework for wolfCrypt with some extra HSM-specific functionality on top. Put more formally: wolfHSM wraps wolfCrypt within a client–server framework, allowing applications to offload cryptographic operations to a secure server while adding key management, non-volatile memory (NVM) management, transport abstraction, and policy enforcement layers on top of wolfSSL/wolfCrypt's standard cryptographic functionality.

2.6 What is the difference between wolfHSM and wolfBoot?

wolfBoot is a secure bootloader focused on authenticated boot and firmware updates. wolfHSM is an HSM framework for runtime cryptography, key storage, and secure processing. However, wolfBoot and wolfHSM are best friends! wolfBoot has deep integration with wolfHSM and can be configured to run both as a wolfHSM client to accelerate and offload secure boot crypto and key storage, or to run on the HSM server core to verify the HSM server application. wolfHSM also knows how to verify wolfBoot images for scenarios where the HSM controls the boot flow on a multicore SoC.

2.7 Is wolfHSM open source?

Yes. wolfHSM is open source under wolfSSL's dual licensing terms. See `LICENSING` for details.

2.8 Does wolfHSM have any external dependencies?

The core wolfHSM library depends only on wolfSSL/wolfCrypt. Platform-specific reference ports may additionally rely on operating system or vendor libraries for startup code, configuration, transport mechanisms, and storage drivers, but no other third-party dependencies are required by the core library.

Due to wolfHSM's modular design, reference platform code can be easily replaced with user-provided transport or storage drivers if the supplied implementations do not meet specific system requirements.

2.9 Does wolfHSM require dynamic memory allocation?

No. wolfHSM is designed to avoid dynamic memory allocation. The library itself does not ever allocate from the heap. wolfCrypt, a dependency of wolfHSM, **does** require allocation for *some* functionality, however this can be configured to use statically allocated fixed-size memory pools, eliminating the need for a runtime heap. See chapter04 of the wolfSSL manual for more details on wolfCrypt static memory. When wolfCrypt is configured to use the static memory feature, wolfHSM applications are guaranteed to never allocate from the heap.

2.10 What is a wolfHSM "port"?

A wolfHSM "port" contains the platform-specific **reference code** that demonstrates how to bring up and run the wolfHSM library on a given device or execution environment. While the core wolfHSM library itself is written in OS-agnostic C and contains no direct hardware or operating system dependencies, the port is what supplies the concrete implementations and drivers required to operate on real hardware.

A typical wolfHSM port includes reference implementations for low-level services such as client–server transport mechanisms, non-volatile storage access, startup and initialization logic, and any required platform or OS configuration. On platforms with hardware cryptographic accelerators or dedicated secure resources, the port may also include example drivers or integration logic to expose those capabilities to the wolfHSM server.

These ports are intentionally modular and illustrative rather than prescriptive. They are provided to show how wolfHSM can be integrated on a specific hardware platform, not to dictate a required architecture. Due to wolfHSM's modular design, all port components can be easily replaced or customized with user-provided implementations while retaining the full functionality of the core wolfHSM library.

Most wolfHSM port implementations are not open source because they interact directly with silicon vendor–specific security hardware and reference device-specific registers, peripherals, and SDK components that are typically covered by the silicon vendor's security-related non-disclosure agreements. This limitation is a practical consequence of working with secure hardware platforms and is not a policy choice by wolfSSL, Inc. For most platforms, wolfSSL only needs to verify that you hold a valid NDA for the device in question in order to share the corresponding port. If you are interested in obtaining a wolfHSM port for a specific platform, please contact facts@wolfssl.com

2.11 I looked at wolfHSM and don't see my specific hardware platform, where is it?

Some platform ports are public (for example, POSIX and STM32H5). The majority of hardware ports are vendor-NDA restricted and are not in this repository. If you do not see your platform, you can either implement a new port using the transport/NVM/flash abstractions or contact wolfSSL for access to restricted ports.

Most wolfHSM port implementations are not open source because they interact directly with silicon vendor–specific security hardware and reference device-specific registers, peripherals, and SDK components that are typically covered by the silicon vendor's security-related non-disclosure agreements. This limitation is a practical consequence of working with secure hardware platforms and is not a policy choice by wolfSSL, Inc. For most platforms, wolfSSL only needs to verify that you hold a valid NDA for the device in question in order to share the corresponding port. If you are interested in obtaining a wolfHSM port for a specific platform, please contact facts@wolfssl.com

2.12 Does wolfHSM support operating system X?

The core wolfHSM library does not rely on operating system primitives and is written in portable C99. It can run in bare-metal environments as well as on operating systems in 32-bit and larger architectures.

Nothing OS-specific should prohibit usage of wolfHSM on a given platform as long as the appropriate transport and storage drivers are supplied.

2.13 Does wolfHSM support compiler X?

wolfHSM (and wolfCrypt) is written in portable C and is designed to build with a wide range of embedded and cross-compilation toolchains. Official support is provided for common, flagship toolchains associated with each device port, and most additional toolchains can be enabled with minimal changes. If you would like to see a specific toolchain officially supported, please contact facts@wolfssl.com

2.14 Does wolfHSM support device/platform X?

In general, yes. Reference ports currently exist for the listed supported platforms. Most platform ports are NDA-restricted by the silicon vendor and must be distributed separately as their own bundle. If a port does not exist for your platform, adding support can typically be accomplished in a matter of weeks, depending on the complexity of the device and desired use case. wolfSSL also routinely adds ports to new devices on request as part of a consulting engagement.

If you are interested in obtaining a restricted port for a platform, or want to see a new device supported, contact facts@wolfssl.com.

2.15 Does wolfHSM support concurrency or multithreading?

The core wolfHSM library does not internally use any threading or parallelism.

The wolfHSM client API is safe to use in a multithreaded environment as long as access to each client context is properly serialized. The client context is not meant to be shared across threads without caller serialization.

The wolfHSM server API is safe to use in a multithreaded environment as long as access to each server context is properly serialized. The server context is not meant to be shared across threads without caller serialization. Global shared resources accessible across server contexts through the server API ARE safe to use in concurrent scenarios as the server library will properly serialize access internally using the internal port-specific mutex abstraction.

For example, two threads must NOT use the server API on one server context shared between the threads. However, it is perfectly acceptable for two threads to poll their own server contexts in parallel, even if the two threads are handling requests that reference the same keys or NVM objects.

2.16 What is the bare minimum my client application needs to do in order to use wolfHSM?

At a minimum, a client application must link against the wolfHSM client library and provide a platform-specific transport that enables communication with the wolfHSM server running in a secure environment. The transport is configured and bound to the client context through a series of configuration structures. Once initialized, the application can call wolfHSM APIs or supported wolfCrypt APIs directly, with cryptographic operations, key access, and secure storage transparently offloaded to the HSM server. The client does not need to manage keys, hardware resources, or the secure execution environment.

See for a concise quickstart example

2.17 What is the bare minimum my server application needs to do in order to use wolfHSM?

At a minimum, a server application must link against the wolfHSM server library, initialize the server context by registering the platform-specific drivers for transport, non-volatile storage, and optionally hardware crypto, then start the request-processing loop, calling `wh_Server_HandleRequestMessage` on a server context to poll for requests from the corresponding client.

Once initialized, the server handles all cryptographic operations, key management, and secure storage on behalf of connected clients. No application-specific logic is required beyond initialization and transport handling, although the server can be extended with custom memory access patterns, callbacks, or services as needed.

2.18 How do I do X with wolfHSM?

First, checkout the quickstart guide in and the descriptions of each high level feature in . You can also look at the example server and client applications in `examples/posix/wh_posix_server` and `examples/posix/wh_posix_client` as well as the individual feature demo examples in `examples/demo/client`.

If you still have questions, reach out to our engineers for direct support at support@wolfssl.com.

3 Quickstart

This quickstart example demonstrates a minimal, end-to-end overview of how to bring up wolfHSM and begin using it from both the client and server sides. It walks through the essential initialization steps required to initialize and run the wolfHSM server and process requests, as well as how to connect a client application to it using a supported transport.

This quickstart uses the built-in shared memory transport for communication, and the NVM flash implementation for NVM object storage. The actual flash drivers bound to the NVM flash layer are not defined, as they would be hardware specific.

The client and server are shown as two separate programs, mirroring a typical deployment where the server runs on a trusted HSM core and the client runs on an application core. Because the shared memory transport exchanges data through two shared buffers, both programs must reference the *same* physical memory: on real hardware these buffers live in a shared SRAM region placed at addresses agreed upon by both cores. Fully runnable POSIX versions (using the TCP and POSIX shared memory transports) are provided in Examples and Demos.

Enough with the high level concepts and jargon, let's dive straight into the code...

3.1 Client Quickstart

```c
#include <stdio.h>  /* for printf() */
#include <string.h> /* for strlen(), memcmp() */

#include "wolfhsm/wh_error.h"          /* WH_ERROR_OK and friends */
#include "wolfhsm/wh_client.h"         /* Client API (includes comm config) */
#include "wolfhsm/wh_transport_mem.h"  /* Shared-memory transport */

#define SHARED_BUFFER_SIZE 4096

/* Request and response buffers shared with the server core. The client writes
 * requests to gReqBuffer and reads responses from gRespBuffer. On real hardware
 * these must be placed (e.g. via the linker) in a memory region shared with the
 * server, so that both cores reference the identical buffers. */
uint8_t gReqBuffer[SHARED_BUFFER_SIZE];
uint8_t gRespBuffer[SHARED_BUFFER_SIZE];

int main(void)
{
    int rc;

    /* 1. Describe the shared-memory transport and its callback table */
    whTransportMemConfig transportCfg[1] = {{
        .req       = gReqBuffer,
        .req_size  = SHARED_BUFFER_SIZE,
        .resp      = gRespBuffer,
        .resp_size = SHARED_BUFFER_SIZE,
    }};
    whTransportClientCb         transportCb[1]  = {WH_TRANSPORT_MEM_CLIENT_CB};
    whTransportMemClientContext transportCtx[1] = {0};

    /* 2. Bind the transport to the client comm configuration */
    whCommClientConfig commCfg[1] = {{
        .transport_cb      = transportCb,
```

```
        .transport_context = (void*)transportCtx,
        .transport_config  = (void*)transportCfg,
        .client_id         = 1, /* unique client identifier (1-15) */
    }};

    /* 3. Assemble the client configuration */
    whClientConfig clientCfg[1] = {{
        .comm = commCfg,
    }};

    /* 4. Initialize the client context */
    whClientContext client[1] = {0};
    rc = wh_Client_Init(client, clientCfg);
    if (rc != WH_ERROR_OK) {
        printf("wh_Client_Init failed: %d\n", rc);
        return 1;
    }

    /* 5. Connect to the server (exchanges client and server IDs) */
    rc = wh_Client_CommInit(client, NULL, NULL);
    if (rc != WH_ERROR_OK) {
        printf("wh_Client_CommInit failed: %d\n", rc);
        return 1;
    }

    /* 6. Use the client API. Here we send a blocking echo request. */
    {
        const char sendBuffer[] = "Hello, wolfHSM!";
        uint16_t   sendLen      = (uint16_t)strlen(sendBuffer);
        char       recvBuffer[WOLFHSM_CFG_COMM_DATA_LEN] = {0};
        uint16_t   recvLen      = 0;

        rc = wh_Client_Echo(client, sendLen, sendBuffer, &recvLen, recvBuffer);
        if (rc == WH_ERROR_OK && recvLen == sendLen &&
            memcmp(sendBuffer, recvBuffer, sendLen) == 0) {
            printf("Server echoed: %.*s\n", recvLen, recvBuffer);
        }
        else {
            /* Error: we weren't echoed back exactly what we sent */
            printf("Echo failed: rc=%d\n", rc);
        }
    }

    /* 7. Close the connection and release the client context */
    (void)wh_Client_CommClose(client);
    (void)wh_Client_Cleanup(client);

    return 0;
}
```

3.2 Server Quickstart

```
#include <stdio.h> /* for printf() */
```

```c
#include "wolfhsm/wh_error.h"           /* WH_ERROR_OK and friends */
#include "wolfhsm/wh_server.h"          /* Server API */
#include "wolfhsm/wh_transport_mem.h"   /* Shared-memory transport */
#include "wolfhsm/wh_nvm.h"             /* NVM abstraction */
#include "wolfhsm/wh_nvm_flash.h"       /* NVM-on-flash implementation */
#include "wolfhsm/wh_flash.h"           /* whFlashCb interface */

#define SHARED_BUFFER_SIZE 4096

/* The same shared buffers referenced by the client (see the client example).
 * The server reads requests from gReqBuffer and writes responses to
 * gRespBuffer. Both programs must reference the identical physical memory. */
uint8_t gReqBuffer[SHARED_BUFFER_SIZE];
uint8_t gRespBuffer[SHARED_BUFFER_SIZE];

/* The low-level flash driver is supplied by the platform port: a whFlashCb
 * callback table plus port-defined context and configuration structures. The
 * actual driver is hardware-specific and omitted here. On a host you can drop
 * in the bundled RAM simulator (wolfhsm/wh_flash_ramsim.h, WH_FLASH_RAMSIM_CB)
 * to run this example without real flash. */
extern const whFlashCb myFlashCb;
extern void*           myFlashContext;
extern const void*     myFlashConfig;

int main(void)
{
    int rc;

    /* 1. Describe the shared-memory transport and its callback table */
    whTransportMemConfig transportCfg[1] = {{
        .req       = gReqBuffer,
        .req_size  = SHARED_BUFFER_SIZE,
        .resp      = gRespBuffer,
        .resp_size = SHARED_BUFFER_SIZE,
    }};
    whTransportServerCb         transportCb[1]  = {WH_TRANSPORT_MEM_SERVER_CB};
    whTransportMemServerContext transportCtx[1] = {0};

    /* 2. Bind the transport to the server comm configuration */
    whCommServerConfig commCfg[1] = {{
        .transport_cb      = transportCb,
        .transport_context = (void*)transportCtx,
        .transport_config  = (void*)transportCfg,
        .server_id         = 1, /* server identifier */
    }};

    /* 3. Build the NVM context on top of the NVM-flash layer and flash driver
     ↪  */
    whNvmFlashConfig nvmFlashCfg[1] = {{
        .cb      = &myFlashCb,
        .context = myFlashContext,
        .config  = myFlashConfig,
    }};
    whNvmFlashContext nvmFlashCtx[1] = {0};
```

```
    whNvmCb           nvmCb[1]       = {WH_NVM_FLASH_CB};

    whNvmConfig nvmCfg[1] = {{
        .cb      = nvmCb,
        .context = nvmFlashCtx,
        .config  = nvmFlashCfg,
    }};
    whNvmContext nvm[1] = {0};

    rc = wh_Nvm_Init(nvm, nvmCfg);
    if (rc != WH_ERROR_OK) {
        printf("wh_Nvm_Init failed: %d\n", rc);
        return 1;
    }

    /* 4. Initialize wolfCrypt and seed the server's crypto context RNG */
    whServerCryptoContext crypto[1] = {0};
    wolfCrypt_Init();
    wc_InitRng_ex(crypto->rng, NULL, INVALID_DEVID);

    /* 5. Assemble the server configuration */
    whServerConfig serverCfg[1] = {{
        .comm_config = commCfg,
        .nvm         = nvm,
        .crypto      = crypto,
        .devId       = INVALID_DEVID, /* software crypto; use a crypto-callback
                                       * devId to offload to hardware */
    }};

    /* 6. Initialize the server context */
    whServerContext server[1] = {0};
    rc = wh_Server_Init(server, serverCfg);
    if (rc != WH_ERROR_OK) {
        printf("wh_Server_Init failed: %d\n", rc);
        return 1;
    }

    /* 7. Mark the transport connected once the shared memory is ready */
    wh_Server_SetConnected(server, WH_COMM_CONNECTED);

    /* 8. Service client requests. HandleRequestMessage is non-blocking and
     * returns WH_ERROR_NOTREADY when no request is pending. */
    while (1) {
        rc = wh_Server_HandleRequestMessage(server);
        if (rc != WH_ERROR_OK && rc != WH_ERROR_NOTREADY) {
            break; /* fatal transport error */
        }
    }

    wh_Server_Cleanup(server);
    return 0;
}
```

3.3 Deep Dive

Now that you have seen the code, lets dive a little deeper and explain what is going on. We will use the server-side code as an example unless otherwise noted, as it also contains the majority of the steps necessary to initialize the client side.

3.3.1 Transport Configuration

The transport is responsible for moving raw bytes between the client and the server. This example uses the built-in two-buffer shared-memory transport declared in `wolfhsm/wh_transport_mem.h`: the client writes requests into one buffer and reads responses from the other, while the server does the reverse.

`whTransportMemConfig` binds the two shared buffers and their sizes, while the `WH_TRANSPORT_MEM_SERVER_CB` macro (and `WH_TRANSPORT_MEM_CLIENT_CB` on the client) populates a callback table that adapts this concrete transport to the abstract transport interface that the comm layer consumes. Because both sides operate on the same two buffers, the memory referenced by `.req` and `.resp` must be physically shared between the client and server. wolfHSM ships with additional transports (TCP and POSIX shared memory) that plug in the exact same way by swapping the config, context, and callback structures. See Communication Layer and Transports for the full list and details.

3.3.2 Comm Layer Configuration

The communication layer sits directly above the transport and implements the request/response protocol: message framing, sequence numbers, and endianness handling. The server binds its transport into a `whCommServerConfig`, providing the callback table, the transport context, the transport config, and a `server_id`. The client's `whCommClientConfig` is analogous but carries a `client_id` instead.

The `client_id` is significant: the server learns each client's identifier during the connection handshake, and uses it to keep per-client key caches and other resources isolated from one another. The request/response protocol these structures drive is the **split-transaction, non-blocking** model described in Client/Server Communication.

3.3.3 NVM Configuration

Non-volatile storage is layered. From the bottom up:

1. A platform **flash driver** implements the `whFlashCb` callbacks (read/program/erase) for a specific device. This is the hardware-specific piece omitted from the example above; wolfHSM also bundles a RAM-backed simulator (`wolfhsm/wh_flash_ramsim.h`, `WH_FLASH_RAMSIM_CB`) that is handy for running on a host.
2. The **NVM flash** implementation (`whNvmFlashConfig` + `WH_NVM_FLASH_CB`) turns that raw flash into an object store with wear-aware, power-fail-safe semantics. It is bound to the flash driver via the `.cb`, `.context`, and `.config` fields.
3. The generic **NVM context** (`whNvmConfig` + `whNvmContext`) presents the high-level object API to the rest of the server. `wh_Nvm_Init()` initializes it from the configuration before the server is started.

This is the same callback-driven layering used throughout wolfHSM, so the NVM flash backend can be swapped for another NVM implementation without touching the server code. See Non-Volatile Memory (NVM) and Flash Abstraction.

3.3.4 wolfCrypt Initialization

The server performs cryptographic operations with wolfCrypt, so `wolfCrypt_Init()` must be called before `wh_Server_Init()`. The server's `whServerCryptoContext` owns a wolfCrypt random num-

ber generator that must be seeded with `wc_InitRng_ex()`.

Passing `INVALID_DEVID` makes the server perform crypto in software. To offload to a hardware accelerator instead, register a wolfCrypt crypto callback and pass its device ID both to `wc_InitRng_ex()` and to the `.devId` field of the server configuration. Note that the client does *not* initialize wolfCrypt for offloaded operations: it transparently routes wolfCrypt API calls to the server by using its device ID — set in the client config's `.devId` field, or the default `WH_DEV_ID` when left `0` — read with `WH_CLIENT_DEVID(client)` after `wh_Client_Init()`. See Cryptography and wolfCrypt Integration.

3.3.5 Initializing the Server Context

`whServerConfig` aggregates the three pieces configured above — the comm config (`.comm_config`), the initialized NVM context (`.nvm`), and the crypto context (`.crypto`) — into a single configuration. `wh_Server_Init()` wires them into the `whServerContext`, which from then on serves as the handle for all server operations.

After initialization, the server must be told when the underlying transport is actually ready for communication by calling `wh_Server_SetConnected(server, WH_COMM_CONNECTED)`. Until the server is connected, `wh_Server_HandleRequestMessage()` returns `WH_ERROR_NOTREADY`. On the client side, the corresponding steps are simply `wh_Client_Init()` followed by `wh_Client_CommInit()`.

3.3.6 Processing Requests

Once connected, the server services requests by repeatedly calling `wh_Server_HandleRequestMessage()`. This function is non-blocking: it returns `WH_ERROR_NOTREADY` when no request is pending, processes and responds to a request when one is available, and only returns a hard error when the underlying transport fails. Errors that occur while *servicing* a request are reported back to the client inside the response message rather than failing the call, so the server loop only needs to break on fatal transport errors.

In this example the loop polls in a busy wait, which is fine for illustration; on real hardware this poll is typically driven by an inter-core interrupt so the HSM core can sleep between requests.

The client side mirrors this. `wh_Client_Echo()` used above is a blocking convenience helper that sends a request and polls for the matching response internally. For finer control, every client operation also exposes the underlying non-blocking split-transaction API as a `wh_Client_*Request()` / `wh_Client_*Response()` pair (for example, `wh_Client_EchoRequest()` and `wh_Client_EchoResponse()`), letting the caller send a request and poll for its response separately.

4 Architecture

4.1 Client/server libraries

wolfHSM is built as two cooperating libraries:

- Client library: linked into a client application and exposes an API for key management, NVM object storage, certificate chain verification, and other core functionality. It also integrates with wolfCrypt and allows client applications to use the wolfCrypt cryptography API directly with transparent offload to the HSM server.
- Server library: linked into a server application running in a trusted environment, typically on an isolated CPU core, and owns the actual cryptographic material, key caches, NVM storage, and hardware integration points.

4.2 Code and API organization

The codebase has the following organizational layout:

- `wolfhsm/`: Header files, both external and internal
- `src/`: Core library source code
- `test/`: Test suite
- `benchmark/`: Benchmark suite
- `examples/`: Example POSIX server and client applications, as well as client-facing demo functions for core features
- `port/`: Public platform port code. Currently only contains POSIX port, as all hardware platforms are behind vendor NDA
- `tools/`: Tools and utilities
- `docs/`: Documentation

4.2.1 Client API

The core client API is defined in `wolfhsm/wh_client.h`. Public client functions are prefixed with `wh_Client_`. Most client applications only need the interfaces provided by this header and the wolfCrypt API. The AUTOSAR SHE client compatibility interface is defined separately in `wolfhsm/wh_client_she.h`.

A client's connection to a server is represented by the `whClientContext` structure. After initialization with the appropriate transport configuration, this context serves as the handle for all client-side wolfHSM operations, ensuring that requests and responses are routed through the correct communication interface. For most operations, state is held by the client, unless the state is cryptographically sensitive.

4.2.2 Server API

The server API is divided across multiple header files in `wolfhsm/wh_server*.h` and may be consumed on a per-module basis. Core server functionality like initialization, teardown, and request processing are defined in `wolfhsm/wh_server.h`. Additional capabilities are exposed through module-specific headers.

A server's connection to a client is represented by the `whServerContext` structure. Unlike the client context, which primarily encapsulates a single communication interface, the server context aggregates a wider set of stateful and platform-specific resources. This includes runtime-initialized callbacks for module implementations, persistent and transient state required to service client requests, and integrations with platform services such as NVM, DMA access, and logging. Some of these resources and state are per-client and others are global.

Once initialized with the appropriate use-case-specific configuration, the server context serves as the central handle for all server-side wolfHSM operations, ensuring that requests and responses are routed through the correct communication interface and processed using the appropriate module implementations and platform resources.

4.2.3 Important Source Modules

- `src/wh_client*.c`: client-side API, request construction, and crypto callback integration.
- `src/wh_server*.c`: server-side functionality including request processing, crypto, key cache control, etc.
- `src/wh_comm.c`: high level communication layer abstraction for server and client.
- `src/wh_transport*.c`: low level transport abstraction with pluggable back-ends (memory buffers, TCP, POSIX shared memory).
- `src/wh_message*.c`: protocol encoding/decoding, fixed-size message framing, and command identifiers.
- `src/wh_nvm*.c`: non-volatile storage abstraction with pluggable back-ends.

4.2.4 Compiling Source Files

When building wolfHSM for the client or server, it is safe to wildcard include all C source files in the `src/` directory and pass them to your compiler. All wolfHSM code is internally protected by library configuration macros ensuring that only relevant code will be conditionally compiled based on the supplied configuration. For more information see Library Configuration.

4.3 Client/Server Communication

All client/server communication follows a strict request/response model. A request is a message sent from a client to a server, and a response is the corresponding reply from the server to that specific request. Each response uniquely maps to a single outstanding request.

4.3.1 Client Communication

The client API uses a **split-transaction, non-blocking** communication model composed of two explicit phases:

1. **Send Request**: The client invokes an API function to transmit a request to the server. The function returns immediately and may report a “not ready” condition if the underlying transport is unable to accept the request, in which case the caller is expected to retry.
2. **Receive Response**: The client invokes the corresponding receive API to poll for the server’s response to a previously sent request. This operation is non-blocking and may return “not ready” if the response has not yet arrived.

Each request is tagged with a sequence number that is internally validated against the received response, ensuring that the client processes the correct reply and does not consume a stale or mismatched message.

4.3.2 Server Communication

The server is designed to operate within an event-driven loop for each connected client, continuously polling for pending requests.

After initialization, the server repeatedly calls `wh_Server_HandleRequestMessage()` on a given `whServerContext` to check for incoming requests in a non-blocking manner. The function returns “not

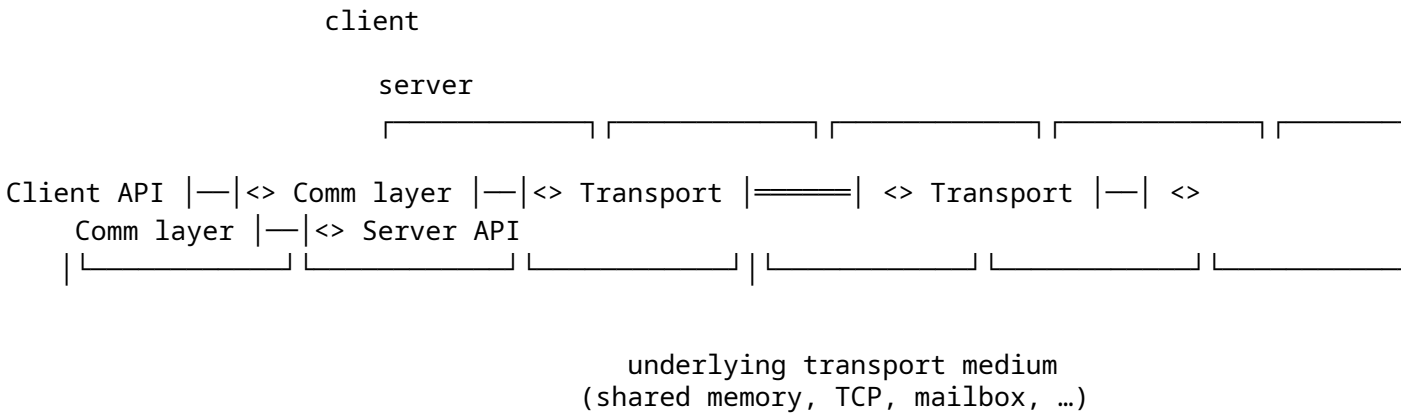
ready" when no request is pending. When a request is available, it is processed internally and dispatched to the appropriate module handler, which generates the corresponding response message that is sent back to the client.

Errors that occur while servicing a client request are treated as non-fatal and are reported to the client as part of the response. The `wh_Server_HandleRequestMessage()` function itself only fails when an error occurs in the underlying communication stack while receiving or transmitting a request.

4.3.3 Communication Stack

wolfHSM's communication is structured as a two-layer stack that sits beneath every client and server API. The **comm layer** owns the request/response framing described above: the versioned packet header, sequence numbering, and the send/receive primitives that the higher-level APIs build on. Beneath it, the **transport layer** is a thin, pluggable interface whose only responsibility is to move a single packet between the two endpoints. The comm layer reaches the underlying medium exclusively through this interface, so the same client and server code runs unchanged whether packets travel over shared memory on a multi-core SoC, a hardware mailbox, TCP, or anything else — supporting a new medium means supplying a transport, not modifying the core library.

The stack is symmetric across the client/server boundary:



For the packet header layout, sequence-numbering rules, and the reference transports wolfHSM ships with, see Communication Layer and Transports.

4.4 Library Configuration

wolfHSM is configured at build time using a set of `WOLFHSM_CFG_XXX` preprocessor macros. These configuration macros are used to:

- Enable or disable library features
- Control sizing and resource limits (e.g. message buffer size, NVM object count, key cache size, etc.)
- Specify user-supplied override functions for platform-specific behavior (e.g. cache operations, system time retrieval, `printf` and stdout functionality, etc.)

Each configuration option provides a sane default that allows the library to build and operate in its default configuration without user intervention. All options may be overridden at build time to customize the library for a specific use case or platform.

The only configuration that **must** be supplied by the user is a mechanism to obtain the system time, either by defining `WOLFHSM_CFG_PORT_GETTIME` or by explicitly disabling system time support with `WOLFHSM_CFG_NO_SYS_TIME`. In practice, this requirement is typically handled by the port layer provided with a given platform and is only relevant when porting wolfHSM to a new environment.

Configuration macros may be provided directly on the compiler command line, or more commonly, via a user-defined configuration header named `wolfhsm_cfg.h`. When using a configuration header (recommended when more than a small number of options are being customized), the user must:

1. Create a file named `wolfhsm_cfg.h` containing `#define` statements for the desired `WOLFHSM_CFG_XXX` values
2. Ensure the file resides in a directory included in the compiler's header search path (for example, via `-I`)
3. Define the `WOLFHSM_CFG` macro when invoking the compiler (for example, `-DWOLFHSM_CFG`)

Once these steps are completed, wolfHSM will use the user-defined configuration values in place of the internal defaults.

Note that the default base configuration and global library settings are defined in `wolfhsm/wh_settings.h`. Every wolfHSM source file includes this header first to establish the configuration environment. When `WOLFHSM_CFG` is defined, `wh_settings.h` conditionally includes `wolfhsm_cfg.h`, ensuring that user-supplied overrides are applied before any internal defaults are used.

For an exhaustive list of all wolfHSM config macros, see .

4.5 Internals Deep Dive: Modular Architecture

wolfHSM is highly modular and built around an internal architecture that uses generic interfaces for core subsystems such as communications/transport, non-volatile memory (NVM) storage, and logging. Concrete implementations of these interfaces are selected and bound at runtime through a configuration structure supplied upon initialization. This design enables independent selection of both software and hardware implementations for each subsystem.

Put another way: Instead of binding to a single implementation or hardware driver stack, each system-facing module exposes an interface that is implemented via callbacks. The core library depends only on these interfaces, not on any platform specifics.

The following snippet is an abstract example of how this looks for an arbitrary wolfHSM "component":

```
#include "wolfhsm/component.h"        /* wolfHSM abstract API reference for a
 ↪  component */
#include "port/vendor/mycomponent.h"  /* Platform specific definitions of
 ↪  configuration
                                       * and context structures, as well as
 ↪  declarations of
                                       * callback functions */

/* Provide the lookup table (vtable) for function callbacks for mycomponent.
 *Note the type is the abstract type provided in wolfhsm/component.h */
whComponentCb my_cb = {MY_COMPONENT_CB};

/* Fixed configuration data.  Note that pertinent data is copied out of the
 ↪  structure
 * during init() */
const myComponentConfig my_config = {
    .my_number = 3,
    .my_string = "This is a string",
}

/* Static allocation of the dynamic state of the myComponent. */
myComponentContext my_context = {0};
```

```
/* Initialization of the component using platform-specific callbacks */
const whComponentConfig comp_config = {
        .cb = my_cb,
        .context = my_context,
        .config = my_config
    };
whComponentContext comp_context = {0};
int rc = wh_Component_Init(comp_context, comp_config);

rc = wh_Component_DoSomething(comp_context, 1, 2, 3);
rc = wh_Component_CleanUp(comp_context);
```

Modules in wolfHSM that are currently implemented in this way are:

- Transports: define how bytes move between client and server. The core library only cares about send/receive semantics and state management.
- NVM: define how persistent storage on the server is formatted and accessed.
- Logging: Internal logging uses configurable logging backends that implement their own schema and read/write/erase/export operations.
- Mutex abstraction: Internal resource serialization is written against a generic “lock” interface that uses port-specific mutex abstractions.

5 Features

This chapter provides a detailed overview of the high level features that wolfHSM provides. Each section is intended to convey *what* a given feature does, what functionality it exposes, and what a developer can build with it. Concrete API usage and signatures are deferred to the client and server API references in and .

5.1 Table of Contents

5.2 Cryptography and wolfCrypt Integration

wolfHSM uses wolfCrypt as its cryptographic provider on both sides of the client/server boundary. On the client, applications call the standard wolfCrypt API directly and the operation runs transparently on the server. On the server, the full set of wolfCrypt's software algorithms is available out of the box, with optional acceleration from port-supplied hardware drivers.

5.2.1 Transparent Offload via Crypto Callbacks

Clients can use the wolfCrypt API directly because of wolfCrypt's crypto callback (cryptoCb) framework. Crypto callbacks let you override selected algorithms at runtime by registering a callback against a device identifier (`devId`). Most wolfCrypt functions take a `devId`, and when it matches a registered device the call is dispatched through that callback instead of running locally.

The wolfHSM client library registers a crypto callback that turns each supported wolfCrypt call into a request/response exchange with the server. The same wolfCrypt source can be retargeted to the HSM by changing only the `devId` — nothing else in the application changes. Each client context registers a device ID chosen by the application in the `.devId` field of `whClientConfig`; leaving the field `0` selects the default `WH_DEV_ID`. `wh_Client_Init()` registers the ID and binds it to that context, and `wh_Client_Cleanup()` unregisters it. At a wolfCrypt call site the application can either read the ID back from the context with the `WH_CLIENT_DEVID(client)` macro, or simply pass the same constant it placed in the config — convenient where the client context is not in scope. Because each client can own a distinct `devId`, a single process can run multiple client connections (to one server or several) and each wolfCrypt call is serviced by exactly the client whose `devId` it was initialized with; a multi-client process must configure a distinct, nonzero `devId` for every client.

In addition to the configured per-client ID, every `wh_Client_Init()` registers the two process-global device IDs:

- `WH_DEV_ID` is registered with the same unified callback as a configured `devId`, so it behaves identically — including honoring the DMA dispatch mode. It is also the ID a client is bound to when its config leaves `.devId` 0.
- `WH_DEV_ID_DMA` (present only with `WOLFHSM_CFG_DMA`) is registered with the DMA-only callback: operations always use the DMA request forms, and algorithms without a DMA variant fail rather than falling back to the standard path. It is reserved for this purpose and is not valid as a configured `.devId`.

The global IDs preserve the behavior of earlier wolfHSM releases: an application with a **single client per process** needs no devId configuration at all and can keep passing `WH_DEV_ID` (or `WH_DEV_ID_DMA`) straight to wolfCrypt functions, exactly as before — they are always registered and available after `wh_Client_Init()`. Because these registrations are process-global and keyed on the integer value, each `wh_Client_Init()` rebinds them to the most recently initialized client, and **any** client's `wh_Client_Cleanup()` unregisters them. In a multi-client process they are therefore unreliable and should not be passed to wolfCrypt; use the per-client configured IDs instead. Both values are overridable at compile time (see Configuration).

Registered device IDs occupy slots in wolfCrypt's fixed-size crypto-callback table (`MAX_CRYPTO_DEVID_CALLBACKS`, default 8): the global IDs occupy one slot each, shared by all clients in the process (every init rebinds the same table entries), and each distinct configured `devId` adds one more. `wh_Client_Cleanup()` releases the client's slots. Applications that run many simultaneous clients in one process may need to raise the wolfCrypt limit.

In effect the callback layer is a transparent RPC framework for wolfCrypt: clients write ordinary wolfCrypt code, and wolfHSM handles request marshaling, transport, dispatch, and response delivery underneath. It also makes prototyping easy — develop against a local wolfCrypt instance, then switch to the HSM by toggling one parameter once the server is available.

5.2.2 Supported Algorithms

The wolfHSM server exposes the full set of wolfCrypt software algorithms, and the client crypto callback supports transparent offload for the most commonly used algorithm families:

- **Symmetric ciphers**: AES in CBC, CTR, ECB, GCM, and CCM modes; AES key wrap
- **Hashing**: SHA-1, SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512), SHA-3
- **Message authentication**: HMAC (over the supported hash functions) and CMAC
- **Asymmetric**: RSA (encryption, signing, key generation), ECC (ECDSA, ECDH), Ed25519, Curve25519
- **Random number generation**: DRBG/RNG backed by the server's entropy source
- **Post-quantum cryptography**: ML-DSA (FIPS 204) and ML-KEM (FIPS 203)

For the authoritative list of algorithms, parameter ranges, and options, see the wolfCrypt API reference. An algorithm not yet wired through the crypto callback can still be used locally against the client's own wolfCrypt instance — only operations dispatched to the client's `devId` are offloaded.

5.2.3 Referencing Keys by ID

When a client offloads an operation, it usually does *not* send the key with the request. The key lives in the server keystore under a numeric **key ID**, and the client refers to it by that ID alone. The bytes never cross the client/server boundary — the client holds only the ID, and the server looks up the material when it runs the operation. This is what lets an HSM guard a private key while still letting a client sign or decrypt with it.

A wolfCrypt key object is tied to a server-side key ID with a per-algorithm SetKeyId call. Every offloaded algorithm has one — wh_Client_RsaSetKeyId, wh_Client_EccSetKeyId, wh_Client_AesSetKeyId, wh_Client_Ed25519SetKeyId, wh_Client_Curve25519SetKeyId, wh_Client_CmacSetKeyId, wh_Client_MlDsaSetKeyId, and so on (each with a matching GetKeyId). You initialize an ordinary wolfCrypt key struct with the client's devId, associate it with a key ID instead of loading key bytes, and call wolfCrypt as usual:

```
RsaKey  rsa;
whKeyId keyId = 4; /* keyId 4 must be resident on the server */

/* Initialize the RSA key context to use wolfHSM offload via the
 * devId of an initialized client context */
wc_InitRsaKey_ex(&rsa, NULL, WH_CLIENT_DEVID(client));

/* Bind the key object to the server-side key */
wh_Client_RsaSetKeyId(&rsa, keyId);

/* Use wolfCrypt as normal — signing runs on the server and the
 * private key never leaves the HSM */
sigLen = wc_RsaSSL_Sign(msg, msgLen, sig, sizeof(sig), &rsa, &rng);
```

The same ID can name a key the client just cached, one provisioned into NVM at the factory, or one the server generated and never exported — the client uses it the same way in every case. How IDs are assigned and structured is covered under Keystore.

5.2.4 Hardware Acceleration and Crypto Affinity

Many of the platforms wolfHSM targets ship a dedicated crypto accelerator alongside their secure core. The server can use these accelerators per-algorithm through the same crypto callback mechanism: a port-supplied callback, registered at server init, redirects supported operations to the vendor's hardware driver, and anything not implemented in hardware falls back to wolfCrypt software. Which algorithms are accelerated depends on the silicon and is documented in each platform's port.

Clients control whether a given crypto request should prefer hardware or software execution through the **crypto affinity** API. Affinity is a per-client setting with two values:

- WH_CRYPTO_AFFINITY_HW (default): the server attempts to execute the operation using the configured hardware crypto device. If the server was not configured with a valid hardware device ID, or if the requested algorithm is not implemented in hardware, the request transparently falls back to wolfCrypt's software implementation.
- WH_CRYPTO_AFFINITY_SW: the server always executes the operation using wolfCrypt's software implementation, bypassing any registered hardware device.

Affinity is stored on the client and sent in the header of every crypto request, so a change takes effect on the next operation with no extra round-trip.

The affinity is set and queried using wh_Client_SetCryptoAffinity and wh_Client_GetCryptoAffinity. See the client API reference for the precise signatures.

5.2.5 Blocking and Non-Blocking Interfaces

Operations invoked through the standard wolfCrypt API are **blocking**: the call does not return until the server responds (or the transport fails). This matches what applications already expect from wolfCrypt and is the simplest way to port existing code to wolfHSM.

For non-blocking, split-transaction behavior, wolfHSM also exposes native client crypto APIs in wolfhsm/wh_client_crypto.h that follow the same send-request / receive-response pat-

tern as the rest of the client API. These come as paired `wh_Client_<Algorithm>Request` and `wh_Client_<Algorithm>Response` calls, so a caller can issue a request, do other work, and poll for the result later. They cover a subset of algorithms (see the client API reference), and blocking and non-blocking calls must not be interleaved on the same `whClientContext` while a request is outstanding.

> **Note**: Because the standard wolfCrypt API is blocking on the client side, applications that need to overlap crypto work with other activity should either use the native non-blocking client API, or run their wolfCrypt calls from a dedicated thread with its own `whClientContext`. See Concurrency Support for guidance on multi-threaded usage.

5.2.6 Crypto Operation Timeouts

A blocking call waits until the server responds (see Blocking and Non-Blocking Interfaces). If the server never answers — it crashed, or the transport stalled — the call would otherwise wait forever. The optional request-timeout feature (`WOLFHSM_CFG_ENABLE_TIMEOUT`) bounds that wait: once the timeout elapses the call returns `WH_ERROR_TIMEOUT` instead of hanging, and the client context stays usable so the application can recover or retry. It is purely client-side — the server is unaware of it — and applies to any blocking request/response, offloaded crypto being the main case.

A timeout is configured per client through the `respTimeoutConfig` field of the `whCommClientConfig`. The client starts the timer when it sends a request, checks it while polling for the response, and stops it once the response arrives. Setting `respTimeoutConfig` to `NULL` disables the timeout (the client waits indefinitely), so the feature can be compiled in but left off for individual clients.

wolfHSM has no built-in notion of time, so the actual time measurement is supplied by the platform through a small callback table (`whTimeoutCb`) of init, set, start, stop, and check-expired functions. This keeps the core free of OS dependencies. The POSIX port ships a ready-made implementation (`POSIX_TIMEOUT_CB`, based on `CLOCK_MONOTONIC`); other platforms supply their own. Durations are expressed in microseconds, with `WH_MSEC_TO_USEC()`, `WH_SEC_TO_USEC()`, and `WH_MIN_TO_USEC()` helper macros for readability. An application may also register an optional expired callback that runs when the timer elapses and can override the expiration — for example, restart the timer to grant a legitimately slow operation more time.

5.3 Non-Volatile Memory (NVM)

wolfHSM exposes persistent storage to the rest of the server as an object store rather than as raw bytes. Keys, certificates, monotonic counters, and any other long-lived state that the server needs to survive a reset are stored as discrete objects in this store, each addressed by a stable identifier and carrying its own metadata. The object store sits on top of a pluggable backend, which in turn talks to platform flash through a thin abstraction provided by the port. This layered design keeps the higher-level subsystems (keystore, certificate manager, image manager, counters) independent of the specifics of any particular flash device.

The NVM library is engineered for fail-safe operation: every mutating operation is structured so that interruption at any point — including loss of power partway through a write or erase — leaves the store in a recoverable state on the next initialization. This property is what allows the rest of wolfHSM to treat NVM as a reliable substrate for security-critical state.

5.3.1 High Level NVM Interface

The NVM library presents non-volatile storage as a collection of opaque, variable-sized **objects**. Each object has three parts:

- A unique 16-bit identifier (`whNvmId`) that the rest of the server uses to refer to it
- A fixed-size **metadata** record describing the object (see Object Metadata and Access Attributes)

- A variable-length **payload** of arbitrary bytes

Applications and higher-level subsystems do not address NVM by byte offset; they create, read, enumerate, and destroy whole objects through the `wh_Nvm_*` API. This object orientation is what allows the keystore, certificate manager, and counter subsystems to share a single backing store without colliding: each subsystem owns a range of identifiers and a set of metadata flags, and the underlying NVM layer is unaware of what the objects mean.

The core operations exposed by the interface are:

- **Add**: write a new object with caller-supplied metadata and payload. Duplicate identifiers are accepted at the NVM layer; the most recently written instance is the one returned on subsequent reads, which provides an in-place update semantic at higher layers.
- **Read**: retrieve all or part of an object's payload by identifier and byte offset, allowing large objects (firmware images, certificate chains) to be streamed out incrementally.
- **List / GetMetadata**: enumerate objects matching an access/flags filter and retrieve metadata for a specific identifier without touching the payload, which is what the server uses to drive directory-style operations and policy enforcement.
- **Destroy**: remove one or more objects. Removal is implemented as a regeneration of the partition with the listed objects omitted, which both deletes the entries and reclaims their space in a single atomic operation.
- **GetAvailable**: query free space and the amount of space that would be reclaimed by a compaction, so callers can make informed decisions before issuing large writes.

All mutating operations are **atomic and power-loss tolerant**: any interruption either leaves the store as it was before the operation or, if the operation completed past its commit point, as it was after. The NVM library does not return success until the new state is durably committed to flash. This guarantee is what makes it safe for the server to commit a key, increment a counter, or update a certificate without an intervening cleanup pass if the system is reset mid-operation.

The library also exposes an explicit **compaction / reclamation** model. Because objects are added by writing into free space rather than overwriting in place (a property of the underlying flash semantics), space occupied by destroyed or superseded objects is not immediately reusable. Compaction can be triggered implicitly by `wh_Nvm_AddObjectWithReclaim()` when the next add would otherwise fail for lack of space, or explicitly by calling `wh_Nvm_DestroyObjects()` with an empty list. Either path regenerates the active partition with only the live objects present and reclaims everything else.

When wolfHSM is built with `WOLFHSM_CFG_THREADSAFE`, each NVM context carries an embedded lock. The lock's lifecycle is managed by `wh_Nvm_Init()` and `wh_Nvm_Cleanup()`, but acquiring and releasing it around operations is the caller's responsibility — the NVM API functions deliberately do not lock internally so that multiple operations can be grouped under a single critical section. See Concurrency Support for the broader threading model.

5.3.2 Object Metadata and Access Attributes

Every NVM object is described by a `whNvmMetadata` record carrying:

- `id`: the unique `whNvmId` identifier
- `len`: the payload length in bytes
- `label`: a fixed-size (`WH_NVM_LABEL_LEN`) byte array that callers may use as a user-defined name or tag for the object
- `access`: a bitfield of access permissions (`whNvmAccess`) describing who may interact with the object
- `flags`: a bitfield of `whNvmFlags` describing policy and behavior

The metadata is written alongside the payload and travels with the object for its entire lifetime. Higher-level subsystems use these fields both to identify objects (the keystore, for example, decodes the structure of `id` to distinguish keys from certificates and counters) and to enforce policy on every access.

The flags field carries the policy attributes that subsystems use to gate operations:

- WH_NVM_FLAGS_NONMODIFIABLE: the object cannot be overwritten or destroyed through the policy-checked APIs
- WH_NVM_FLAGS_NONDESTROYABLE: the object cannot be destroyed (but may still be modified)
- WH_NVM_FLAGS_NONEXPORTABLE: the object's payload cannot be read back out through the policy-checked APIs
- WH_NVM_FLAGS_SENSITIVE: marks the object as holding secret material, so subsystems can apply zeroization and audit behavior accordingly
- WH_NVM_FLAGS_EPHEMERAL: the object should not be cached or committed
- WH_NVM_FLAGS_LOCAL: the object was generated locally on the server (as opposed to imported)
- WH_NVM_FLAGS_USAGE_*: key usage policy bits (ENCRYPT, DECRYPT, SIGN, VERIFY, WRAP, DERIVE) consumed by the keystore to constrain how a key may be used; see Key Usage Policies

The NVM library exposes both a raw and a policy-checked variant of the mutating and reading APIs (wh_Nvm_AddObject vs. wh_Nvm_AddObjectChecked, wh_Nvm_DestroyObjects vs. wh_Nvm_DestroyObjectsChecked, wh_Nvm_Read vs. wh_Nvm_ReadChecked). The checked variants honor the flags above and return WH_ERROR_ACCESS when the requested operation would violate them; the unchecked variants are used by server-internal code paths that need to manage the state itself (for example, to clear NONMODIFIABLE during a controlled revocation flow). Because policy enforcement happens server-side at the NVM layer, no client request can bypass it.

The access field is used to express coarser-grained permissions (owner / other / user buckets, with read/write/exec/special bits) that higher layers may consult, and is the primary filter used by wh_Nvm_List() when enumerating objects.

5.3.3 NVM Backends

The wh_Nvm_* API is implemented against a backend callback table (whNvmCb) that abstracts the details of how objects are actually laid out on storage. The core library does not depend on any particular backend — selecting a backend is part of server configuration, and ports or applications can supply their own implementations against the same interface. wolfHSM ships with two reference backends, both built on top of the flash abstraction:

- **nvm_flash** (wh_nvm_flash.c): the default backend, suitable for flash devices with small write granularity (8 bytes or less). It manages two equal-sized partitions in flash, with one designated as active at any time. New objects are added by programming directly into free space at the end of the active partition, which keeps write amplification low for read-heavy and append-dominated workloads. A directory of object state is cached in RAM and rebuilt from flash at initialization. Destruction of objects (and explicit compaction) is performed by regenerating the inactive partition with only the surviving objects, then atomically switching the active partition pointer and erasing the old one. An interruption before the switch leaves the previous partition intact; an interruption after the switch is recovered by completing the erase of the now-inactive partition on the next boot.
- **nvm_flash_log** (wh_nvm_flash_log.c): an alternative backend designed for flash devices with **large write granularity** (e.g. 64 bytes) where every program operation must be aligned and padded to that boundary. It also uses a two-partition layout, but caches the entire active partition in RAM and rewrites the whole inactive partition on every mutation. Each partition header carries a monotonic epoch counter, and the partition with the highest epoch is treated as authoritative on the next initialization. The implementation favors simplicity and a uniform write pattern at the cost of higher write amplification, which is acceptable on the read-heavy workloads it is intended for. Selected at build time via WOLFHSM_CFG_SERVER_NVM_FLASH_LOG.

Both backends bind to a whFlashCb flash driver supplied by the port; the choice between them is a function of the underlying flash device's program granularity and the application's write profile, not of any user-facing feature. Ports targeting microcontrollers with conventional NOR flash typically use

nvm_flash; ports targeting devices whose program operation is fundamentally a 32- or 64-byte page write are better served by nvm_flash_log.

5.3.4 Flash Abstraction

The lowest layer of the NVM stack is the whFlashCb interface, a small callback table that the port supplies to describe how to read, program, erase, and verify the platform's flash. The NVM backends — and any user-supplied backend — speak only through this interface, which keeps them entirely portable across flash devices.

The interface comprises:

- Init / Cleanup: lifecycle management for the underlying driver
- PartitionSize: returns the partition size, which is also the minimum erase granularity and the alignment used by the NVM backends
- Read: copy bytes out of flash at a given offset
- Program: write bytes into previously-erased flash at a given offset
- Erase: erase one or more partitions back to their blank state
- Verify: compare flash contents against a buffer, used after programming to confirm the write succeeded
- BlankCheck: confirm that a region is in the erased state, used during recovery and partition selection
- WriteLock / WriteUnlock: optional protection against accidental programming or erasure of a region

wolfHSM ships with two reference flash drivers usable on host platforms and in testing:

- **POSIX file-backed flash** (port/posix/posix_flash_file.c): persists flash contents to a host file, suitable for development, simulation, and the POSIX server example
- **RAM-backed flash simulator** (wolfhsm/wh_flash_ramsim.h): emulates flash semantics (erase-then-program, partition alignment, configurable erased-byte value) entirely in RAM, used by the test suite and useful when bringing up a new port

Vendor-supplied flash drivers ship with the platform ports under port/<vendor>/. New platforms are integrated into wolfHSM by implementing the whFlashCb callback set against the device's flash controller; nothing in the NVM library above this layer needs to change.

Write-through requirement (port maintainers). wolfHSM's power-loss guarantees assume the port's Program and Verify callbacks are write-through to the physical medium: Program must make the data durable before it returns, and Verify must read back from the medium rather than from any volatile write cache. A backend that buffers writes in a cache that can be lost on power failure breaks this assumption — on the next boot a committed object can roll back to a prior value. For stateless key material this is only a durability concern, but for **stateful or monotonic objects it is a security issue**: a rolled-back LMS or XMSS private key reuses a one-time signature index, enabling forgery, and a rolled-back monotonic counter defeats anti-rollback and replay protection. wolfHSM cannot detect or enforce this property, so a port whose flash controller caches writes must either disable that caching or issue an explicit flush before Program/Verify return.

5.3.5 Optional NVM Backing

The NVM subsystem described above is **optional**. A server can be initialized with whServerConfig.nvm == NULL, in which case it runs with no persistent object store at all. This suits clients and cores that only need cached-key cryptography and have no flash available for an NVM partition — at the cost of a reduced feature set, since everything that depends on persistent storage becomes unavailable.

With no NVM, the keystore is effectively cache-only. A key is served from the RAM key cache when present; a cache miss would normally fall back to NVM, but with no NVM configured it simply reports `WH_ERROR_NOTFOUND` — the same result as if the key were absent from the store. Keys are made available by *priming* the cache out of band: either by caching key material directly on the server, or by having the client supply wrapped keys that are unwrapped directly into the cache.

What works with no NVM:

- Cryptographic operations against keys that are primed in the cache.
- Key caching, eviction, and (cache-only) erase.
- SHE encrypt/decrypt/CMAC and secure boot against keys primed in the cache.
- Key wrap/unwrap and unwrap-and-cache, provided the wrapping key (KEK) is primed in the cache.

What requires NVM, and so fails gracefully at runtime when it is absent (returning an error rather than crashing):

- The NVM object request API (list/read/add/destroy).
- Certificate-chain verification against trusted roots stored in NVM.
- Monotonic counters.
- Committing a cached key to persistent storage (`wh_Server_KeystoreCommitKey`).
- SHE key persistence and the SHE PRNG seed (`LOAD_KEY` of non-RAM keys, `INIT_RND`, `EXTEND_SEED`), and image-signature loading.

  Note: When global keys (`WOLFHSM_CFG_GLOBAL_KEYS`) are enabled, the shared global key cache normally lives inside the NVM context. With no NVM there is no shared store, so global keys (USER `0`) are served from the per-context local cache instead. They remain usable when primed, but are not shared across server contexts as they would be with NVM present.

When NVM **is** configured, all of the above behavior is unchanged.

5.4 Keystore

The keystore manages the lifecycle of cryptographic key material on the server. It sits on top of NVM and is the layer every crypto operation goes through to reference a key. Clients refer to keys by a stable 16-bit identifier, not by the key bytes; the material stays server-side and is only returned through explicit, policy-checked operations. This is what lets the server enforce per-key usage policy, isolate keys between clients, and offload bulk crypto to hardware without exposing key bytes outside the trust boundary.

The keystore has three responsibilities that the rest of this section covers in turn:

- Managing a fast working set of keys in RAM (the **key cache**) layered over the slower NVM object store, with explicit commit and load operations between the two
- Enforcing **isolation between clients**, so that a key cached by one client is not visible to another — and, optionally, relaxing this isolation for explicitly designated global keys
- Implementing **wrapped keys** that can leave the server under the protection of a server-resident key encryption key, and **usage policies** that constrain how every individual key may be used at every crypto request

5.4.1 Key Cache, Key IDs, and NVM Backing Store

The keystore is two-tier: a fixed-size **key cache** in server RAM holds the working set of keys, and persistent **NVM** behind it holds the keys that must survive a reset. Every key in use lives in the cache; NVM is the durable copy, loaded into a cache slot on demand.

Keys are named by a 16-bit identifier (whKeyId), which has two forms — a simple one the client uses and a fuller one the server uses internally:

- **Client-side**: Each client gets a dedicated namespace of 255 key identifiers that are specific to and only accessible by that client. These IDs range from [1, 255], where 0 is the reserved sentinel value WH_KEYID_ERASED used internally to mark empty key slots (this sentinel value is also used to request a dynamically assigned ID for a key cache operation — see the keystore API documentation for more information). The client can also set a flag bit in the keyId top byte to ask for special handling — bit 8 for a global key, bit 9 for a wrapped key. That is all a client ever deals with, and the WH_CLIENT_KEYID_MAKE_* macros in wolfhsm/wh_client.h set those flags for it.
- **Server-side**: internally every key has a globally unique id that also encodes *what* the key is and *who* owns it. When a request arrives, the server expands the client's provided keyId number into this full form, and collapses it back on the way out (wh_KeyId_TranslateFromClient() and its inverse). Client code never touches the internal fields.

The server-side whKeyId packs three fields into its 16 bits:

- **TYPE** (top 4 bits): the kind of object — WH_KEYTYPE_CRYPTO for ordinary crypto keys, WH_KEYTYPE_SHE for AUTOSAR SHE keys, WH_KEYTYPE_COUNTER for monotonic counters, WH_KEYTYPE_WRAPPED for wrapped-key metadata, and WH_KEYTYPE_NVM for non-key NVM objects that share the same id space.
- **USER** (middle 4 bits): the owning client. Value 0 is reserved for the global-key namespace when WOLFHSM_CFG_GLOBAL_KEYS is enabled.
- **ID** (low 8 bits): the number the client chose.

The USER field is what gives each client its own private key space. The server fills it with the connection's client id (assigned at init and checked against WH_CLIENT_ID_MAX), so when two clients both use "key 5" they map to different whKeyId values and cannot touch each other's key. Every client sees the same [1, 255] range, and the ranges never overlap.

Turning to the cache itself: it is statically allocated inside the server context (wolfHSM uses no dynamic memory) and sized at build time by two pairs of macros:

- WOLFHSM_CFG_SERVER_KEYCACHE_COUNT × WOLFHSM_CFG_SERVER_KEYCACHE_BUFSIZE: the number of *regular* slots and the largest key that fits in one
- WOLFHSM_CFG_SERVER_KEYCACHE_BIG_COUNT × WOLFHSM_CFG_SERVER_KEYCACHE_BIG_BUFSIZE: the number of *big* slots and the largest key that fits in one

The two tiers keep one large key (e.g. an an ML-DSA-87 private key) from dictating the slot size for every smaller key. A cached key goes to the regular cache if it fits and the big cache otherwise; both follow the same eviction and policy rules.

Each cached key carries its full whNvmMetadata record alongside the key bytes, plus an internal **committed flag** marking whether a copy also exists in NVM. This flag is what makes the cache a true working set: when it needs a slot and none is free, the keystore evicts a key that is already committed (and can be reloaded later), but never an uncommitted one. Uncommitted keys are RAM-only, so the caller must commit them to survive eviction or reset; if the cache fills with uncommitted keys, the next cache operation returns WH_ERROR_NOSPACE instead of silently dropping material.

A client drives this tier with five operations:

- **Cache**: write key bytes and metadata into a server cache slot. The key is usable immediately but RAM-only.
- **Commit**: copy a cached key into NVM as a durable object. The cache copy is marked committed and becomes a candidate for eviction.
- **Evict**: drop the cache copy. If the key was committed, the NVM copy remains and reloads on the next reference; if it was uncommitted, the key is gone.
- **Export**: read a cached key's bytes back to the client, subject to WH_NVM_FLAGS_NONEXPORTABLE.

- **Erase**: remove the key from both cache and NVM in one operation.

Operations that take a `whKeyId` don't care whether the key is in the cache or only in NVM: on first use the server loads it from NVM into a cache slot — the implementation calls this *freshening* — and serves later operations from the cache until eviction.

Two more behaviors round out the cache model:

- A key whose metadata carries `WH_NVM_FLAGS_EPHEMERAL` is never committed to NVM, no matter what the caller requests. This suits short-lived material (session keys, transient keypairs) where an NVM write is never warranted.
- The 24-byte `label` field travels with the key for its whole lifetime in both cache and NVM. It is opaque to the keystore — a caller-supplied name or tag — and is returned alongside the key on export, so applications can tell keys apart without keeping their own mapping. The exception to this is in the SHE layer, where it is used to store SHE-specific metadata.

5.4.2 Global Keys

By default a key cached by one client is invisible to every other client, thanks to the USER field in `whKeyId`. But sometimes clients on a single HSM genuinely need to share key material. Copying such a key into every client's namespace would waste cache and NVM and complicate provisioning.

The optional **global keys** feature (`WOLFHSM_CFG_GLOBAL_KEYS`) adds a parallel keystore namespace shared by all clients. Global keys live in their own cache and their own NVM id range, but are otherwise used exactly like local keys: any operation that takes a `whKeyId` — including the wolfCrypt crypto callback — accepts a global keyId unchanged.

Internally, global keys reserve USER field value `0` (`WH_KEYUSER_GLOBAL`) — a value no client can hold if the global key feature is enabled. The global cache lives in the NVM context (`whNvmContext::globalCache`), not a server context, because server contexts are per-connection while NVM and global keys are shared across them. On each keystore operation the server checks the keyId: USER `0` routes to the global cache, anything else to the connection's local cache. There is no separate global API — `wh_Client_KeyCache`, `wh_Client_KeyCommit`, `wh_Client_KeyExport`, and the crypto callbacks all work transparently with global keyIds.

Clients designate a key as global by setting the `WH_KEYID_CLIENT_GLOBAL_FLAG` bit (bit 8) in the request keyId. The recommended way to do this is via the `WH_CLIENT_KEYID_MAKE_GLOBAL()` macro:

- `whKeyId k = WH_CLIENT_KEYID_MAKE_GLOBAL(5)` constructs a client-facing global keyId for numeric ID 5
- Passing this keyId to `wh_Client_KeyCache` causes the server to store the key in the global cache and (on commit) in the global NVM range
- Passing the same keyId from any other connected client retrieves the same key

Global keys interact with the cache and NVM tiers in the same way as local keys, including the commit/evict/freshen flow, eviction of committed-only slots, and policy enforcement at the NVM layer. The only practical differences are the cache they occupy and the visibility they grant.

> **Security note**: Because a global key is reachable by every client connected to the server, the security boundary it provides is the server itself, not any particular client. Global keys should be reserved for material that is genuinely shared across the trust domains of the connected clients — typically vendor-provisioned roots and shared symmetric keys for inter-client communication — and should not be used as a workaround for per-client key management. Per-key usage flags (see Key Usage Policies) apply to global keys exactly as they do to local keys, and should be used to constrain how a shared key may be used regardless of which client invokes the operation.

5.4.3 Wrapped Keys

A key that lives entirely inside the server is protected by its trust boundary: only the server can read its bytes, and policy is enforced before every use. Some workflows still need to move key material outside that boundary — to back keys up to off-device storage, to transport a key between systems during provisioning, or even to support wolfHSM on an HSM platform without dedicated NVM. wolfHSM's **wrapped keys** feature (`WOLFHSM_CFG_KEYWRAP`) does this safely, with the server mediating every step.

A *wrapped key* is a key whose payload — the key bytes and its `whNvmMetadata` — has been encrypted and authenticated under another key resident on the server, the **key encryption key (KEK)**. Because the KEK never leaves the server, the wrapped blob can be handed to the client, stored on a host filesystem, sent over an untrusted channel, or pushed to off-device storage, and only the server (with the same KEK) can recover the original key.

The wrap format used by wolfHSM is a length-prefixed, authenticated encryption blob. AES-GCM is the currently supported wrap cipher and is requested by passing `WC_CIPHER_AES_GCM` to the wrap APIs; the on-wire layout is:

```
[ IV (12 bytes) | AuthTag (16 bytes) | AES-GCM( metadata || key ) ]
```

The metadata is bound into the authenticated plaintext so that the wrapped blob carries not only the key bytes but also its policy, label, and identifier — a recipient cannot strip or substitute metadata without invalidating the authentication tag. The on-wire constants `WH_KEYWRAP_AES_GCM_IV_SIZE`, `WH_KEYWRAP_AES_GCM_TAG_SIZE`, and `WH_KEYWRAP_AES_GCM_HEADER_SIZE` are defined in `wolfhsm/wh_common.h` and may be used by callers to size wrap output buffers. The maximum wrappable key size is controlled by `WOLFHSM_CFG_KEYWRAP_MAX_KEY_SIZE`.

The lifecycle exposed to clients consists of three primary operations:

- **Wrap**: the client supplies plaintext key bytes, a metadata template, and the keyId of a server-resident KEK; the server encrypts the (metadata || key) blob with the KEK and returns the wrapped blob to the client. The plaintext is not written to NVM as part of this operation.
- **Unwrap-and-export**: the client supplies a wrapped blob and the KEK's keyId; the server decrypts the blob, authenticates the tag, and returns the recovered metadata and key bytes to the client. This is the operation used by host-side workflows that need to consume the key off-device, for example to inject it into a non-HSM peer.
- **Unwrap-and-cache**: the client supplies a wrapped blob and the KEK's keyId; the server decrypts the blob and installs the recovered key directly into the keystore cache as if `wh_Client_KeyCache` had been called locally with the recovered bytes. This is the more common operation in production deployments, since it lets a key live on disk in encrypted form and be hydrated into the HSM at runtime without the plaintext ever transiting the client. Clients can then commit the unwrapped key to NVM if they wish.

In all three operations the KEK is identified by its existing keyId in the keystore, must carry the `WH_NVM_FLAGS_USAGE_WRAP` usage flag, and is enforced server-side by the keystore policy machinery. A key without the `WRAP` usage flag cannot be used to wrap or unwrap regardless of any client request.

A parallel pair of APIs — `wh_Client_DataWrap` and `wh_Client_DataUnwrap` — applies the same construction to arbitrary application data rather than key material. These are useful when a client needs the same authenticated-encryption guarantee for non-key payloads using a key resident in the HSM.

> **Note**: Wrapped key identifiers are signaled on the wire by setting `WH_KEYID_CLIENT_WRAPPED_FLAG` (bit 9) in the request keyId, which the server translates internally to `WH_KEYTYPE_WRAPPED`. Clients construct wrapped-key identifiers using `WH_CLIENT_KEYID_MAKE_WRAPPED()`, and the combined wrapped-and-global form using `WH_CLIENT_KEYID_MAKE_WRAPPED_GLOBAL()`; both are defined in `wolfhsm/wh_client.h`.

5.4.4 Hardware-Only Keys

Some platforms provide key material that the HSM core can read but that never lives in the wolfHSM keystore at all — KEKs burned into OTP or fuses, keys held by an SoC key-management block, or other hardware-provisioned secrets. The optional **hardware keystore front-end** (`WOLFHSM_CFG_HWKEYSTORE`) lets clients reference such keys as KEKs in wrapped-key operations while guaranteeing the material never enters the key cache, never touches NVM, and is never returned to a client.

The front-end (`wolfhsm/wh_hwkeystore.h`) is a platform-agnostic, fully configurable abstraction over the hardware keystore, following the same backend-callback-table paradigm as the lock abstraction and logging modules. A backend provides a `whHwKeystoreCb` callback table whose required `GetKey` callback copies a requested key's bytes into a caller-provided buffer on demand, plus optional `Init` and `Cleanup` callbacks for backend setup/teardown. The callback table, an opaque backend context and config (for backend-specific state), and a lock (serializing callback invocations when the backing hardware is shared across server threads under `WOLFHSM_CFG_THREADSAFE`) are described by a `whHwKeystoreConfig` and held in a `whHwKeystoreContext`. The server application initializes the context once with `wh_HwKeystore_Init()` — which binds the callback table and invokes the backend `Init` callback if present — and binds it to one or more server contexts through the optional `hwKeystore` member of `whServerConfig`, the same ownership pattern as the NVM context.

Clients designate a hardware-only key by setting `WH_KEYID_CLIENT_HW_FLAG` (bit 10) in the request keyId, normally via the `WH_CLIENT_KEYID_MAKE_HW()` macro in `wolfhsm/wh_client.h`. The server translates the flag to the internal key type `WH_KEYTYPE_HW` and routes accordingly:

- In the keywrap operations (key wrap, unwrap-and-export, unwrap-and-cache, and data wrap/unwrap), a hardware-only KEK id causes the server to fetch the KEK from the hardware keystore into a local stack buffer via `wh_HwKeystore_GetKey()`, perform the AES-GCM operation, and zeroize the buffer before returning. The KEK is never cached and never appears in any response. An unwrap-and-cache under a hardware KEK still caches the unwrapped payload — the payload is an ordinary key; only the KEK is hardware-resident.
- Everywhere else — key cache, export (including the public-only and DMA variants), commit, evict, erase, revoke, and every wolfCrypt operation routed through the crypto callback — a hardware-only keyId is rejected with `WH_ERROR_ACCESS`.

Because hardware-only keys carry no `whNvmMetadata`, the usage-flag policy machinery does not apply to them. The largest key the keywrap path will fetch from the backend is bounded by `WOLFHSM_CFG_HWKEYSTORE_MAX_KEY_SIZE` (default 32 bytes, sized for an AES-256 KEK), which should be respected (or enforced), by the port layer.

5.4.4.1 Supported Components Hardware-only keys can be consumed by exactly one wolfHSM component today — the **keywrap API**, where a `WH_CLIENT_KEYID_MAKE_HW()` id names the KEK. The supported operations are:

- **Key wrap** (`wh_Client_KeyWrap`) — wrap a client-supplied key under the hardware-resident KEK.
- **Key unwrap-and-export** (`wh_Client_KeyUnwrapAndExport`) — unwrap a blob and return the recovered key to the client.
- **Key unwrap-and-cache** (`wh_Client_KeyUnwrapAndCache`) — unwrap a blob into the key cache; only the KEK is hardware-resident, while the unwrapped payload becomes an ordinary cached key.
- **Data wrap and unwrap** (`wh_Client_DataWrap`, `wh_Client_DataUnwrap`) — apply the same authenticated encryption to arbitrary application data.

No other component can name a hardware-only key: a direct wolfCrypt operation through the crypto callback, or any keystore lifecycle operation (cache, export, commit, evict, erase, revoke), is rejected with `WH_ERROR_ACCESS`. **Support is limited to keywrap KEK usage for now** — broader use, such as referencing a hardware-only key directly in a crypto operation, may be added in a future release.

5.4.5 Key Usage Policies

A key's whNvmMetadata carries a flags field that the keystore checks on every server-side operation to constrain how the key may be used. The flags fall into two groups: **lifecycle flags** that govern whether a key may be modified, destroyed, exported, or cached at all, and **usage flags** that govern which cryptographic operations it may take part in.

The lifecycle flags are the same whNvmFlags bits described in Object Metadata and Access Attributes and are enforced uniformly across NVM objects: WH_NVM_FLAGS_NONMODIFIABLE, WH_NVM_FLAGS_NONDESTROYABLE, WH_NVM_FLAGS_NONEXPORTABLE, WH_NVM_FLAGS_SENSITIVE, and WH_NVM_FLAGS_EPHEMERAL. For keys specifically, the most consequential of these is NONEXPORTABLE, which prevents wh_Client_KeyExport and the full per-algorithm export helpers from returning the key bytes to the client. The public-only export path is deliberately exempt from this flag because public key material is non-sensitive; the corollary is that a key marked NONEXPORTABLE can still be made useful for client-side verification by exporting only its public half.

The usage flags constrain which cryptographic operations a given key may participate in:

- WH_NVM_FLAGS_USAGE_ENCRYPT: the key may be used to encrypt
- WH_NVM_FLAGS_USAGE_DECRYPT: the key may be used to decrypt
- WH_NVM_FLAGS_USAGE_SIGN: the key may be used to produce signatures or MACs
- WH_NVM_FLAGS_USAGE_VERIFY: the key may be used to verify signatures or MACs
- WH_NVM_FLAGS_USAGE_WRAP: the key may be used as a KEK for wrapped keys or for data wrapping
- WH_NVM_FLAGS_USAGE_DERIVE: the key may be used as input to a key derivation function

Multiple usage flags may be combined, and WH_NVM_FLAGS_USAGE_ANY is a convenience constant equal to the bitwise OR of all USAGE bits. A key whose metadata carries no USAGE bits at all is treated as not permitted for any cryptographic use — attempting to use it returns WH_ERROR_USAGE. This is intentional: a default-zero metadata does not silently grant access; the application must explicitly opt in to each operation a key may perform.

Policy enforcement happens server-side on every relevant operation. Every wolfCrypt request that flows through the crypto callback routes through the keystore's usage check before the underlying primitive runs, and the operation is rejected with WH_ERROR_USAGE if the flag for that operation is not set. Because enforcement is server-side, and because the metadata is bound into the wrapped blob for wrapped keys, no client request can bypass it.

Lifecycle and usage flags are bound to a key at the moment it is first cached or generated and travel with the key into NVM on commit. They cannot be edited in place through the standard client API — once a key has been created with USAGE_SIGN only, the only way to also grant USAGE_VERIFY is to erase the key and recreate it (or, for a public verification key, to extract the public half through the public-only export path and use it as an independent key). This non-editability is what makes the policy useful as a security control: an attacker who compromises a client cannot loosen the policy on an already-provisioned key.

The keystore additionally provides a **revocation** operation (wh_Client_KeyRevoke) that clears all USAGE_* bits and sets WH_NVM_FLAGS_NONMODIFIABLE on a key without destroying the underlying storage. After revocation, every cryptographic use of the key returns WH_ERROR_USAGE, and the key cannot be re-enabled. Revocation is persisted to NVM for committed keys and survives reset; for cache-only keys, eviction has the same effect since the key cannot be reloaded. This makes revocation useful for emergency rotation (taking a compromised signing key out of circulation without immediately reclaiming its NVM slot) and for staged decommissioning, while still leaving the key bytes in place for audit or forensic recovery.

Concrete examples of policy-driven scenarios that this machinery supports:

- A **signing-only** key for code signing: USAGE_SIGN | NONEXPORTABLE | NONMODIFIABLE. The key can produce signatures but never leaves the HSM and cannot be silently replaced.

- A **verification-only** public key: `USAGE_VERIFY`. Attempting to use it for signing returns `WH_ERROR_USAGE` even though the key object is cryptographically capable of either.
- A **KEK** for wrapped-key workflows: `USAGE_WRAP | NONEXPORTABLE`. The key can wrap and unwrap other keys but cannot itself be exported or used for general encryption.
- A **derivation root** for session keys: `USAGE_DERIVE | NONEXPORTABLE | SENSITIVE`. The root is bound to the HSM and is consumed only by KDF operations that produce shorter-lived material.

5.5 Certificate Management

wolfHSM provides a server-resident **certificate manager** that handles the storage of trusted root certificates and the verification of X.509 certificate chains against them. Clients submit a candidate chain and the NVM id of a trusted root — or a list of root ids, with the multi-root API — and the server validates the chain and returns a single yes/no answer, optionally extracting the leaf's public key into the keystore for later crypto operations. The feature is enabled with `WOLFHSM_CFG_CERTIFICATE_MANAGER`. Two opt-ins extend it: `WOLFHSM_CFG_CERTIFICATE_MANAGER_ACERT` adds RFC 5755 attribute certificates, and `WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE` adds the trusted certificate verify cache.

Under the hood, chain verification is delegated to wolfSSL's `WOLFSSL_CERT_MANAGER`, which the server instantiates per request and populates with the requested root(s). This means that the full set of X.509 validation behaviors that wolfSSL implements can be leveraged through wolfHSM.

5.5.1 Trusted Root Storage

Trusted root certificates are stored as ordinary NVM objects (see Non-Volatile Memory). Each root is a DER-encoded X.509 certificate written into NVM under a caller-chosen `whNvmId` with full `whNvmMetadata` — access bits, flags, and label — so that the same access-control machinery that applies to keys also applies to roots.

The lifecycle operations exposed to clients are:

- **Add trusted**: write a root certificate into NVM under a caller-supplied id with the requested metadata. Roots are commonly added with `WH_NVM_FLAGS_NONMODIFIABLE` (and optionally `WH_NVM_FLAGS_NONEXPORTABLE`) so that a compromised client cannot tamper with the trust anchor after provisioning.
- **Erase trusted**: destroy a previously installed root. Subject to the same policy bits as other NVM destroy operations, so a root marked `NONDESTROYABLE` will not be removed.
- **Read trusted**: read a stored root back out to the client. Read access is gated on `WH_NVM_FLAGS_NONEXPORTABLE` if the root was provisioned non-exportable, the read request returns `WH_ERROR_ACCESS` regardless of who issued it.

The maximum size of an individual root is bounded by `WOLFHSM_CFG_MAX_CERT_SIZE`.

5.5.2 Chain Verification

A candidate certificate chain is presented to the server as a single buffer of concatenated DER-encoded X.509 certificates ordered from CA-down-to-leaf, such that each certificate appears in the buffer after the certificate that signed it. The server walks the buffer one ASN.1 SEQUENCE at a time, verifies each certificate against the current trust store, and if the certificate is itself a CA, promotes it into the trust store so the next certificate in the chain can chain to it. The single certificate that is *not* marked as a CA is treated as the leaf, and the verification succeeds only if the leaf chains all the way back to a trusted root that was installed in NVM.

Two verification variants are exposed:

- **Single-root verify** (`wh_Client_CertVerify`): the client gives the NVM id of one trusted root. The chain must anchor to that root.

- **Multi-root verify** (`wh_Client_CertVerifyMultiRoot`): the client gives a list of up to `WOLFHSM_CFG_CERT_MAX_VERIFY_ROOTS` root ids. The chain is accepted if it anchors to *any* of them. Ids not present in NVM are skipped, and the order does not matter.

Multi-root verify lets a client send one fixed list of acceptable roots and run unchanged on any device, no matter which of those roots that device actually has. For example, every device in a fleet can be shipped the same root list while each holds only its own subset. The matching server-side functions are `wh_Server_CertVerify` and `wh_Server_CertVerifyMultiRoot`.

Verification returns a single status code: `WH_ERROR_OK` on a trusted chain, `WH_ERROR_CERT_VERIFY` if no anchor matches, `WH_ERROR_NOTFOUND` if the multi-root call found none of the supplied root ids in NVM, or a more specific error for malformed input.

5.5.2.1 Caching the Leaf Public Key

A common pattern in HSM-mediated workflows is to verify a peer's certificate chain and then use the leaf's public key for subsequent cryptographic operations (signature verification, key exchange, etc.). To support this without round-tripping the public key through the client, every verify call accepts a `WH_CERT_FLAGS_CACHE_LEAF_PUBKEY` flag and an associated keyId:

- If the flag is set and the supplied keyId is `WH_KEYID_ERASED`, the server allocates a fresh unique keyId, extracts the leaf's SubjectPublicKeyInfo into the keystore's "big" cache slot under that id, and returns the keyId to the client.
- If the flag is set and the supplied keyId is a concrete value, the server caches the public key under that id (subject to keystore policy).
- The metadata applied to the cached key is supplied by the caller via the `cachedKeyFlags` argument, so the leaf key inherits an appropriate usage policy (e.g. `WH_NVM_FLAGS_USAGE_VERIFY | WH_NVM_FLAGS_NONEXPORTABLE`) as soon as it is materialized.

The cached key behaves like any other key in the keystore from that point on: it can be used in wolfCrypt operations, committed to NVM by the client, evicted, and so on. The plaintext key bytes never leave the server during this flow — the chain enters and the keyId comes out. Both variants offer a convenience call that sets the flag for you: `wh_Client_CertVerifyAndCacheLeafPubKey` and `wh_Client_CertVerifyMultiRootAndCacheLeafPubKey` (each with a DMA form).

5.5.2.2 DMA Variants

When `WOLFHSM_CFG_DMA` is enabled, parallel DMA variants of all certificate operations let the server read the chain (or write the root, in the case of add/read trusted) directly from client memory rather than copying it through the message buffer. This lifts the per-chain size ceiling above what `WOLFHSM_CFG_COMM_DATA_LEN` would otherwise allow and is the recommended path for verifying long chains or large root certificates. The DMA allowlist applies as it does for any DMA-backed feature — the server will refuse to read a client buffer outside its configured allowed regions. See DMA Support.

5.5.3 Trusted Certificate Verify Cache

Every verify runs a signature check at each link in the chain. When the same CA certificates keep showing up — many clients chaining through one issuing CA, or one client re-checking the same peer over and over — that work repeats even though the result never changes. The optional **trusted certificate verify cache** (`WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE`) remembers CAs that have already verified and skips the signature check when they reappear.

Each cache entry is the SHA-256 hash of a verified CA, tagged with the set of trusted-root ids that were loaded at the time. As the server walks a chain, it hashes each certificate and checks the cache first. A hit skips the signature check; everything else proceeds normally (the CA is still added to the trust store, the leaf key is still extracted if asked). A miss verifies the certificate the usual way, then caches it if it is a CA.

Only CAs are cached, never leaf certificates. Caching a leaf would be unsafe: a later request that sends the leaf on its own could get a cache hit and pass, even though its issuer isn't loaded and the signature check would have failed. CAs are safe to cache because each one is verified as a link in a full chain before it is trusted.

The root-id tag is what makes a cached result safe to reuse. A lookup hits only if the entry's roots are all among the roots the current caller has loaded. This is safe because adding trusted roots can never undo a verify that already passed — so if the original roots are still trusted, the cached result still holds. Single-root verifies make the most reusable entries (one root, easy to match); multi-root verifies make narrower ones that need more roots to match. Both kinds share the same cache.

When a trusted root is added or erased, the server automatically drops every entry that referenced that id. It drops the whole entry rather than just that one id, because the removed root may have been the anchor that made the chain pass. This stops a reused id from producing a stale hit under a root that is no longer there.

The cache is on by default once compiled in, and clients can manage it at runtime:

- **Clear** (`wh_Client_CertVerifyCacheClear`): empty the cache. The next verify re-checks and re-caches.
- **Enable/disable** (`wh_Client_CertVerifyCacheSetEnabled`): turn caching off (which also empties it) or back on. Disable it when you want every verify to run the full signature check.

When full, the cache overwrites the oldest entry first. It holds `WOLFHSM_CFG_CERT_VERIFY_CACHE_COUNT` entries, and each entry tracks up to `WOLFHSM_CFG_CERT_MAX_VERIFY_ROOTS` roots.

By default each client has its own cache. Defining `WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE_GLOBAL` makes one shared cache (with its own lock), so a CA verified by one client can hit for another, and clear/disable then apply to every client. This stays safe: each hit is still checked against the caller's own loaded roots, so no client can borrow another's trust anchors.

> **Note**: A cache hit skips wolfSSL's verify path, so it also skips any verify callback you registered (`whServerCertConfig.verifyCb`, or `wh_Server_CertSetVerifyCb` at runtime). If your callback must run on every chain, leave the cache disabled.

5.5.4 Attribute Certificate (Acert) Support

When built with `WOLFHSM_CFG_CERTIFICATE_MANAGER_ACERT`, wolfHSM also accepts and verifies RFC 5755 **attribute certificates**. An attribute certificate (acert) is a short-lived, separately signed assertion of *attributes* — typically roles, group memberships, or authorization claims — bound to a holder identified by a conventional X.509 identity certificate.

The server-side verification model for acerts is deliberately narrower than for ordinary X.509 chains: an acert is verified directly against the public key of a single trusted root certificate stored in NVM, rather than being walked through a chain. The server reads the trusted root by id, extracts its public key and algorithm, and calls `wc_VerifyX509Acert` to validate the acert's signature. Successful verification means the acert was signed by the holder of the named root, with the validity period, holder binding, and other RFC 5755 fields enforced by wolfCrypt's acert parser; semantic interpretation of the contained attributes is left to the application.

5.6 Communication Layer and Transports

The communication layer is the substrate that carries every client request to the server and every server response back. It sits between the high-level client/server APIs and the platform-specific medium that actually moves bytes between them, and is responsible for everything that has to be true for those APIs to behave as a coherent request/response protocol regardless of where the two sides are physically located. Concretely, it provides a fixed-MTU packet framing with a versioned header, a sequence-numbering scheme that lets the client match each response to its outstanding

request, and a pluggable transport interface that the platform implements once and the rest of the library never has to know about.

The stack is two layers:

- The **comm layer** (`whCommClient` and `whCommServer`, declared in `wolfhsm/wh_comm.h`) provides packet framing, sequence numbering, and the public send/receive functions that the higher-level APIs build on.
- The **transport layer** (`whTransportClientCb` and `whTransportServerCb`) is a small callback table that delivers complete packets between the two endpoints. The comm layer speaks to it through this interface and never touches the underlying medium directly.

Higher-level wolfHSM APIs — the keystore client, the wolfCrypt callback, the NVM client, and every other client subsystem — all build their requests and responses on top of `whCommClient`/`whCommServer` and are agnostic to which transport is in use underneath.

5.6.1 Communication Layer

Each request or response is a single packet composed of an 8-byte `whCommHeader` followed by up to `WOLFHSM_CFG_COMM_DATA_LEN` bytes of payload. The header carries:

- A **magic** field that combines a 1-byte endianness marker and a 1-byte protocol version. The endianness marker lets the receiving side detect a mismatched byte order and use the protocol's translation helpers (`wh_Translate16/32/64`) to byte-swap multi-byte fields as it parses them, so heterogeneous client/server pairings on a single SoC do not need a separate framing layer to agree on byte order.
- A **kind** field that identifies the message: an 8-bit group naming the subsystem (`COMM`, `NVM`, `KEY`, `CRYPTO`, `CERT`, `SHE`, `COUNTER`, `AUTH`, `CUSTOM`, ...) and an 8-bit action within that group. The server uses the group to dispatch the packet to the correct subsystem handler and the action to invoke the specific operation. This is what allows a single transport connection to be multiplexed across every wolfHSM feature.
- A **seq** field, incremented on each client request and copied verbatim onto the matching response. The client validates the sequence number on receipt and rejects mismatched or stale packets, which is what makes the split-transaction client API safe in the face of late or duplicated responses.
- An **aux** field that conveys a session identifier (or `NORESP` for fire-and-forget requests) on the way out and a coarse outcome code (`OK`, `ERROR`, `FATAL`, `UNSUPP`) on the way back, separately from any payload-carried return code.

The comm layer itself is **stateless** in the protocol sense: each request/response transaction stands alone, no session state is required for the server to interpret it, and the only client-side state that persists between calls is the next sequence number and whether a request is currently outstanding. Higher-level features that do require session state (authentication, monotonic counters, etc.) layer it on top of the comm packets rather than embedding it in the protocol itself. The practical consequence is that a single server can serve multiple independent clients over multiple independent transport instances without those clients sharing any state at the comm layer.

The comm layer is also **non-blocking and split-transaction** end-to-end: `wh_CommClient_SendRequest` and `wh_CommClient_RecvResponse` return `WH_ERROR_NOTREADY` rather than waiting on the transport, and the same convention is propagated up through every higher-level client API. Transport errors below this layer surface as either `WH_ERROR_NOTREADY` (retry) or `WH_ERROR_ABORTED` (fatal — clean up and reinitialize the connection), and the higher-level APIs distinguish between the two so applications can implement appropriate retry or recovery logic. See Client/Server Communication for the broader request/response model that this layer implements.

5.6.2 Transport Backends

The transport interface — `whTransportClientCb` on the client side and `whTransportServerCb` on the server side — is a four-function callback table (`Init`, `Send`, `Recv`, `Cleanup`) that delivers complete, MTU-sized packets between client and server. The comm layer speaks only through this interface, so wolfHSM is integrated into a new platform by supplying a transport implementation rather than by modifying the core library. Transports are expected to deliver packets reliably and in order, one at a time and up to `WH_COMM_MTU` bytes; framing, sequencing, dispatch, and policy enforcement are all handled above.

wolfHSM ships with several reference transports that between them cover the most common system topologies:

- **Memory buffer transport** (`wolfhsm/wh_transport_mem.h`): the canonical transport for systems where the client and server share memory — typically a multi-core SoC where the server runs on a secure core and clients run on application cores. Two pre-allocated buffers, one for requests and one for responses, hold the active packet alongside a small control/status header used for flow control. The implementation is small and dependency-free so that ports can use it directly or layer hardware notification (mailbox, interrupt) on top.
- **POSIX TCP transport** (`port/posix/posix_transport_tcp.h`): carries the packet exchange over a length-prefixed TCP stream. The server listens on a configured port and the client connects to it. This is the default transport for the POSIX server example and the recommended development transport, because the client and server can live in separate host processes — or on separate machines — without any additional platform integration.
- **POSIX shared memory transport** (`port/posix/posix_transport_shm.h`): hosts the same request/response buffer layout used by the memory buffer transport in a named POSIX shared memory object, with optional space for DMA-style buffers alongside it. This lets the client and server run as independent processes on the same host while exercising the shared-memory code paths a multi-core SoC would use in production.
- **TLS-over-TCP transport** (`port/posix/posix_transport_tls.h`): wraps the POSIX TCP transport in a wolfSSL-secured channel, with support for certificate-based authentication and PSK. It is intended for deployments where the client and server are physically separated and the link between them cannot be trusted. The packet framing above the TLS session is identical to the plain TCP transport, so higher-level code does not change between the two.
- **ARMv8-M TrustZone NSC bridge transport** (`port/armv8m-tz/wh_transport_nsc.h`): a synchronous transport for ARMv8-M Cortex-M parts where the server runs in the secure world and clients run in the non-secure world. The non-secure client `Send` calls a single `cmse_nonsecure_entry` veneer (`wcs_wolfhsm_transmit`) the integrator provides; that veneer hands the request to the secure-side server inline and returns the response in the same call, which `Recv` then yields. There is no polling or shared-memory ring. The transport is target-agnostic across ARMv8-M parts; the veneer, flash/NVM adapter, and server init are supplied by the secure-side integration. The reference integration is the wolfBoot STM32H5 port.

Beyond the reference transports, platform ports for embedded targets typically supply hardware-specific transports — silicon mailboxes, interrupt-driven inter-core channels, vendor IPC blocks — by implementing the same callback interface. The comm-layer contract is purely "deliver one packet, in order," so a transport need only marshal bytes between the two sides.

Choosing a transport is primarily a function of system topology:

- **Single SoC, server on a secure core**: the memory buffer transport (or a port-supplied hardware mailbox transport built on the same packet model) is the natural choice.
- **Development against a host-side server, or remote production access**: the POSIX TCP transport for trusted links and the TLS-over-TCP transport when confidentiality or authentication on the link is required.
- **Multi-process workloads on a single host**: the POSIX shared memory transport keeps the client

and server in separate address spaces while preserving shared-memory packet semantics, which is useful both for testing port-supplied shared-memory transports and for validating client integrations against an example server without modifying either.

The choice of transport does not affect any other part of the system — the same client code, the same wolfCrypt calls, and the same server initialization sequence work across every transport.

5.7 DMA Support

Every request that travels through the communication layer is bounded by `WOLFHSM_CFG_COMM_DATA_LEN`, and operations that exceed that bound must otherwise be split across multiple round-trips. For workloads that have direct memory-sharing pathways between client and server — most commonly a multi-core SoC where the secure core can address the application core's RAM, or a host environment with shared memory — wolfHSM provides an optional **DMA** mode in which the server reads and writes the client's buffers in place rather than marshaling them through the message buffer. The feature is enabled with `WOLFHSM_CFG_DMA` and is layered onto the existing client/server protocol: a parallel set of DMA-aware request kinds carry pointers and lengths into the client's address space instead of inline data, and the server resolves those pointers under server-enforced policy before touching the underlying memory.

The motivating use cases all involve payloads that are either too large or too inconveniently placed to copy through the comm buffer:

- **Bulk symmetric crypto** over multi-kilobyte messages (AES-CBC/CTR/GCM/ECB, CMAC, streaming SHA-2 updates) where the cost of two copies dominates the cost of the cryptographic primitive.
- **Bulk asymmetric crypto** over large messages or signature buffers (ML-DSA sign/verify, Ed25519 sign/verify) and the associated key import/export paths.
- **Large NVM objects** (`wh_Client_NvmAddObjectDma`, `wh_Client_NvmReadDma`) where the object payload is larger than `WOLFHSM_CFG_COMM_DATA_LEN`.
- **Certificate chain verification** (see DMA Variants) where the chain itself may be several kilobytes and the application already holds it in its own memory.
- **In-place image verification** by the image manager, which is the canonical case: the image being authenticated is already mapped in flash or RAM, and copying it through the comm buffer would defeat the purpose.

5.7.1 DMA Dispatch Mode (`wh_Client_SetDmaMode`)

For wolfCrypt-mediated operations, opt-in to DMA is a per-client dispatch mode rather than a separate device ID: the application calls `wh_Client_SetDmaMode(client, 1)` (or sets `.preferDma` in the client's `whClientDmaConfig` at init), and subsequent wolfCrypt calls made with `WH_CLIENT_DEVID(client)` construct DMA-flavored requests whose payloads carry pointers and lengths into the client's address space rather than inline data. The server-side dispatcher recognizes the DMA request kind and, for each referenced buffer, hands the pointer to the server's DMA address-processing path (described below) before invoking the underlying wolfCrypt primitive. The set of algorithms that have a DMA path mirrors the most performance-sensitive subset of the supported algorithms; with DMA mode preferred, an algorithm without a DMA path automatically falls back to the standard (non-DMA) request, so no call-site changes are needed. The mode can be toggled at any time — `wh_Client_SetDmaMode(client, 0)` returns the client to standard dispatch, and `wh_Client_GetDmaMode()` reads the current setting.

The global `WH_DEV_ID_DMA` device ID is also always registered when `WOLFHSM_CFG_DMA` is enabled and always produces DMA-flavored requests for the most recently initialized client; unlike the DMA dispatch mode, it does not fall back to the standard path for algorithms without a DMA variant. See Transparent Offload via Crypto Callbacks for the global device IDs' single-client-per-process scope.

For the non-crypto subsystems (NVM, certificate manager, image manager, key cache/export, and the data-wrap API) the DMA-aware request kinds are exposed as *Dma variants of the corresponding client API functions — wh_Client_NvmAddObjectDma, wh_Client_KeyCacheDma, wh_Client_KeyExportDma, wh_Client_CertVerifyDma, and so on. See the client API reference for the full set.

5.7.2 Pre-Access and Post-Access Callbacks

A pointer that is valid in the client's address space is not necessarily valid in the server's. On a multi-core SoC the two cores may have distinct memory maps, distinct cache hierarchies that need to be synchronized before and after a shared-buffer access, or both. wolfHSM does not bake any particular assumption about this relationship into the server; instead, it exposes a port-supplied callback that the server invokes around every client-memory access, paired with optional flags to control cache behavior. The callback is registered at server initialization via wh_Server_DmaRegisterCb (or as part of whServerDmaConfig).

The callback signature is:

```
int whServerDmaClientMemCb(whServerContext* server,
                           uintptr_t        clientAddr,
                           void**           serverPtr,
                           size_t           len,
                           whServerDmaOper  oper,
                           whServerDmaFlags flags);
```

It is called twice per access — once before, once after — with the oper argument distinguishing the four phases:

- WH_DMA_OPER_CLIENT_READ_PRE — the server is about to read len bytes from clientAddr. The callback should translate the address into a server-accessible pointer in *serverPtr and, on a system with caches, invalidate or flush the corresponding cache lines so the subsequent read sees up-to-date client memory.
- WH_DMA_OPER_CLIENT_READ_POST — the read has completed. Tear-down hook for any state the PRE callback set up.
- WH_DMA_OPER_CLIENT_WRITE_PRE — the server is about to write len bytes to clientAddr. The callback translates the address and may perform whatever cache preparation the platform requires (flushing dirty lines from the server cache, for example, so the subsequent write is the authoritative copy).
- WH_DMA_OPER_CLIENT_WRITE_POST — the write has completed. On a system where the client must observe the write through its own cache, this is the hook that invalidates the client-visible cache lines.

If no callback is registered, the server uses the client address directly as *serverPtr, which is the right behavior for a system with a flat shared address space and coherent caches. Ports that need either address translation or cache maintenance supply a callback that handles both; the callback is the single extension point for both concerns.

For platforms where the client buffer is not directly memcpy-able even after address translation — for example, when the only path to client memory is through a hardware FIFO or register window — wolfHSM additionally exposes a whServerDmaMemCopyCb callback under WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY. When registered (via wh_Server_DmaRegisterMemCopyCb), this callback replaces the internal memcpy between server and client memory entirely, and is the only operation that touches the client side of the transfer.

The same PRE/POST callback model is also available on the **client side** through wh_Client_DmaRegisterCb, with an identical whClientDmaClientMemCb signature. The client callback is invoked before the request is sent and after the response is received, and is the right place for any work that has

to happen in the client's address space before the server is ever told about a buffer — pinning pages, flushing the client's view of a cache line, or substituting the application's pointer with one that lives in a region the server can actually reach. The POSIX shared-memory transport illustrates the last case: an application buffer allocated from the process's ordinary heap is not visible to the server because it lies outside the mapped shared-memory segment, so the transport's client callback (`posixTransportShm_ClientStaticMemDmaCallback`) detects that the supplied address falls outside the DMA region, allocates a bounce buffer inside the shared segment on `*_READ_PRE`/`*_WRITE_PRE`, copies the application data into it for the read direction, and reports the in-segment offset as the address the server should use. The matching POST phase copies any server-written bytes back to the original application buffer and frees the bounce buffer. From the application's perspective the original wolfCrypt call is unchanged; the client callback transparently bridges the gap between the application's address space and the address space the server can address. Client-side and server-side callbacks are independent — a port may register either, both, or neither, depending on which side needs the translation.

The `whDmaFlags` argument carries per-request hints supplied by the client. Currently the only defined flag is `cacheForceInvalidate`, which the client sets when it has reason to believe the server should not trust any cached view of the buffer (after a DMA write by another agent, for example). Additional flags are reserved for future protocol extensions.

5.7.3 Address Allowlisting

Because DMA gives the server the ability to read or write any address the client passes, an unconstrained DMA path would let a compromised or buggy client direct the server to access memory it has no business touching — kernel memory, other processes' buffers, peripherals mapped into the address space. wolfHSM accordingly enforces, in the server, an **allowlist** of address ranges that DMA requests are permitted to reference. The allowlist is supplied by the port at server initialization (`wh_Server_DmaRegisterAllowList` or `whServerDmaConfig::dmaAddrAllowList`) and is consulted on every PRE phase of every DMA operation.

The allowlist is two parallel tables — one for client reads (server reads from client memory) and one for client writes (server writes to client memory) — so that a region can be made readable without being writable. Each entry is a `{addr, size}` pair, and a DMA request is accepted only if the requested range is **fully contained** within at least one entry in the relevant table. Partial overlap is treated as a failure, and a request that fails the check is rejected with `WH_ERROR_ACCESS` before any access actually occurs. POST phases skip the check on the assumption that the address was already validated at PRE.

When the allowlist is not registered the server allows every address, which is appropriate only for tightly trusted single-application systems and for development. **Production deployments should always supply an allowlist** that covers exactly the shared-memory regions the application is permitted to use; the allowlist is the primary mechanism that bounds the blast radius of a misbehaving client and is the difference between DMA being a performance feature and DMA being an exfiltration channel. The size of the allowlist is bounded at compile time by `WOLFHSM_CFG_DMAADDR_COUNT` entries per direction; a zero-sized entry is treated as unused.

A symmetric `whDmaAddrAllowList` is also available on the client side for clients that wish to validate addresses locally before issuing a DMA request (`whClientDmaConfig::dmaAddrAllowList`). Client-side validation is advisory — the server's check is the authoritative one — but it lets a client fail fast when its own code accidentally constructs an out-of-range pointer.

5.7.4 32-bit vs. 64-bit Address Handling

The DMA path is designed so that a 32-bit client can interoperate with a 64-bit server, and vice versa, without either side losing information. All DMA pointers on the wire are 64-bit (`uint64_t`) regardless of the native pointer size on either endpoint: the client zero-extends its address into the request and the server narrows on receipt if its pointers are 32-bit (with an overflow check that

rejects out-of-range values). The runtime callback signatures use `uintptr_t` so the platform's native pointer width drives address translation. The build distinguishes the two cases via `WH_DMA_IS_32BIT` and `WH_DMA_IS_64BIT`, derived automatically from the system pointer size or set explicitly via `WOLFHSM_CFG_DMA_PTR_SIZE` when the build needs to override that default.

This wire-format choice is what makes asymmetric topologies — a 64-bit secure host serving a 32-bit application core, for example — work without special-case message structures.

5.8 AUTOSAR SHE Subsystem

The AUTOSAR Secure Hardware Extension (SHE) is an automotive industry specification for a small, fixed-function security module embedded alongside an ECU's application core. It defines a set of 128-bit AES key slots, an encrypted key update protocol, a deterministic PRNG, and a CMAC-based secure boot mechanism — together intended to give an ECU the minimum trusted-crypto surface needed for in-vehicle networks. wolfHSM ships an optional SHE implementation, enabled with `WOLFHSM_CFG_SHE_EXTENSION`, that layers a spec-compliant SHE server and client API on top of the existing wolfHSM stack. The intent is that an application written against the SHE command set can use wolfHSM as its SHE module without modification, while still benefiting from the broader wolfHSM keystore, NVM, and transport infrastructure underneath.

The SHE extension is independent of the rest of the crypto API: a server can be built with both SHE and the generic wolfCrypt offload enabled, and clients can mix SHE commands with ordinary wolfCrypt calls over the same connection. SHE keys and native crypto keys share the underlying NVM store but live in disjoint namespaces in the keystore's type field, so the two subsystems cannot collide. The primary use case is automotive ECU firmware that needs to remain conformant with AUTOSAR's SHE expectations — secure boot, MAC-authenticated CAN traffic, encrypted key provisioning — while running on hardware (or a software emulation) that does not provide a dedicated SHE peripheral.

5.8.1 Client API and Command Set

The SHE client API is declared in `wolfhsm/wh_client_she.h` and maps one-to-one onto the AUTOSAR SHE command set. Each spec command is exposed as a `wh_Client_She*` function, with the same `Request` / `Response` split-transaction variants that the rest of the wolfHSM client API uses (see Blocking and Non-Blocking Interfaces). The full set comprises:

- **Secure boot**: `wh_Client_SheSecureBoot` (`CMD_SECURE_BOOT`) — drives the three-phase INIT/UPDATE/FINISH state machine and reports the boot result through the status register
- **Key update**: `wh_Client_SheLoadKey` (`CMD_LOAD_KEY`) — performs the encrypted M1–M5 key update protocol against any slot other than `RAM_KEY`
- **Plain key update**: `wh_Client_SheLoadPlainKey` (`CMD_LOAD_PLAIN_KEY`) and `wh_Client_SheExportRamKe` (`CMD_EXPORT_RAM_KEY`) — load the volatile `RAM_KEY` directly, and export it as an M1–M5 blob bound to the master ECU key for transfer to a peer
- **PRNG**: `wh_Client_SheInitRnd` (`CMD_INIT_RNG`), `wh_Client_SheRnd` (`CMD_RND`), and `wh_Client_SheExtendSeed` (`CMD_EXTEND_SEED`) — initialize, draw from, and reseed the spec's deterministic PRNG
- **Bulk crypto**: `wh_Client_SheEncEcb` / `wh_Client_SheEncCbc` / `wh_Client_SheDecEcb` / `wh_Client_SheDecCbc` (`CMD_ENC_*` / `CMD_DEC_*`) — AES-ECB and AES-CBC encrypt and decrypt against a selected key slot
- **MAC**: `wh_Client_SheGenerateMac` / `wh_Client_SheVerifyMac` (`CMD_GENERATE_MAC` / `CMD_VERIFY_MAC`) — CMAC generation and verification against a selected key slot
- **Status**: `wh_Client_SheGetStatus` (`CMD_GET_STATUS`) — reads the SHE status register (SREG)

In addition to the spec commands, wolfHSM exposes two non-standard helpers that fill gaps left by the spec's assumption of dedicated hardware:

- wh_Client_SheSetUid: explicitly programs the 15-byte ECU UID that the key update protocol binds against. The AUTOSAR spec assumes this value is hardware-fused; wolfHSM needs a software path to install it, and rejects most SHE operations until it has been set.
- wh_Client_ShePreProgramKey: writes a key directly into a SHE NVM slot, bypassing the encrypted M1–M5 protocol. This exists to support initial provisioning on a blank device — once a MASTER_ECU_KEY exists, all subsequent updates can go through the spec-compliant protocol.

All SHE commands return one of the spec's WH_SHE_ERC_* error codes (SEQUENCE_ERROR, KEY_NOT_AVAILABLE, WRITE_PROTECTED, KEY_UPDATE_ERROR, etc.) alongside the wolfHSM transport return code, so applications can distinguish protocol-level failures from communication failures.

5.8.2 SHE Key Slots and the wolfHSM Keystore

The AUTOSAR SHE specification defines sixteen 128-bit AES key slots, identified by IDs 0 through 15, with fixed roles for several of them:

- **SECRET_KEY** (ID 0): the master secret consumed by the PRNG derivation
- **MASTER_ECU_KEY** (ID 1): the ECU's identity key, used as the authorization key for key updates
- **BOOT_MAC_KEY** (ID 2): the key used to compute the bootloader CMAC during secure boot
- **BOOT_MAC** (ID 3): the expected CMAC digest of the bootloader, compared against during secure boot
- **KEY_4...KEY_13** (IDs 4–13): ten general-purpose user key slots
- **RAM_KEY** (ID 14): a volatile slot that lives only in the cache and is lost on power cycle
- **PRNG_SEED** (ID 15): the persistent PRNG seed state

wolfHSM does not implement these slots as a parallel storage layer; they are stored as ordinary objects in the NVM and keystore, with the SHE-specific roles encoded in the keyId. Each SHE key is given a whKeyId constructed with the WH_KEYTYPE_SHE type field, the connection's client ID in the USER field, and the SHE slot number in the ID field. The WH_KEYTYPE_SHE type means SHE objects are distinct from WH_KEYTYPE_CRYPTO keys even when they share the same numeric slot number, and the USER field gives every connected client its own independent set of sixteen SHE slots — a property that follows directly from the keystore's per-client isolation model and that the SHE spec itself does not require but that wolfHSM provides for free.

The SHE spec also requires every key to carry a 28-bit monotonic update counter and a 5-bit set of protection flags (WRITE_PROTECT, BOOT_PROTECT, DEBUGGER_PROTECTION, USAGE, WILDCARD). wolfHSM stores these by repurposing the first eight bytes of the NVM label field as a whSheMetadata record holding the counter and flags in big-endian order; conversion is done by wh_She_Meta2Label / wh_She_Label2Meta. This means SHE keys need no additional NVM machinery beyond what the generic object store already provides: the counter and protection flags survive resets exactly like the key payload itself, and the SHE-side update logic — counter strictly-increasing checks, WRITE_PROTECT enforcement, and the rest — is implemented on top of the existing metadata round-trip.

RAM_KEY is the one exception to NVM-backed storage. The spec defines it as volatile, so the server caches the loaded key in its key cache but never calls into the NVM layer for it; eviction or reset clears it. All other slots, including PRNG_SEED, persist.

5.8.3 Encrypted Key Update Protocol (M1–M5)

The most intricate piece of the SHE spec is the encrypted key update protocol that runs underneath CMD_LOAD_KEY. wolfHSM implements it as specified: the client constructs three input messages (M1, M2, M3) by encrypting and CMACing the new key and its metadata under keys derived from a chosen authorization key, sends them to the server, and receives two response messages (M4, M5) that prove the server stored the new key correctly.

The protocol uses a key derivation function based on the **Miyaguchi-Preneel** one-way compression construction (`wh_She_AesMp16` in `wh_she_crypto.c`), with the exact derivation constants — `KEY_UPDATE_ENC_C`, `KEY_UPDATE_MAC_C`, `PRNG_KEY_C`, `PRNG_SEED_KEY_C` — that the AUTOSAR specification mandates. From the authorization key the server derives an encryption key (K1) and a MAC key (K2) for verifying the request, and from the *new* key it derives a separate pair (K3, K4) for proving storage in the response.

The server-side handler enforces all of the spec's update constraints in addition to verifying M3 and decrypting M2: the new counter must be strictly greater than the previous counter for that slot (rollback protection), the existing `WRITE_PROTECT` flag must not be set, the UID in M1 must match the ECU's configured UID (unless the existing key has `WILDCARD` set and M1 carries the all-zero UID), and the authorization key referenced by the AID field in M1 must exist. Only after all of these pass is the new key written into NVM and the response constructed.

5.8.4 Secure Boot

SHE secure boot is implemented as a three-phase state machine that the client drives via `CMD_SECURE_BOOT`:

1. **INIT**: the client supplies the total bootloader length; the server reads `BOOT_MAC_KEY` from NVM, initializes a CMAC with a 12-byte zero prefix followed by the length, and transitions to the UPDATE state.
2. **UPDATE**: the client streams the bootloader into the CMAC in arbitrary-sized chunks; the server feeds each chunk into the running CMAC and stays in UPDATE until the cumulative length matches the value declared in INIT.
3. **FINISH**: the server finalizes the CMAC and compares it byte-for-byte against the stored `BOOT_MAC` (slot ID 3). A match sets `WH_SHE_SREG_BOOT_OK` in the status register; a mismatch leaves `BOOT_OK` clear. Either outcome sets `BOOT_FINISHED` and transitions the state machine to a terminal state.

While the state machine is in any state other than SUCCESS, the SHE handler refuses every non-boot command except `CMD_GET_STATUS` and `CMD_SET_UID`, returning `WH_SHE_ERC_SEQUENCE_ERROR`. This is what allows the SHE module to gate cryptographic services on a successful boot measurement: once boot has succeeded, the rest of the SHE command set unlocks; on a boot failure the keys remain inaccessible and only status queries are honored.

The bootloader bytes are supplied through the standard message buffer in chunks of up to `WOLFHSM_CFG_COMM_DATA_LEN`. For large bootloaders this is the natural place to opt into DMA — a future variant of the secure boot handler could read the bootloader image directly out of flash using the DMA address-translation path — but the current implementation is purely buffer-based.

5.8.5 Deterministic PRNG

The SHE PRNG is deterministic, seeded from the master `SECRET_KEY` and a persisted `PRNG_SEED`, and is meant to be used both for spec-defined operations (the M5 verification path, internal nonce generation) and as a standards-compliant entropy source for the application:

- `CMD_INIT_RNG`: the server reads `SECRET_KEY` and `PRNG_SEED` from NVM, derives `PRNG_KEY` and a new state via the spec's Miyaguchi-Preneel construction, advances the seed, and writes the new seed back to NVM. After this completes `WH_SHE_SREG_RND_INIT` is set in the status register.
- `CMD_RND`: each invocation runs an AES-CBC encryption of the current PRNG state under `PRNG_KEY` to produce 16 bytes of output and advance the state.
- `CMD_EXTEND_SEED`: mixes 16 bytes of caller-supplied entropy into the state via the same Miyaguchi-Preneel compression and writes the updated seed back to NVM, so reseeding survives reboots.

The PRNG's `state` and derived `prngKey` live in the per-connection `whServerSheContext`, while the persisted `PRNG_SEED` (slot ID 15) lives in NVM exactly like any other SHE key. This is the only piece of SHE that maintains live cryptographic state in the server context rather than in the keystore.

5.8.6 Status Register (SREG)

The SHE status register is an 8-bit field that reports the module's current secure-boot and PRNG state. The wolfHSM implementation maps the spec's bits as follows:

- `SECURE_BOOT` (bit 1): set if a `BOOT_MAC_KEY` has been provisioned for the connected client
- `BOOT_FINISHED` (bit 3): set after secure boot has completed, regardless of outcome
- `BOOT_OK` (bit 4): set only if secure boot succeeded
- `RND_INIT` (bit 5): set after `CMD_INIT_RNG` has succeeded for the current session

The `BUSY` (bit 0), `BOOT_INIT` (bit 2), `EXT_DEBUGGER` (bit 6), and `INT_DEBUGGER` (bit 7) positions are reserved and not currently driven by the implementation; the corresponding spec-defined behaviors (debugger-presence interlocks, asynchronous busy reporting) are intentionally out of scope for the software implementation and would be supplied by a hardware-backed port if needed.

5.8.7 Integration with the Rest of wolfHSM

The SHE extension is built on top of the same infrastructure as every other wolfHSM feature, with a few specific touchpoints worth calling out:

- **NVM**: SHE keys are ordinary NVM objects under the `WH_KEYTYPE_SHE` namespace; they inherit fail-safe atomicity, partition compaction, and the rest of the NVM guarantees. The 24-byte `label` field carries SHE-specific counter and flag metadata.
- **Keystore**: SHE keys live in the same per-client `whKeyId` space as crypto keys, with the TYPE field disambiguating the two. SHE keys do not currently consume the per-key usage flag policy machinery — usage constraints are expressed through the SHE-spec flag set in the label instead — but lifecycle flags like `NONMODIFIABLE` apply at the NVM layer just as they do for any other object.
- **Communication layer**: every SHE command is a packet under the `WH_MESSAGE_GROUP_SHE` group and is dispatched through the comm layer like any other request. SHE clients work over every available transport without modification.
- **Global keys** and **wrapped keys**: not currently supported for SHE keys — the SHE keyId namespace uses the per-client USER field and does not interpret `WH_KEYUSER_GLOBAL` or the wrapped flag. Applications that need to share a key across clients must do so by provisioning it into each client's SHE namespace separately.

A typical automotive deployment uses the SHE extension end-to-end: the bootloader and `BOOT_MAC` are programmed into NVM at production using `wh_Client_ShePreProgramKey`, the device's UID is set on first boot with `wh_Client_SheSetUid`, secure boot is run on every reset via `wh_Client_SheSecureBoot`, in-field key updates flow through the encrypted `CMD_LOAD_KEY` protocol, and CAN message authentication uses `wh_Client_SheGenerateMac` / `wh_Client_SheVerifyMac` against pre-provisioned user-slot keys.

5.9 Non-Volatile Monotonic Counters

wolfHSM provides **non-volatile monotonic counters**: server-resident 32-bit values that are guaranteed never to decrease across resets and power cycles. They are the building block for anti-rollback checks on firmware versions, replay protection, audit tallies, and unique-per-boot nonces — anywhere an application needs a persistent, strictly-increasing value that a client cannot rewind.

5.9.1 Counter Semantics

A counter is a 32-bit unsigned value supporting four operations:

- **Init**: create the counter with a caller-supplied starting value, or overwrite an existing counter. **Reset** is the same call with value zero. This is the only path that can lower a counter, intended for provisioning rather than runtime use.
- **Increment**: atomically read, add one, and write back. Returns the new value.
- **Read**: return the current value without modifying it.
- **Destroy**: remove the counter from NVM. Subsequent reads and increments return `WH_ERROR_NOTFOUND` until it is re-initialized.

Increment **saturates at `UINT32_MAX`** rather than rolling over: once the counter reaches the maximum it stays there, and no NVM write is performed. Silent rollover would defeat the monotonicity guarantee, so the subsystem refuses to wrap even at the cost of losing further increments. Applications approaching saturation should treat the counter as exhausted and rotate to a new identifier.

Every mutating operation is committed by the NVM layer before the response is returned, so a power loss leaves the counter at either its pre- or post-increment value but never in between. The implementation uses `wh_Nvm_AddObjectWithReclaim`, so the partition is compacted in place as needed and a frequently-incremented counter does not accumulate dead entries.

5.9.2 Counter Identifiers and Storage

A counter is referenced by a 16-bit `whNvmId` supplied by the caller, with `WH_KEYID_ERASED` (0) reserved as invalid. Internally the server encodes it as a `whKeyId` with TYPE = `WH_KEYTYPE_COUNTER`, USER = the connection's client id, and ID = the supplied value. This means counters inherit the keystore's per-client isolation — each client has its own counter namespace — and that counter id 5 and key id 5 are distinct objects in the same NVM store.

The 32-bit value is stored in the **`label` field of the object's `whNvmMetadata`** with a zero-length payload. A counter therefore lives entirely in the metadata that the NVM layer already reads on every directory operation, so an increment is a single metadata write and a read is satisfied by `wh_Nvm_GetMetadata` alone. The remainder of the label and the access/flags fields are unused by the counter subsystem. Counters share the `WOLFHSM_CFG_NVM_OBJECT_COUNT` object budget with keys and other NVM objects.

5.9.3 Client API

The operations are exposed in `wolfhsm/wh_client.h`:

- `wh_Client_CounterInit(ctx, counterId, &value)` — create or overwrite with the supplied initial value
- `wh_Client_CounterReset(ctx, counterId, &value)` — initialize to zero
- `wh_Client_CounterIncrement(ctx, counterId, &value)` — atomically increment, return the new value
- `wh_Client_CounterRead(ctx, counterId, &value)` — read without modifying
- `wh_Client_CounterDestroy(ctx, counterId)` — remove from NVM

Each function has split-transaction `Request` / `Response` counterparts for non-blocking use. Requests are dispatched under `WH_MESSAGE_GROUP_COUNTER` through the standard communication layer.

5.10 Image Manager

The image manager is a server-side facility for authenticating an arbitrary region of memory — typically a firmware image, but equally a data blob, a configuration record, or any other contiguous payload — against a cryptographic signature or MAC using a key resident in the server. It is enabled

with WOLFHSM_CFG_SERVER_IMG_MGR and is the canonical mechanism by which a wolfHSM-equipped system can drive HSM-mediated secure boot of an application core, gate execution of a dynamically loaded image, or perform a periodic runtime integrity check of a code or data region without ever exposing the verifying key or the signature to the client.

The verification model is straightforward: each managed image is described by a pointer and length into the memory the server can address, plus the keyId of the verification key and either an NVM id for the signature or an indication that the signature is embedded in the image itself. At verification time, the server reads the image (in place, via DMA, when available), loads the key from the keystore, retrieves the signature, runs a verify method against the configured algorithm, and invokes an application-supplied action callback with the result.

5.10.1 Image Configuration

Images are registered at server initialization through a whServerImgMgrConfig that points to an array of whServerImgMgrImg records. The maximum number of managed images is bounded at compile time by WOLFHSM_CFG_SERVER_IMG_MGR_MAX_IMG_COUNT. Each whServerImgMgrImg carries:

- addr / size: the location and size of the image payload in server-addressable memory
- hdrAddr / hdrSize: for image formats whose signature is embedded in the image itself (such as wolfBoot), the location of the header from which the signature and ancillary metadata are extracted
- keyId: the keyId of the verification key
- sigNvmId: for image types whose signature lives in NVM, the whNvmId of the signature object; for cert-chain image types, the whNvmId of the trusted root certificate
- imgType: one of WH_IMG_MGR_IMG_TYPE_RAW, WH_IMG_MGR_IMG_TYPE_WOLFBOOT, or WH_IMG_MGR_IMG_TYPE_WOLFBOOT_CERT, which tells the framework how to load the key and signature
- verifyMethod: the verify callback that runs the actual cryptographic check
- verifyAction: the post-verification callback invoked with the verify result

Once registered, an image can be verified individually by reference (wh_Server_ImgMgrVerifyImg), by index into the registered array (wh_Server_ImgMgrVerifyImgIdx), or in bulk against every registered image (wh_Server_ImgMgrVerifyAll). All three calls return both the cryptographic outcome and the action callback's return value through a whServerImgMgrVerifyResult so the caller can distinguish a verification failure from an action failure.

5.10.2 Verify Methods

A *verify method* is the callback that performs the cryptographic check against a (image, key, signature) triple. The signature is:

```
int verifyMethod(whServerImgMgrContext* ctx,
                 const whServerImgMgrImg* img,
                 const uint8_t* key, size_t keySz,
                 const uint8_t* sig, size_t sigSz);
```

A verify method returns WH_ERROR_OK on a successful verification, WH_ERROR_NOTVERIFIED when the signature does not match, or a negative error code for an operational failure (DMA error, malformed input, missing key). wolfHSM ships with several built-in verify methods:

- wh_Server_ImgMgrVerifyMethodEccWithSha256: ECDSA P-256 signature over the SHA-256 hash of the image
- wh_Server_ImgMgrVerifyMethodRsaSslWithSha256: RSA PKCS#1 v1.5 signature over the SHA-256 hash of the image
- wh_Server_ImgMgrVerifyMethodAesCmac: AES-128 CMAC over the image bytes

- `wh_Server_ImgMgrVerifyMethodWolfBootRsa4096WithSha256`: RSA-4096 verification of a wolfBoot-formatted image (see wolfBoot Image Support)
- `wh_Server_ImgMgrVerifyMethodWolfBootCertChainRsa4096WithSha256`: cert-chain-based RSA-4096 verification of a wolfBoot image

Applications can supply their own verify method to support algorithms not represented in the built-in set, or to layer additional checks on top of an existing one — for example, validating a monotonic counter against a non-volatile counter inside a wrapper verify method to add anti-rollback protection. The maximum signature size handled by the framework is `WOLFHSM_CFG_SERVER_IMG_MGR_MAX_SIG_SIZE`, whose default accommodates RSA-4096.

5.10.3 Verify Actions

The *verify action* is the application-supplied callback invoked after the verify method completes, with the result of verification as an argument. Its signature is:

```
int verifyAction(whServerImgMgrContext* ctx,
                 const whServerImgMgrImg* img,
                 int verifyResult);
```

The action callback is the extension point through which the image manager produces an externally visible effect. Typical actions include releasing reset on the application core whose image was just verified, jumping to the verified image's entry point, latching a hardware "verified" signal, or simply logging the outcome. Because the action receives the verify result, it is the right place to implement both success and failure handling: a boot-time action might release reset on success and assert a fault pin on failure, while a periodic-integrity action might do nothing on success and force a reset on failure.

A default no-op action, `wh_Server_ImgMgrVerifyActionDefault`, is provided for cases where the caller wants the verification result back but does not need any side effect.

5.10.4 wolfBoot Image Support

wolfHSM understands the wolfBoot image header format natively so that a wolfHSM-equipped system can serve as the verifier for a wolfBoot-staged image without the client having to parse the header itself. Two wolfBoot image types are recognized:

- `WH_IMG_MGR_IMG_TYPE_WOLFBOOT`: the signature is extracted from the wolfBoot TLV header and verified against a key resident in the server's keystore (identified by `keyId`). This corresponds to the standard wolfBoot signing model where the signing key is known in advance and provisioned into the HSM.
- `WH_IMG_MGR_IMG_TYPE_WOLFBOOT_CERT`: the image carries a certificate chain inside its wolfBoot header; the chain is verified against a trusted root in NVM (identified by `sigNvmId`) using the certificate manager, and the leaf certificate's public key is then used to verify the image signature. This matches the wolfBoot cert-chain mode and is the right choice when the signing key is rotated independently of the on-device trust anchor.

In both cases the framework parses the header at `hdrAddr`, locates the signature TLV, validates the wolfBoot magic and public key hint, and feeds the appropriate `(image, key, signature)` triple into the wolfBoot verify method. The application's `verifyAction` is invoked exactly as for a raw image.

5.10.5 In-Place Access via DMA

The image manager is fully DMA-aware. When `WOLFHSM_CFG_DMA` is enabled, the server reads the image header and payload directly from the address the caller registered, rather than having to copy the image through the comm message buffer. This is essential in practice because firmware images

are typically megabytes in size, are already mapped into flash or RAM by the client, and would be impractical to ship over the comm protocol.

The DMA path runs the standard pre- and post-access callbacks around every read of the image, so any cache maintenance and address translation the platform requires is handled by the same port-supplied hook that DMA crypto uses. The DMA address allowlist is consulted on every access, so the image manager cannot be coerced into reading from a region the port has not explicitly permitted.

The image manager is not gated on `WOLFHSM_CFG_DMA`, and the same verification flow works in builds without DMA. In that configuration the server dereferences the registered image and header addresses directly instead of routing them through the pre- and post-access callbacks, so no address translation, cache maintenance, or allowlist check is performed. This mode is appropriate when the image already lives in memory the server can address natively — for example, an image resident in the server's own flash, or a single-address-space build where client and server share a memory map — and the caller is responsible for ensuring that `img->addr` and `img->hdrAddr` (and the lengths derived from them) refer to memory the server is permitted to read. Systems with a real client/server address-space boundary should enable DMA so the allowlist and translation callbacks apply.

5.11 Custom Callbacks

wolfHSM's built-in feature set covers the common HSM workload, but it cannot anticipate every application-specific service that an embedded system may need to run in its secure environment. The **custom callback** feature is wolfHSM's extension point for these cases: an application registers one or more callback functions in the server's dispatch table, and clients invoke them by ID through the same request/response pipeline that carries every other wolfHSM operation. The wire framing, dispatch, and byte-order translation are handled by the library; the contents of the request and response payload are defined entirely by the application.

The mechanism is intended for operations that naturally belong on the secure side of the trust boundary but are not part of the standard HSM API. Typical uses include proprietary key derivation routines that consume on-device material, application-specific authentication or monitoring protocols that need to run alongside the standard crypto offload, and anything else specific to hardware or proprietary application functionality. Because custom callbacks reuse the standard communication layer, the same callback works over every supported transport, and the server's normal polling loop dispatches custom requests interchangeably with built-in ones.

5.11.1 Server-Side Registration and Dispatch

A custom callback is a function of type `whServerCustomCb`, defined in `wolfhsm/wh_server.h`:

```
typedef int (*whServerCustomCb)(
    whServerContext* server,
    const whMessageCustomCb_Request* req,
    whMessageCustomCb_Response*      resp
);
```

The callback receives the dispatching server context, a translated request structure populated from the client's message, and an output response structure that wolfHSM will marshal back to the client when the callback returns. The callback populates `resp->data` with any output payload and may set `resp->err` to a wolfHSM error code; the callback's own return value is reported to the client as `resp->rc`. The library does not interpret either field — they are simply propagated to the client unchanged, with the convention that `rc` is invalid whenever `err` is non-zero.

Callbacks are registered against a 16-bit **action ID** in the server's dispatch table:

```
int wh_Server_RegisterCustomCb(whServerContext* server, uint16_t action,
                               whServerCustomCb handler);
```

The table is statically sized at build time by WOLFHSM_CFG_SERVER_CUSTOMCB_COUNT, and the action ID is the index into it. Action IDs are application-defined: they have no meaning to wolfHSM beyond identifying which slot in the table to dispatch to. Registration may occur at any point in the server's lifetime — there is no requirement to install all callbacks before the server starts handling requests — so a server application can register or replace handlers dynamically based on its own state. A request that arrives for an action ID with no registered handler is reported back to the client with resp->err == WH_ERROR_NOHANDLER.

5.11.2 Client-Side Invocation

A client invokes a registered callback by populating a whMessageCustomCb_Request and sending it through the standard split-transaction client API:

- wh_Client_CustomCbRequest() dispatches the request to the server
- wh_Client_CustomCbResponse() polls for the matching response

The two functions follow the same non-blocking pattern as the rest of the client API. Because dispatch is by action ID, the client must agree with the server on the meaning of each ID — application code typically defines a shared header that names every action ID and the data layout for each.

To allow a client to query the server before invoking a callback whose presence it is unsure of, wh_Client_CustomCbCheckRegistered() returns WH_ERROR_OK if the supplied action ID has a handler installed and WH_ERROR_NOHANDLER if it does not.

5.11.3 Request and Response Messages

The request and response messages are declared in wolfhsm/wh_message_customcb.h:

```
typedef struct {
    uint32_t               id;
    uint32_t               type;
    whMessageCustomCb_Data data;
} whMessageCustomCb_Request;

typedef struct {
    uint32_t id;
    uint32_t type;
    int32_t  rc;
    int32_t  err;
    whMessageCustomCb_Data data;
} whMessageCustomCb_Response;
```

The id field carries the action ID, echoed back unchanged on the response for client-side bookkeeping. The type field is a hint to the callback describing how to interpret data. The response additionally carries rc (the callback's own return value) and err (a wolfHSM-defined error code populated by the framework or by the callback).

The data field is a union that exposes three pre-defined shapes plus a raw buffer for application-specific schemas:

```
typedef union {
    struct { uint32_t client_addr, client_sz, server_addr, server_sz; } dma32;
    struct { uint64_t client_addr, client_sz, server_addr, server_sz; } dma64;
    struct { uint8_t data[WOLFHSM_CFG_CUSTOMCB_LEN]; } buffer;
} whMessageCustomCb_Data;
```

The dma32 and dma64 variants carry pointer-and-length pairs for systems where the client wants the server to operate directly on its memory; the buffer variant is a fixed-size opaque payload, sized by

WOLFHSM_CFG_CUSTOMCB_LEN, into which the application encodes whatever structure it likes. Custom callbacks do **not** automatically run the server's DMA address-translation or allowlist machinery for the dma* shapes — they are simply a convention for passing addresses, and the callback is responsible for any address handling or policy enforcement it needs.

The type field is an enum (whMessageCustomCb_Type) whose first eight values are reserved by wolfHSM (WH_MESSAGE_CUSTOM_CB_TYPE_DMA32, WH_MESSAGE_CUSTOM_CB_TYPE_DMA64, the internal WH_MESSAGE_CUSTOM_CB_TYPE_QUERY, and several reserved slots), with application-defined types beginning at WH_MESSAGE_CUSTOM_CB_TYPE_USER_DEFINED_START (value 8). The framework recognizes the reserved types for byte-order translation of the corresponding data union variant, but **user-defined types are passed through unmodified, so a callback that needs to interoperate between endpoints of different endianness must perform its own translation**.

5.11.4 Constraints

- The maximum number of registered callbacks is fixed at build time by WOLFHSM_CFG_SERVER_CUSTOMCB_COUNT. Valid action IDs are in the range [0, WOLFHSM_CFG_SERVER_CUSTOMCB_COUNT).
- The data payload in either direction is bounded by WOLFHSM_CFG_CUSTOMCB_LEN. Operations whose payload exceeds this limit must either split across multiple requests or use the dma32 / dma64 shapes to point at a larger buffer in shared memory.
- The callback's return value is reported to the client through resp->rc; wolfHSM error semantics are conveyed by the library separately through resp->err. Callbacks should reserve err for genuine wolfHSM-defined failures (the WH_ERROR_* set) and use rc for application-defined results.

5.11.5 Example

This example registers a single callback under action ID 0 that handles three kinds of request: a DMA32 payload describing a buffer in client memory, and two application-defined types each carrying a different struct in the buffer variant. A shared header defines the action ID, the user-defined type values, and the per-type payload structs:

```
/* my_custom_cb.h - shared between client and server */
#include "wolfhsm/wh_message_customcb.h"

#define MY_CUSTOM_CB_ID 0

enum {
    MY_TYPE_A = WH_MESSAGE_CUSTOM_CB_TYPE_USER_DEFINED_START,
    MY_TYPE_B,
};

typedef struct { int foo; int bar; } myCustomCbDataA;
typedef struct { int noo; int baz; } myCustomCbDataB;
```

The server registers the callback and then enters its standard request-handling loop:

```
#include "wolfhsm/wh_server.h"
#include "my_custom_cb.h"

static int myCustomCb(whServerContext*                 server,
                      const whMessageCustomCb_Request* req,
                      whMessageCustomCb_Response*      resp)
{
    int rc = 0;
    resp->err = WH_ERROR_OK;
```

```
    switch (req->type) {
        case WH_MESSAGE_CUSTOM_CB_TYPE_DMA32: {
            uint8_t* ptr = (uint8_t*)((uintptr_t)req->data.dma32.client_addr);
            rc = doWorkOnClientAddr(ptr, req->data.dma32.client_sz);
            break;
        }
        case MY_TYPE_A:
            rc = doWorkWithTypeA((myCustomCbDataA*)req->data.buffer.data);
            break;
        case MY_TYPE_B:
            rc = doWorkWithTypeB((myCustomCbDataB*)req->data.buffer.data);
            break;
        default:
            resp->err = WH_ERROR_BADARGS;
            break;
    }
    return rc;
}

int main(void) {
    whServerContext serverCtx;
    whServerConfig  serverCfg = { /* server config */ };

    wh_Server_Init(&serverCtx, &serverCfg);
    wh_Server_RegisterCustomCb(&serverCtx, MY_CUSTOM_CB_ID, myCustomCb);

    while (1) {
        wh_Server_HandleRequestMessage(&serverCtx);
    }
}
```

The client verifies the callback is registered, then issues requests against the supported types:

```
#include "wolfhsm/wh_client.h"
#include "my_custom_cb.h"

int main(void) {
    whClientContext clientCtx;
    whClientConfig  clientCfg = { /* client config */ };
    int err = 0;

    wh_Client_Init(&clientCtx, &clientCfg);

    if (wh_Client_CustomCbCheckRegistered(&clientCtx, MY_CUSTOM_CB_ID, &err)
            != WH_ERROR_OK) {
        return -1; /* callback not registered on this server */
    }

    whMessageCustomCb_Request  req  = {0};
    whMessageCustomCb_Response resp = {0};

    /* DMA-style invocation: hand the server a pointer in our address space */
    uint8_t buf[LARGE_SIZE] = { /* ... */ };
    req.id   = MY_CUSTOM_CB_ID;
```

```
    req.type = WH_MESSAGE_CUSTOM_CB_TYPE_DMA32;
    req.data.dma32.client_addr = (uint32_t)((uintptr_t)buf);
    req.data.dma32.client_sz   = sizeof(buf);
    wh_Client_CustomCbRequest(&clientCtx, &req);
    wh_Client_CustomCbResponse(&clientCtx, &resp);

    /* Application-defined invocation: pass a struct in the buffer payload */
    myCustomCbDataA a = { /* ... */ };
    memset(&req, 0, sizeof(req));
    req.id   = MY_CUSTOM_CB_ID;
    req.type = MY_TYPE_A;
    memcpy(req.data.buffer.data, &a, sizeof(a));
    wh_Client_CustomCbRequest(&clientCtx, &req);
    wh_Client_CustomCbResponse(&clientCtx, &resp);
}
```

The same pattern scales naturally: each action ID can dispatch to a different callback, and a single callback can multiplex any number of `type` values to handle distinct sub-operations behind one ID.

5.12 Concurrency Support

Many systems that integrate an HSM have multiple threads, cores, or subsystems issuing cryptographic operations at the same time. wolfHSM is designed to support these workloads while keeping the request/response protocol simple and predictable. Concurrency is achieved by the server processing requests from multiple **independent client sessions in parallel**: each session still handles requests sequentially, but the sessions themselves can be scheduled concurrently. wolfHSM v1.4.0 introduced the locking infrastructure that makes this safe; everything described in this section applies from that version onward.

The library itself never spawns threads or assumes a particular runtime — the embedding application controls scheduling. wolfHSM provides the building blocks (per-session context structures, an internal lock abstraction, a transport interface that does not constrain dispatch) and lets the application choose between a simple round-robin loop, an event-driven dispatcher, or one thread per session.

5.12.1 Per-Context Threading Model

The unit of concurrency on both sides of the connection is the context:

- A `whClientContext` represents a single client session with the server. It must be used by **one thread at a time**: there is at most one in-flight request per session, and the matching response must be received before the next request can be issued. Applications that want to issue requests in parallel create multiple client contexts — typically one per worker thread — each connected to its own server session.
- A `whServerContext` represents the server side of a single client session. It is likewise **single-threaded from the caller's perspective**: the application must not call `wh_Server_HandleRequestMessage()` for the same context from two threads at once. Concurrency across sessions is achieved by creating multiple server contexts and dispatching them independently.

Shared server-side state — the NVM object store, the global key cache, the authentication state, and anything else that lives outside a single session — is protected by the internal lock infrastructure described below, so multiple server contexts can safely operate against the same backing store at the same time.

5.12.2 The Lock Abstraction

To keep the core library free of OS dependencies, wolfHSM serializes access to shared resources through a generic lock abstraction declared in `wolfhsm/wh_lock.h`. Each shared resource embeds its own `whLock` instance, and the platform port supplies a callback table (`whLockCb`) that implements the four lifecycle operations against the native synchronization primitive of choice:

- `init` / `cleanup` — set up and tear down the platform-specific lock state
- `acquire` / `release` — blocking exclusive lock and unlock

Reference implementations cover the common cases:

- **POSIX pthread mutexes** (`port/posix/posix_lock.h`), used by the POSIX server example and the threadsafe stress tests
- Ports for embedded RTOS targets are expected to register a callback table backed by FreeRTOS mutexes, atomic spinlocks, or any other primitive the platform already provides

The feature is gated by `WOLFHSM_CFG_THREADSAFE`. When the macro is undefined, all locking calls compile to no-ops and the build has zero overhead from the concurrency machinery; Single-threaded deployments pay nothing for the abstraction. When the macro is defined but no callback table is registered for a given resource, the runtime also degrades to no-op locking; this is the right behavior for a thread-safe build that nonetheless drives a particular subsystem from a single thread.

Locking is scoped per resource rather than global, so the topology is flexible: a deployment can share one mutex across all subsystems, give each shared resource its own mutex, or anywhere in between.

5.12.3 Concurrent Server Pattern

The reference servers shipped with most platform ports use a single-threaded round-robin loop: one thread iterates over each registered `whServerContext` and processes at most one request per pass. This is the simplest model and is sufficient for many embedded deployments.

When more concurrency is needed, a common pattern is to dedicate one thread to each server context:

```
int main(void)
{
    whServerContext serverA;
    whServerContext serverB;

    whServerConfig serverConfigA = { /* server configuration */ };
    whServerConfig serverConfigB = { /* server configuration */ };

    ThreadType threadA;
    ThreadType threadB;

    /* Bind server configuration to the transports, NVM, and platform locks
     * for shared resources - omitted for clarity */

    /* Initialize server contexts, binding to configuration */
    wh_Server_Init(&serverA, &serverConfigA);
    wh_Server_Init(&serverB, &serverConfigB);

    /* Create one processing thread per client connection */
    threadA = thread_create(serverThread, &serverA);
    threadB = thread_create(serverThread, &serverB);

    thread_join(threadA);
    thread_join(threadB);
```

```
    return 0;
}

/* Blocking request-processing loop for a single client */
void* serverThread(void* arg)
{
    whServerContext* server = (whServerContext*)arg;
    int ret = WH_ERROR_OK;

    while (ret == WH_ERROR_OK) {
        ret = wh_Server_HandleRequestMessage(server);
    }
    return NULL;
}
```

Each client is serviced by a dedicated thread, so requests from different clients can execute in parallel; scheduling priority is left entirely to the underlying OS or runtime. Production systems may replace the tight loop with a blocking wait on a transport-specific event or interrupt, calling `wh_Server_HandleRequestMessage()` only when work is available. Other valid strategies include dispatching from a transport interrupt, an event-driven reactor, or a worker pool — the API does not impose a scheduling model.

5.12.4 Transports and Concurrency

Server-side concurrency is independent of the transport layer. The transport only moves bytes between client and server; it does not determine how the server schedules request handling.

Because each client/server pair allows only one in-flight request, concurrency comes from running **multiple clients** in parallel, not from pipelining requests within a single session. Transports that serialize messages from many clients through a single shared channel — a hardware mailbox or MPSC ring buffer, for example — therefore do not increase server concurrency on their own; they would require an additional dispatch layer to fan messages back out to per-client server contexts. The highest practical concurrency on these platforms is typically achieved with the shared-memory transport, where each client owns a dedicated request/response buffer and the per-client server contexts can execute truly in parallel.

5.12.5 Crypto Under Concurrency

How a cryptographic operation behaves under concurrency depends on whether it is served by software or hardware:

- **Software crypto** runs entirely inside wolfCrypt using ephemeral per-request operation contexts, so it works naturally across concurrent server threads without any additional coordination.
- **Hardware crypto** can be approached in several ways. The most common is to rely on wolfCrypt's hardware abstraction layer, which serializes accelerator access using its own mutex mechanisms. Alternatively, the server application can restrict hardware access to a single privileged client by registering the hardware crypto callback only for that session — useful in safety-critical or real-time deployments where one client needs deterministic uncontended access. A third option is to use the crypto affinity feature and let clients themselves coordinate hardware use, which fits trusted-client environments with a cooperative allocation policy.

The right choice depends on the platform, the accelerator, and the application's contention profile; wolfHSM intentionally supports all three.

5.13 Authentication Manager

> **Note**: The authentication manager is currently **experimental** and has known issues. It is not yet suitable as a production security boundary.

wolfHSM provides an optional **authentication manager**, enabled with `WOLFHSM_CFG_ENABLE_AUTHENTICATION`, that authenticates clients to the server and checks every incoming request against a per-user permission model. It is transport-agnostic: the same login flow and authorization check apply over every supported transport.

The subsystem has three responsibilities:

- **Authentication**: verify a client's identity using a PIN or an X.509 certificate
- **Session tracking**: bind the authenticated identity to the client's connection so that subsequent requests carry that identity automatically
- **Authorization**: on every request, check that the active identity is permitted to invoke the requested operation, and reject the request if not

5.13.1 Authentication Methods

A client authenticates by calling `wh_Client_AuthLogin` with a username and one of two credential methods:

- **PIN** (`WH_AUTH_METHOD_PIN`): the client supplies a PIN; the server hashes it with SHA-256 and compares the digest in constant time against the stored hash.
- **Certificate** (`WH_AUTH_METHOD_CERTIFICATE`): the client supplies a DER-encoded X.509 certificate; the server verifies it against the trusted CA stored as that user's credential using the certificate manager. Requires `WOLFHSM_CFG_CERTIFICATE_MANAGER`.

A successful login returns a `whUserId` and records that identity as the active session for the connection. A failed login leaves the connection unauthenticated. `wh_Client_AuthLogout` clears the session, and the server also clears it automatically when the comm channel is closed so that a reconnecting client cannot inherit a stale identity.

5.13.2 Sessions and the Authorization Gate

Each server context carries at most one authenticated session at a time — the `whAuthContext` embedded in the server context holds the active `whUserId` and the user's permissions. A client that needs to operate as a different user must log out and log back in.

Every request received by the server is checked against the active session before it is dispatched to the corresponding subsystem handler:

- If no user is logged in, only comm-layer requests and the `LOGIN` action are permitted; every other request returns `WH_AUTH_PERMISSION_ERROR`.
- If a user is logged in, `LOGOUT` is always permitted, and every other request is gated against the user's permissions.

The gate lives in the server's front-end request handler, so individual subsystems do not need to perform their own auth checks. When the auth manager is compiled in but no auth context is configured at server initialization, the gate is skipped entirely and the server processes all requests without enforcement; this preserves compatibility with builds that do not need authentication.

5.13.3 Permissions

A user's permissions are described by a `whAuthPermissions` record with three pieces:

- **Group bitmap**: a per-group allow boolean. A request whose message group is not allowed is rejected without further checks.

- **Action bitmap**: for each allowed group, a 256-bit mask of which actions within that group are permitted. A request is allowed only if both its group and its action bit are set.
- **Admin flag**: a separate capability that gates user-management operations (`UserAdd`, `UserDelete`, `UserSetPermissions`) and cross-user logout. The core forbids non-admin sessions from promoting another user to admin regardless of backend behavior.

The helper macros `WH_AUTH_SET_ALLOWED_GROUP`, `WH_AUTH_SET_ALLOWED_ACTION`, `WH_AUTH_CLEAR_ALLOWED_GR` and `WH_AUTH_CLEAR_ALLOWED_ACTION` build permission sets at provisioning time, and `WH_AUTH_SET_IS_ADMIN` toggles the admin flag.

`whAuthPermissions` also carries a small per-user `keyIds` allowlist and the data model includes a `CheckKeyAuthorization` callback intended to constrain which keys a user may exercise. Per-key authorization is a placeholder in the current implementation — the callback is defined but no crypto or key handler invokes it yet.

Holding an action bit lets a session issue a request, but the backend still enforces per-target authorization on top of it. In the default base backend, `USER_SET_CREDENTIALS` lets a non-admin update its **own** credentials only (a cross-user credential change additionally requires admin and otherwise fails with `WH_ERROR_ACCESS`), while `USER_DELETE` and `USER_SET_PERMISSIONS` remain admin-only regardless of the action bit. The caller's `whUserId` is passed to the backend as `current_user_id` so it can distinguish a self-service change from a cross-user one. A custom auth backend may implement a different per-target policy.

5.13.4 User Management Invariants

Beyond the group/action gate, the core enforces a fixed set of invariants on user-management operations before the request reaches the storage backend. These checks live in `wh_Auth_UserAdd` (and its peers in `wh_auth.c`) so that they hold for every backend, including custom ones, and a backend cannot accidentally relax them.

When a session adds a user with `wh_Auth_UserAdd`, the following rules apply to a **non-admin** caller. An admin caller is exempt from rules 1 through 3 and is passed straight through to the backend.

1. **Only an admin can create an admin.** A non-admin caller that requests the admin flag on the new user is rejected with `WH_AUTH_PERMISSION_ERROR`. This complements the existing rule that a non-admin cannot promote an existing user to admin.
2. **New permissions must be a subset of the caller's own.** Every group bit, every action bit, and every entry in the new user's `keyIds` allowlist must already be held by the creating user. If the new user would gain any group, action, or key the caller does not itself possess, the request fails with `WH_AUTH_PERMISSION_ERROR`. A non-admin can therefore only ever delegate down, and cannot mint a user more privileged than itself.
3. **Non-admins cannot create credential-less users.** A non-admin caller must supply non-empty credentials for the new user. A `NULL` or zero-length credential is rejected with `WH_AUTH_PERMISSION_ERROR`. This prevents a non-admin from creating a back-door account that anyone can assume without authenticating. An admin may still create a user with no credentials, for example to provision the credential separately.
4. **No duplicate usernames.** The base backend rejects an add whose username matches an existing user with `WH_ERROR_BADARGS`, regardless of the caller's privilege.

Credential updates through `wh_Auth_UserSetCredentials` add a further check on top of the self-service rule described under Permissions. When the target user already has a credential set, the caller must also present the matching **current** credential. The base backend verifies it in constant time before accepting the replacement, and a missing or mismatched current credential fails with `WH_ERROR_ACCESS`. PINs are hashed with SHA-256 for both the comparison and storage, so the plaintext PIN is never retained.

`wh_Auth_UserDelete` and `wh_Auth_UserSetPermissions` remain admin-only operations in the base backend.

5.13.5 Pluggable Backend

The authentication manager does not own the user database itself. All operations that read or modify user state — login, user add/delete, permission updates, credential updates — are dispatched through a whAuthCb callback table that the application supplies at server initialization. The storage backend is therefore a port-time decision: an in-memory table for development, an NVM-backed store for production, or a connector to an external identity service.

wolfHSM ships with a default in-memory backend (wh_auth_base.c) used by the POSIX server example and the test suite. It holds up to WH_AUTH_BASE_MAX_USERS users in a static array, hashes PINs with SHA-256, and runs certificate verification against the user's stored CA when the certificate manager is built in. The base backend is intentionally simple and is **not** persisted to NVM — deployments that need user records to survive reset must supply their own backend.

Custom backends implement the whAuthCb vtable and register their context through whAuthConfig. The core handles locking, session state, and the request-time authorization gate; the backend is responsible for storage, credential verification, and any backend-specific overrides through the optional CheckRequestAuthorization and CheckKeyAuthorization callbacks. These overrides see the core's preliminary decision and can flip it either way, which lets a backend layer additional policy (time-of-day restrictions, audit hooks, per-key allowlists) on top of the default group/action check.

6 Utilities

This chapter describes the auxiliary tools that ship alongside the wolfHSM client and server libraries. These utilities are not part of the runtime library but support the workflows around it — provisioning an NVM image for a device, measuring the performance of a configured server, and validating that a port or build configuration behaves correctly. Each section describes *what* the utility does and *how* to drive it on the supported platforms; the underlying subsystems exercised by these tools are covered in and the API references.

6.1 Table of Contents

6.2 NVM Provisioning Tool

The NVM provisioning tool (`tools/whnvmtool/`) is a host-side utility that builds a pre-populated wolfHSM NVM image from a configuration file. It is intended for device provisioning: rather than having the server populate its NVM at runtime, the integrator describes the desired initial contents — a set of NVM objects and keys, each with its metadata ID, access permissions, flags, label, and a path to the binary payload — and the tool produces a single image file that can be programmed into the device's flash at manufacture or used in place to back a `whNvmFlash` provider in simulation. Currently the tool targets the `whNvmFlash` provider; the generated image is binary, and can be converted to Intel HEX with the standard `objcopy` workflow for use with automated programmers.

Because the on-flash layout depends on build-time configuration, the tool must be compiled against the same wolfHSM version as the target server and with a matching `WOLFHSM_CFG_NVM_OBJECT_COUNT`, and the `--size` argument must match the server's `whNvmFlash` partition size. For the full configuration file schema, command-line options, hex conversion recipe, and test workflow, see `tools/whnvmtool/README.md`.

6.3 Benchmark Suite

6.3.1 Benchmark Suite Overview

The benchmark suite (`benchmark/`) is a standalone wolfHSM client application that measures the round-trip cost of cryptographic operations against a configured wolfHSM server, from the perspective of a client. The numbers it reports therefore reflect the end-to-end performance a real client would observe in the same runtime environment: server-side computation, transport overhead, and any port-specific acceleration all rolled together.

The benchmark app consists of individual modules that each measure the various cryptographic algorithms that wolfHSM exposes. Each module runs its operation a configurable number of iterations, and the framework reports either operations per second or throughput in bytes per second, depending on the algorithm class.

The same client application builds and runs against any supported port: on POSIX the client and server run in separate threads of the host process, and on embedded targets the application links into the

port's runtime alongside a board-specific timer and `printf`. Iteration counts, data buffer sizes, DMA buffers, and timing/printing hooks are all overridable through `WOLFHSM_CFG_BENCH_*` macros so the suite can be tuned to the constraints of the target.

For the full list of configuration macros, the module interface, instructions for adding a new benchmark, and the internal layout of the framework, see `benchmark/README.md`.

6.3.2 Running Benchmarks on POSIX

To compile and run the benchmark application on a POSIX host system using the POSIX server port:

```
cd benchmark
make clean
make
make run
```

To use the DMA versions of algorithms that support it, pass the `DMA=1` environment variable to the build command:

```
make clean
make DMA=1
make run
```

6.3.3 Running Benchmarks on Real Hardware

Each hardware port ships with its own instructions for compiling and running the benchmark application on the target — refer to the README under the corresponding `port/<vendor>/<chip>/` directory for board-specific details such as toolchain setup, linker scripts, and timer configuration. At a high level, integrating the suite into a port-specific application reduces to three steps.

1. Compile the framework sources alongside the application. Add `benchmark/wh_bench.c`, `benchmark/wh_bench_ops.c`, and `benchmark/wh_bench_data.c` to the build, together with the per-algorithm modules under `benchmark/bench_modules/`. The standalone CLI wrapper in `benchmark/wh_bench_main.c` assumes a POSIX command-line environment and is typically omitted on embedded targets, where the application invokes the benchmark entry points directly.

2. Define the required configuration macros. At a minimum, define `WOLFHSM_CFG_BENCH_ENABLE` to compile the suite in. On non-POSIX targets the framework has no portable way to obtain wall-clock time, so the port must supply a microsecond timer with the signature `uint64_t timer(void)` and point `WOLFHSM_CFG_BENCH_CUSTOM_TIME_FUNC` at its name. If the platform lacks a working `printf`, define `WOLFHSM_CFG_BENCH_CUSTOM_PRINTF` to a port-supplied formatted-print routine. Iteration counts (`WOLFHSM_CFG_BENCH_CRYPT_ITERS`, `WOLFHSM_CFG_BENCH_KG_ITERS`, `WOLFHSM_CFG_BENCH_PK_ITERS`), buffer sizes (`WOLFHSM_CFG_BENCH_DATA_BUFFER_SIZE`, `WOLFHSM_CFG_BENCH_DMA_BUFFER_SIZE`), and buffer placement (`WOLFHSM_CFG_BENCH_CUSTOM_DATA_BUFFERS`, `WOLFHSM_CFG_BENCH_CUSTOM_DMA_BUFFER`) are all optional overrides used to fit the suite into the target's resource budget.

3. Drive the benchmark client from the application. The benchmark utility is purely client-side: it issues requests to a wolfHSM server and times the responses. The server it talks to can be any wolfHSM server instance that is listening on the same transport — most commonly the application's own production server, which exercises the exact configuration whose performance you are trying to measure. For convenience the suite also exposes `wh_Bench_ServerCfgLoop(whServerConfig*)`, a minimal server processing loop that dispatches incoming client requests until the client disconnects, which can be used in lieu of a production server when one is not yet available. On the client side, `wh_Bench_ClientCfg(whClientConfig*, int transport)` initializes a client, runs the full suite, and tears the client down; `wh_Bench_ClientCtx(whClientContext*, int transport)` is the equivalent entry point when the application already manages the client context lifecycle. How the

client and the server are scheduled relative to each other — separate cores, separate tasks, or cooperatively from a main loop — is determined by the port.

6.4 Test Suite

6.4.1 Test Suite Overview

The test suite (`test/`) is a standalone wolfHSM application that exercises the library's unit and integration tests against a configured client and server. Tests are grouped one-per-component (NVM, comm, crypto, keystore, certificates, SHE, image manager, authentication, etc.), each in its own `wh_test_*.c` source file, and are wired together by the top-level driver in `wh_test.c`. The suite validates wolfHSM itself rather than wolfCrypt; the full wolfCrypt test suite can additionally be run as a wolfHSM client by enabling `WOLFHSM_CFG_TEST_WOLFCRYPTTEST`, which exercises the crypto callback path end-to-end.

The same test sources build for POSIX hosts and for embedded targets. Tests that depend on POSIX facilities (sockets, pthreads, file-backed flash) are compiled in only when `WOLFHSM_CFG_TEST_POSIX` is defined, so an embedded port pulls in just the portable subset and selects whichever modules its configuration supports. Output goes through `WOLFHSM_CFG_PRINTF` and assertions go through `WOLFHSM_CFG_TEST_ASSERT_FUNC`, so both can be redirected to port-supplied implementations.

For the full list of test modules, supported build options, and code coverage workflow, see `test/README.md`.

6.4.2 Running Tests on POSIX

To compile and run the full test suite on a POSIX host system using the POSIX server port:

```
cd test
make clean
make
make run
```

Feature-specific builds are selected with the same makefile variables documented in — for example `DMA=1`, `SHE=1`, `AUTH=1`, `TLS=1`, `THREADSAFE=1`, `TESTWOLFCRYPT=1`. Development and CI builds also commonly set `ASAN=1` (address sanitizer), `TSAN=1` (thread sanitizer, mutually exclusive with `ASAN`), `DEBUG=1`, or `COVERAGE=1` (instruments the build for `gcovr`; see `make coverage`).

To build a client-only driver that connects to an already-running server over TCP (or TLS when `TLS=1` is set), pass `CLIENT_ONLY=1`:

```
make clean
make CLIENT_ONLY=1
make run
```

6.4.3 Running Tests on Real Hardware

Each hardware port ships with its own instructions for compiling and running the test suite on the target — refer to the README under the corresponding `port/<vendor>/<chip>/` directory for board-specific details. At a high level, integrating the suite into a port-specific application reduces to three steps.

1. Compile the framework sources alongside the application. Add `test/wh_test.c`, `test/wh_test_common.c`, and the per-module `test/wh_test_*.c` files for the components you wish to validate. Each module is independent, so an embedded port can pick a subset (e.g. crypto, keystore, certificates) and omit modules whose features the build does not enable. Leave `WOLFHSM_CFG_TEST_POSIX` undefined so the POSIX-only paths (sockets, pthreads, file-backed flash) are excluded.

2. Define the required configuration macros. Define `WOLFHSM_CFG_TEST_UNIT_NO_MAIN` to suppress the default `main()` so the application can call the test entry points itself. If the platform lacks a working stdlib `assert()`, define `WOLFHSM_CFG_TEST_ASSERT_FUNC` to a port-supplied assertion routine; output is already routed through `WOLFHSM_CFG_PRINTF` (see). Enable individual test categories by defining the same feature macros that gate them in the library (e.g. `WOLFHSM_CFG_DMA`, `WOLFHSM_CFG_SHE_EXTENSION`, `WOLFHSM_CFG_TEST_WOLFCRYPTTEST`).

3. Drive the test entry points from the application. Like the benchmark suite, the tests are purely client-side: they issue requests to a wolfHSM server and validate the responses. The server can be any wolfHSM server listening on the same transport — typically the application's own production server. To run the full client-side test set against a client the application has already configured, call `whTest_ClientConfig(whClientConfig*)`; individual modules expose their own client entry points (`whTest_CryptoClientConfig`, `whTest_SheClientConfig`, `whTest_TimeoutClientConfig`, etc.) when only a subset is desired. The suite also exposes `whTest_ServerCfgLoop(whServerConfig*)`, a minimal server processing loop suitable for in-process or co-resident-core test runs when no production server is available. How the client and server are scheduled relative to each other — separate cores, separate tasks, or cooperatively from a main loop — is determined by the port.

7 Examples

This chapter describes the example applications and demo code that ship with wolfHSM. These are *not* part of the runtime library — they live under `examples/` and exist so that an integrator can see a complete, end-to-end wolfHSM client and server running on a host, and so that a developer can find a working, runnable answer to the question "how do I do *X* in wolfHSM?" The example applications wire together the transport, comm, NVM, and crypto subsystems described in ; the demos are the port-agnostic exercises those applications run.

7.1 Table of Contents

7.2 POSIX Example Server and Client

The POSIX example consists of two simple host applications — a server (`examples/posix/wh_posix_server/`) and a client (`examples/posix/wh_posix_client/`) — that talk to each other over a transport of the user's choosing. They are intentionally minimal: each is a single `main()` that configures the appropriate context, calls the wolfHSM init routines, and then either services requests (server) or issues them (client). They are the smallest complete demonstration of a wolfHSM deployment on a system with a real OS, and they double as the reference for what a port-specific server and client application need to do at startup.

The server runs as a foreground process that waits for a client to connect on the selected transport, dispatches incoming requests through `wh_Server_HandleRequestMessage()`, and exits when the client disconnects. The client connects to a running server, sends a fixed sequence of `Echo` messages to verify the transport is alive, and — if invoked with `--test` — runs the full demo client library against the server before disconnecting. Both applications share a small set of helper configuration sources (`wh_posix_*_cfg.c`) that build the per-transport `whClientConfig` / `whServerConfig` structures; the configuration helpers are deliberately separated from the `main()` driver so the same transport plumbing can be lifted into an integrator's own application.

7.2.1 Building and Running

The applications expect to find wolfHSM and wolfSSL as sibling directories. From a clean checkout:

```
cd examples/posix/wh_posix_server && make
cd examples/posix/wh_posix_client && make
```

This produces `wh_posix_server.elf` and `wh_posix_client.elf` under each directory's `Build/`. Launch the server in one shell and the client in another:

```
./examples/posix/wh_posix_server/Build/wh_posix_server.elf
./examples/posix/wh_posix_client/Build/wh_posix_client.elf
```

To exercise the demo client library against the running server, pass `--test` to the client:

```
./wh_posix_client.elf --test
```

The full set of supported build options for the example applications is documented in ; the same `DMA=1`, `SHE=1`, `AUTH=1`, `TLS=1` knobs that gate features in the test and benchmark suites apply here.

7.2.2 Transport Selection

Both applications accept a `--type <transport>` argument that selects which configuration helper builds the comm context. The supported types depend on how the example was compiled:

- `tcp` — POSIX TCP socket on `127.0.0.1:23456` (the default).
- `shm` — POSIX inter-process shared memory.
- `dma` — Shared memory plus a wolfSSL static-memory DMA buffer, where requests pass DMA offsets instead of inline payloads. Requires `WOLFSSL_STATIC_MEMORY`.
- `tls` — wolfSSL TLS over TCP. Requires `WOLFHSM_CFG_TLS`.
- `psk` — TLS with a pre-shared key. Requires `WOLFHSM_CFG_TLS` and `!NO_PSK`.

The server and client must be started with the same `--type`. Each transport is defined by a corresponding `wh_PosixServer_Example<Type>Config() / wh_PosixClient_Example<Type>Config()` helper that an integrator can copy verbatim into their own application.

7.2.3 Server NVM Initialization

By default the server starts with an empty NVM. Two arguments let the example pre-populate it before the dispatch loop begins:

- `--key <path> --id <keyId> [--client <clientId>]` loads a single DER-encoded key from disk and caches it under the specified key ID for the specified client (default client 12).
- `--nvminit <path>` reads a whnvmtool-style configuration file and loads every key and object it lists.

Both forms are conveniences specific to the POSIX example — in a real deployment, NVM contents typically come from a pre-built image programmed at manufacture (see). They are included here so the demo client library can exercise paths that assume keys or objects already exist on the server.

7.3 Demo Client Library

The demo client library (`examples/demo/client/`) is a collection of self-contained C functions, one file per feature area, that drive the wolfHSM client API through a representative workflow for each subsystem. The library is port-agnostic: it depends only on a fully initialized `whClientContext*` and the wolfHSM client headers. The POSIX example client is one consumer of this library, but the same code compiles and runs from any port that can hand it an initialized client context — and that is the point.

7.3.1 Philosophy

The demos are **living documentation expressed as code**. They are written to answer "how do I do *X* in wolfHSM?" by being the shortest complete program that does *X*, with the surrounding setup and teardown spelled out so the reader can copy and adapt without guessing. Because the library is checked into the same repository as the runtime code, it cannot fall out of sync with the API the way prose documentation can: if a function signature changes, the demos break the build, and the demos are updated as part of the same change.

Two consequences of that philosophy are worth calling out explicitly, because they make the demo code look different from the rest of the codebase:

- **Aggressive inline commentary at the expense of error checking.** A production caller would propagate every return code, free every allocation on every path, and handle every edge case. The demos generally check return codes only enough to bail out cleanly at the top level and instead spend their lines explaining *why* the next call is shaped the way it is. The goal is for the reader to walk away understanding the *intent* of the sequence, not a robust template they can

paste into production. Production code should treat the demos as a starting point and add the missing rigor.
- **Clarity over efficiency.** The demos preference straight-line code, fixed-size local buffers, and explicit step-by-step sequences over the more compact or efficient idioms a production integration would use. Where there is a tension between "how a developer learns this" and "how a developer should ship this," the demos pick the former.

Each demo function takes a `whClientContext*` and returns `0` on success or a wolfHSM error code on failure, so the same set of demos can be wired into an integrator's own application — not just the POSIX example client — by calling them after their `wh_Client_Init()` succeeds. The top-level `wh_DemoClient_All()` in `wh_demo_client_all.c` runs the full suite in order and is what the POSIX client invokes when launched with `--test`.

7.3.2 Demo Categories

The demos are organized by wolfHSM feature area, with one source/header pair per area. Each category lives in `examples/demo/client/wh_demo_client_<area>.c` and is gated by the same build-time configuration macros as the underlying feature, so a demo for a feature that is not compiled in is simply elided from the suite.

- **NVM** (`wh_demo_client_nvm.c`) — Adding, reading back, enumerating, and reclaiming non-volatile objects through the client NVM API. Shows the full lifecycle of an NVM object including metadata, access flags, and reclamation of freed space.
- **Keystore** (`wh_demo_client_keystore.c`) — Caching raw key material in the server, querying cached keys by ID and label, committing cached keys to NVM, and using a cached key from a wolfCrypt operation. The reference for the basic key-cache / NVM-backing-store flow described in .
- **Key Wrapping** (`wh_demo_client_keywrap.c`) — Importing wrapped key blobs, unwrapping them on the server, and using the resulting cached key without ever exposing plaintext key material on the client. Gated by `WOLFHSM_CFG_KEYWRAP`.
- **Cryptography** (`wh_demo_client_crypto.c`) — The largest demo file: end-to-end signing, verification, key agreement, symmetric encryption, KDF, and MAC examples for each algorithm wolfHSM supports through the crypto callback path. Covers RSA, ECC, Curve25519, AES-CBC, AES-GCM, HKDF, CMAC, and CMAC-KDF, each in both an "import a key as part of the call" form and a "use a key already in the cache" form so the reader can see the two shapes of the API side-by-side.
- **Secure Boot** (`wh_demo_client_secboot.c`) — A complete provisioning-and-boot workflow: generating a server-side signing keypair, hashing an image with SHA-256, signing the hash, and later re-verifying it on boot. Acts as a worked example of how the keystore, NVM, and crypto subsystems compose for an image-authentication use case.
- **Authentication** (`wh_demo_client_auth.c`) — Logging in as a user, exercising role-based access controls on protected operations, and logging out. Gated by `WOLFHSM_CFG_ENABLE_AUTHENTICATION`. The POSIX example client logs in as an `admin` user after running the auth demos so subsequent demos run with full privileges.
- **Counters** (`wh_demo_client_counter.c`) — Reading, incrementing, and resetting non-volatile monotonic counters through the client counter API.
- **wolfCrypt Test Passthrough** (`wh_demo_client_wctest.c`) — Runs the standard wolfCrypt unit test suite as a wolfHSM client, exercising every supported algorithm through the crypto callback path. Gated by `WH_DEMO_WCTEST`. This is the same surface validated by the test utility in , but invoked from inside an example application rather than the standalone test runner.
- **wolfCrypt Benchmark Passthrough** (`wh_demo_client_wcbench.c`) — Runs the standard wolfCrypt benchmark as a wolfHSM client. Useful for sanity-checking that an integration produces sensible numbers; for thorough measurement use the dedicated benchmark suite.

To add a demo for a new feature, drop a new `wh_demo_client_<area>.c/.h` pair into `examples/demo/client/`, gate it on the appropriate `WOLFHSM_CFG_*` macro, and call it from

`wh_DemoClient_All()` under the same guard. The new demo is then automatically picked up by every port-specific example application that builds against the demo library.

8 Integration

This chapter describes how wolfHSM integrates with other wolfSSL libraries and products. wolfHSM is designed to slot into an existing wolfSSL-based stack rather than replace it: the same wolfCrypt API drives the cryptographic work, the same TLS stack secures network links, and the same firmware authentication tooling produces signed images. The sections below summarize the integration surface with each library, and refer the reader to the documentation for that library when the details live there.

8.1 Table of Contents

8.2 wolfSSL and wolfCrypt

wolfSSL/wolfCrypt is a hard dependency of wolfHSM and is used in three distinct roles, all of which are covered in detail elsewhere in this manual:

- **Cryptographic provider** on both sides of the client/server boundary. The client crypto callback routes wolfCrypt API calls to the server, and the server uses wolfCrypt to perform the actual cryptographic work. See Cryptography and wolfCrypt Integration and the broader discussion in Architecture.
- **Certificate and ASN.1 handling** behind the certificate manager — chain verification, leaf public key extraction, and acert parsing are all delegated to wolfSSL. See Certificate Management.
- **TLS transport** for client/server links that cross an untrusted network. The `posix_transport_tls` reference transport wraps the plain POSIX TCP transport in a wolfSSL-secured session (with optional PSK), with no change to the framing above. See Transport Backends.

Because wolfCrypt is already covered exhaustively in the architecture and feature chapters, this chapter does not repeat the integration details. Treat wolfSSL/wolfCrypt as a prerequisite of every wolfHSM build, and refer to those chapters when configuring it.

8.3 wolfBoot

wolfBoot is wolfSSL's portable secure bootloader. wolfBoot and wolfHSM are designed to work together: when both are present on a platform, wolfBoot can use wolfHSM for all of the cryptographic work and key storage it performs during firmware authentication, eliminating the need for wolfBoot to handle key material directly. The integration is bidirectional — wolfBoot can act as a wolfHSM *client*, or it can host an embedded wolfHSM *server* — and a wolfHSM server application can independently verify wolfBoot-formatted images on behalf of its own clients. This section summarizes the integration surface; For more information, refer to wolfBoot's `docs/wolfHSM.md`, which covers per-platform configuration, build options, and HAL requirements in detail.

8.3.1 wolfBoot as a wolfHSM Client

In **client mode**, wolfBoot is a wolfHSM client like any other application: it links the wolfHSM client library, opens a transport to a separate wolfHSM server, and offloads firmware signature verification

(and the hashing that feeds it) through the standard wolfCrypt crypto callback. Image signing keys are provisioned onto the server in advance, and wolfBoot references them by `keyId` rather than holding the key material itself. The build option `WOLFBOOT_ENABLE_WOLFHSM_CLIENT` selects this mode, and `WOLFBOOT_USE_WOLFHSM_PUBKEY_ID` together with the keygen `--nolocalkeys` option produces a keystore that contains only key metadata, with the actual public key resident on the HSM.

The wolfBoot HAL supplies the wolfHSM client context, the transport configuration, and the device/key identifiers (`hsmClientCtx`, `hsmDevIdHash`, `hsmDevIdPubKey`, `hsmKeyIdPubKey`) that wolfBoot uses to direct crypto callback calls at the right wolfHSM resources. wolfBoot's algorithm support over wolfHSM covers RSA-2048/3072/4096, ECDSA P-256/P-384/P-521, ML-DSA at security levels 2/3/5, and SHA-256, with the actual set available on a given target gated by the HAL.

This mode is the natural fit for a multi-core SoC where wolfBoot runs on the application core and the wolfHSM server runs on a separate secure core, communicating over the platform's shared-memory or mailbox transport. It is also the mode used by the wolfBoot simulator, which talks to the example POSIX TCP server over loopback.

8.3.2 wolfBoot on the wolfHSM Server

In **server mode**, wolfBoot links the wolfHSM *server* library and runs an embedded wolfHSM server inside the bootloader itself, using the wolfHSM server API directly rather than over a transport. There is no external HSM in this configuration; wolfBoot owns the NVM, the keystore, and the crypto subsystem, and any wolfHSM features it needs (notably certificate chain verification) are invoked locally. The build option `WOLFBOOT_ENABLE_WOLFHSM_SERVER` selects this mode, and is mutually exclusive with the client-mode option.

Server mode is the right choice when the bootloader has no separate secure core to delegate to but still benefits from wolfHSM's keystore, certificate manager, and NVM abstraction — for example, when the application that runs after boot is itself a wolfHSM server and the bootloader needs to share its provisioning. The HAL supplies the server context, NVM initialization, and the NVM IDs of any pre-provisioned root CA certificates used for certificate chain verification.

8.3.3 Verifying wolfBoot Images from a Server Application

Independently of which mode wolfBoot itself is running in, a wolfHSM **server application** can verify wolfBoot-formatted images on behalf of its clients using the image manager. The image manager understands the wolfBoot TLV header natively and exposes two verify methods specifically for wolfBoot images:

- `WH_IMG_MGR_IMG_TYPE_WOLFBOOT` — verifies the image against a key resident in the server's keystore, matching wolfBoot's standard signing model.
- `WH_IMG_MGR_IMG_TYPE_WOLFBOOT_CERT` — verifies a certificate chain embedded in the wolfBoot header against a trusted root in NVM and then uses the leaf public key to verify the image, matching wolfBoot's cert-chain signing mode.

The full mechanism — header parsing, signature TLV extraction, public-key-hint validation, and the DMA-aware payload reads that make verifying multi-megabyte images practical — is documented in wolfBoot Image Support. The practical upshot is that a wolfBoot client and a wolfHSM-equipped system can share a single image format and a single trust anchor: the same `.bin` that wolfBoot would verify locally can be verified by a wolfHSM server through the image manager, and the same root CA provisioned on the HSM works for both flows.

8.4 wolfIP

Coming soon ;-)

8.5 wolfGuard (WireGuard Transport)

Coming soon ;-)

8.6 wolfSentry

Coming soon ;-)

8.7 wolfTPM

Coming soon ;-)

9 Configuration

wolfHSM is configured entirely at build time through a set of WOLFHSM_CFG_XXX preprocessor macros. This chapter is an exhaustive reference for every supported configuration macro: what it does, what its default is, and which subsystem it affects. For an overview of *how* the configuration system is wired together, see Library Configuration; this chapter focuses on the macros themselves.

9.1 Table of Contents

9.2 How to Override Configuration Values

There are two supported ways to override a WOLFHSM_CFG_XXX value:

1. **Compiler command line**. Pass -DWOLFHSM_CFG_XXX=value (or -DWOLFHSM_CFG_XXX for boolean-style flags) when invoking the compiler. This is the simplest approach when only one or two values need to change, and is what the wolfHSM test/ and benchmark/ Makefiles use to map their make variables (DMA=1, SHE=1, THREADSAFE=1, etc.) onto the corresponding macros.
2. **User configuration header (wolfhsm_cfg.h)**. Create a header named wolfhsm_cfg.h containing #define WOLFHSM_CFG_XXX value statements, place it on the compiler's include search path, and define the top-level WOLFHSM_CFG macro when invoking the compiler (-DWOLFHSM_CFG). When WOLFHSM_CFG is defined, the central wolfhsm/wh_settings.h header includes wolfhsm_cfg.h first, so user-supplied values override the internal defaults. This is the recommended approach when more than a handful of options are being customized. The reference examples under examples/posix/ and test/config/ follow this pattern.

Every wolfHSM source file includes wolfhsm/wh_settings.h first. The header walks each WOLFHSM_CFG_XXX macro, supplies a default if the user has not defined one, and performs a small amount of cross-checking (for example, refusing to build WOLFHSM_CFG_KEYWRAP together with WOLFHSM_CFG_NO_CRYPTO). Every option listed in this chapter has a sensible default; the only macro that the user **must** supply is either WOLFHSM_CFG_PORT_GETTIME or WOLFHSM_CFG_NO_SYS_TIME (see Time and System Services).

9.3 Core Library and Roles

These macros select which halves of the wolfHSM library are compiled in, and gate features that are shared by both the client and server.

Macro	Default	Description
WOLFHSM_CFG_ENABLE_CLIENT	Undefined	If defined, compile client-side functionality (wh_Client_* APIs, crypto callback, message marshalling for client requests). Define this in client-only builds and in combined client/server builds.
WOLFHSM_CFG_ENABLE_SERVER	Undefined	If defined, compile server-side functionality (wh_Server_* APIs, request dispatcher, server-side keystore, NVM, crypto, etc.). Define this in server-only and combined builds.
WOLFHSM_CFG_ENABLE_TIMEOUT	Undefined	If defined, compile the client-side support for blocking request timeouts (wh_Client_SetRecvTimeout and the timeout-aware variants of the blocking client APIs). Requires a working WH_GETTIME_US().
WOLFHSM_CFG_NO_CRYPTO	Undefined	If defined, build wolfHSM without any wolfCrypt dependency. All crypto-related code, message types, and key-cache crypto paths are excluded; the resulting build is useful for porting and for purely transport/NVM-focused integrations. Incompatible with WOLFHSM_CFG_KEYWRAP.
WOLFHSM_CFG_INFOVERSION	"01.01.01"	String reported by the server in response to the version-info request. Override to embed a build- or vendor-specific version stamp.
WOLFHSM_CFG_INFOBUILD	"12345678"	String (typically a short git/build hash) reported by the server in response to the build-info request.

9.4 Communication and Protocol Sizing

These macros control the over-the-wire sizing of the request/response protocol shared between the client and server. The same value must be used on both ends of any given client/server pair.

Macro	Default	Description
`WOLFHSM_CFG_COMM_DATA_LEN`	`1280`	Maximum length, in bytes, of the data payload portion of a single request or response message. This sets the upper bound on how much data a single non-DMA request (key cache, certificate verify, large block crypto, etc.) can carry; messages larger than this must be split or use a DMA variant. Larger values raise per-context RAM usage and the size of any transport-side buffers.
`WOLFHSM_CFG_CUSTOMCB_LEN`	`256`	Maximum size, in bytes, of a single custom-callback message payload (see Custom Server Callbacks). Independent of `WOLFHSM_CFG_COMM_DATA_LEN`.

9.5 Time and System Services

wolfHSM relies on a microsecond-resolution system time for benchmark measurements, log timestamps, and (when enabled) request timeouts. Exactly one of the two macros below **must** be supplied by the port.

Macro	Default	Description
`WOLFHSM_CFG_PORT_GETTIME`	None — port must supply	Function-like macro that returns the current system time as a `uint64_t` count of microseconds. wolfHSM wraps it as `WH_GETTIME_US()` and uses it for timestamps and elapsed-time accounting. The POSIX port supplies `posixGetTime`; new ports must provide an equivalent.

Macro	Default	Description
WOLFHSM_CFG_NO_SYS_TIME	Undefined	If defined, all internal calls to obtain the system time return zero, removing the need for the port to supply WOLFHSM_CFG_PORT_GETTIME. Disables meaningful benchmark output and log timestamps; intended for very early porting work.

9.6 Cryptography Features

These macros enable or tune optional cryptographic subsystems built on top of wolfCrypt. All of them are silently ignored when WOLFHSM_CFG_NO_CRYPTO is defined.

Macro	Default	Description
WOLFHSM_CFG_SHE_EXTENSION	Undefined	If defined, compile the AUTOSAR SHE subsystem (SHE message types, SHE key slots, M1-M5 update protocol, SHE-specific RNG and SREG handling). Requires wolfCrypt built with AES, WOLFSSL_CMAC, WOLFSSL_AES_DIRECT, and HAVE_AES_ECB.
WOLFHSM_CFG_KEYWRAP	Undefined	If defined, compile the key-wrap subsystem (wh_Client_KeyWrap* / server counterparts). Uses AES-GCM internally and therefore requires wolfCrypt built with AES and HAVE_AESGCM. Incompatible with WOLFHSM_CFG_NO_CRYPTO.
WOLFHSM_CFG_KEYWRAP_MAX_KEY_SIZE	2048	Maximum size, in bytes, of a key that can be wrapped or unwrapped in a single operation. Only consulted when WOLFHSM_CFG_KEYWRAP is defined.
WOLFHSM_CFG_KEYWRAP_MAX_DATA_SIZE	2048	Maximum size, in bytes, of the plaintext or wrapped payload carried by a single key-wrap request. Only consulted when WOLFHSM_CFG_KEYWRAP is defined.

Macro	Default	Description
WOLFHSM_CFG_HWKEYSTORE	Undefined	If defined, compile the hardware keystore front-end (wh_HwKeystore_*) and hardware-only key support (WH_KEYTYPE_HW, WH_CLIENT_KEYID_MAKE_HW()). Hardware-only keys are served on demand by a user-supplied callback and are usable only as keywrap KEKs; they never enter the key cache or NVM and are never exported. See Hardware-Only Keys.
WOLFHSM_CFG_HWKEYSTORE_MAX_KEY_SIZE	32	Maximum size, in bytes, of a key served by the hardware keystore backend; sizes the local buffer that holds a hardware KEK for the duration of a keywrap operation. Only consulted when WOLFHSM_CFG_HWKEYSTORE is defined.
WOLFHSM_CFG_GLOBAL_KEYS	Undefined	If defined, enable the global-keys feature, allowing keys to be cached so that they are visible to every client rather than scoped to the caching client. See Global Keys for a full discussion of the API and security implications.
WH_DEV_ID	0x5748534D ("WHSM")	Value of the process-global crypto device ID registered by every wh_Client_Init() with the unified client crypto callback. Also the device ID bound to a client whose whClientConfig.devId is left 0. Override it if the default collides with another crypto-callback device ID in the application. See Transparent Offload via Crypto Callbacks for registration lifetime, multi-client rules, and wolfCrypt callback-table sizing (MAX_CRYPTO_DEVID_CALLBACKS, default 8).

Macro	Default	Description
WH_DEV_ID_DMA	0x57444D41("WDMA")	Value of the process-global DMA-only crypto device ID, registered by every wh_Client_Init() when WOLFHSM_CFG_DMA is defined. Reserved: not valid as a whClientConfig.devId. Override it if the default collides with another crypto-callback device ID in the application.

9.7 Keystore and Key Cache

These macros size the server-side key cache. The cache is split into “regular” slots (sized for common symmetric and EC keys) and “big” slots (sized for RSA-class keys); both are statically allocated.

Macro	Default	Description
WOLFHSM_CFG_SERVER_KEYCACHE_COUNT	8	Number of regular RAM key-cache slots on the server.
WOLFHSM_CFG_SERVER_KEYCACHE_BUFSIZE	256	Size, in bytes, of each regular key-cache slot.
WOLFHSM_CFG_SERVER_KEYCACHE_BIG_COUNT	1	Number of “big” RAM key-cache slots on the server, used for large keys (e.g. RSA, ML-DSA).
WOLFHSM_CFG_SERVER_KEYCACHE_BIG_BUFSIZE	1280	Size, in bytes, of each big key-cache slot. Should be at least the largest key the server is expected to hold (e.g. ~1024 bytes for an RSA-4096 private key).

9.8 NVM Storage

Macro	Default	Description
WOLFHSM_CFG_NVM_OBJECT_COUNT	32	Maximum number of objects the NVM directory can hold simultaneously (RAM directory cache *and* the on-disk directory it mirrors). Determines the upper bound on the number of keys, certificates, counters, and user objects that can coexist in NVM at one time.

Macro	Default	Description
WOLFHSM_CFG_SERVER_NVM_FLASH_LOG	Undefined	If defined, compile the log-structured NVM flash backend (wh_nvm_flash_log). When enabled it can be selected at runtime as an alternative to the regular flash backend; useful for flash parts that tolerate fewer erases or that prefer append-only update patterns.

9.9 Certificate Manager

Macro	Default	Description
WOLFHSM_CFG_CERTIFICATE_MANAGER	Undefined	If defined, compile the server-side certificate manager (trusted-root storage, chain verification, optional leaf-public-key caching). Required by WOLFHSM_CFG_CERTIFICATE_MANAGER_ACERT.
WOLFHSM_CFG_CERTIFICATE_MANAGER_ACERT	Undefined	If defined, also compile attribute-certificate (RFC 5755) support into the certificate manager. Requires wolfSSL built with WOLFSSL_ACERT and WOLFSSL_ASN_TEMPLATE.
WOLFHSM_CFG_MAX_CERT_SIZE	WOLFHSM_CFG_COMM_DATA_LEN, or 4096 when WOLFHSM_CFG_DMA is defined	Maximum size, in bytes, of a certificate that the manager will accept. The DMA default is larger because certificate verification requests no longer have to fit inside a single comm-buffer-sized message.
WOLFHSM_CFG_CERT_MAX_VERIFY_ROOTS	8	Maximum number of trusted-root NVM IDs accepted in a single wh_Server_CertVerifyMultiRoot request. Bounded so that the non-DMA wire request still fits within WOLFHSM_CFG_COMM_DATA_LEN alongside the candidate chain, and so the inline DMA request struct remains a fixed-size POD.

9.10 Image Manager

Macro	Default	Description
WOLFHSM_CFG_SERVER_IMG_MGR	Undefined	If defined, compile the server-side image manager (manifest-driven boot/runtime image verification).
WOLFHSM_CFG_SERVER_IMG_MGR_MAX_IMG_COUNT	4	Maximum number of images that a single image-manager configuration can track at one time.
WOLFHSM_CFG_SERVER_IMG_MGR_MAX_SIG_SIZE	512	Maximum signature size, in bytes, that the image manager will allocate buffer space for. The default accommodates RSA-4096; raise it when using signature schemes with larger signatures.

9.11 Custom Server Callbacks

Macro	Default	Description
WOLFHSM_CFG_SERVER_CUSTOMCB_COUNT	8	Number of custom-callback dispatch slots reserved on the server. Each registered callback occupies one slot; sets the upper bound on the number of distinct custom callback IDs an application can register.

9.12 DMA Support

These macros gate and tune DMA-mode crypto and large-buffer operations.

Macro	Default	Description
WOLFHSM_CFG_DMA	Undefined	If defined, compile the DMA-capable code paths: the per-client DMA dispatch mode (wh_Client_SetDmaMode()), DMA message types, pre/post access callbacks, the address allowlist machinery, and the WH_DEV_ID_DMA crypto device. Without this macro, DMA APIs are stubbed out.

Macro	Default	Description
WOLFHSM_CFG_DMAADDR_COUNT	10	Number of entries in the DMA address allowlist used by the server to validate client-supplied DMA buffers.
WOLFHSM_CFG_DMA_PTR_SIZE	Compiler-detected (__SIZEOF_POINTER__)	Override the assumed DMA pointer size, in bytes (must be 4 or 8). Auto-detection works for GCC/Clang and IAR; define this explicitly for any toolchain that does not provide __SIZEOF_POINTER__.
WOLFHSM_CFG_DMA_ALT_PTR_SIZE	Undefined	If defined, allows the DMA pointer size to differ from the native CPU pointer size (e.g. a 32-bit-pointer server reachable from a 64-bit-pointer client). When undefined, wh_settings.h refuses to build with a mismatched WOLFHSM_CFG_DMA_PTR_SIZE.
WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY	Undefined	If defined, expose hooks that let the integrator override the client-to-server and server-to-client memory copy used during DMA requests. Useful when DMA buffers live in shared memory that requires custom invalidation or cache maintenance beyond the standard XCACHE* macros.

9.13 Authentication

Macro	Default	Description
WOLFHSM_CFG_ENABLE_AUTHENTICATION	Undefined	If defined, compile the authentication manager on both client and server: session establishment, the authorization gate, per-client permissions, and the pluggable auth backend (wh_Auth_*). All authenticated message types and the in-request session header are conditional on this macro.

9.14 Concurrency and Thread Safety

Macro	Default	Description
WOLFHSM_CFG_THREADSAFE	Undefined	If defined, compile the lock abstraction (wh_Lock_*) into shared server resources: the global key cache, NVM operations, the authentication manager, and any port-supplied shared crypto hardware. Requires the port to supply lock callbacks via whLockConfig. When undefined, all lock operations expand to no-ops with zero runtime overhead. See Concurrency Support.

9.15 Transports

Macro	Default	Description
WOLFHSM_CFG_TLS	Undefined	If defined, compile the POSIX TLS transport (posix_transport_tls). Used by the POSIX example applications to wrap their client/server connection in wolfSSL TLS or PSK. Requires a wolfSSL build that includes the relevant TLS features (!NO_PSK for PSK mode, etc.).

9.16 Logging Subsystem

Macro	Default	Description
WOLFHSM_CFG_LOGGING	Undefined	If defined, compile the server-side logging subsystem (wh_log_*): structured log records, the ring-buffer backend, and the optional printf-style sink. Without this macro, the logging APIs are stubbed out.

Macro	Default	Description
WOLFHSM_CFG_LOG_MSG_MAX	256	Maximum size, in bytes, of a single log-message buffer, including the null terminator. Formatted log messages longer than this are truncated.

9.17 Debug and Print Configuration

These macros control the diagnostic output that wolfHSM emits at runtime. They are independent of WOLFHSM_CFG_LOGGING, which is a structured-event channel.

Macro	Default	Description
WOLFHSM_CFG_DEBUG	Undefined	If defined, enable the base debug print macros (WH_DEBUG_PRINT, WH_DEBUG_CLIENT, WH_DEBUG_SERVER). Without it, every debug print expands to do {} while (0).
WOLFHSM_CFG_DEBUG_VERBOSE	Undefined	If defined, additionally enable the verbose debug macros (WH_DEBUG_CLIENT_VERBOSE, WH_DEBUG_SERVER_VERBOSE) which include function name and line number, plus the verbose hexdump helper. Implies the prerequisites of WOLFHSM_CFG_DEBUG.
WOLFHSM_CFG_HEXDUMP	Auto-enabled when either debug macro is set	If defined, compile wh_Utils_Hexdump. Pulled in implicitly by either WOLFHSM_CFG_DEBUG or WOLFHSM_CFG_DEBUG_VERBOSE (and by the verbose hexdump helper), or can be defined manually for use by integrator code. Brings in <stdio.h>.
WOLFHSM_CFG_PRINTF	printf(<stdio.h>)	Function or function-like macro used as the underlying print primitive for every debug print. Must match the signature int func(const char* fmt, ...). Override this for targets without a working printf, or to redirect debug output through a vendor logging API.

9.18 Benchmark Suite

These macros are consumed only when building the wolfHSM benchmark suite (`benchmark/`). They are documented in more detail in and in `benchmark/README.md`.

Macro	Default	Description
`WOLFHSM_CFG_BENCH_ENABLE`	Undefined	If defined, compile the benchmark suite into the build. Required to use any of the other `BENCH_*` macros.
`WOLFHSM_CFG_BENCH_MAIN`	Undefined	If defined together with `WOLFHSM_CFG_BENCH_ENABLE`, also compile the standalone benchmark `main()` (`benchmark/wh_bench_main.c`).
`WOLFHSM_CFG_BENCH_CRYPT_ITERS`	1000	Number of iterations executed per symmetric-crypto benchmark (AES, HMAC, RNG, echo).
`WOLFHSM_CFG_BENCH_KG_ITERS`	10	Number of iterations executed per key-generation benchmark.
`WOLFHSM_CFG_BENCH_PK_ITERS`	10	Number of iterations executed per public-key benchmark (ECC sign/verify, RSA sign/verify, ML-DSA, Curve25519 KA).
`WOLFHSM_CFG_BENCH_DATA_BUFFER_SIZE`	0x4000 (16 KiB)	Size, in bytes, of each of the two static input/output buffers used by the data-plane benchmarks.
`WOLFHSM_CFG_BENCH_DMA_BUFFER_SIZE`	0x8000 (32 KiB)	Size, in bytes, of the static DMA buffer used by DMA benchmarks. Only consulted when `WOLFHSM_CFG_DMA` is defined.
`WOLFHSM_CFG_BENCH_CUSTOM_DATA_BUFFERS`	Undefined	If defined, the benchmark suite will not allocate its own input/output buffers and will instead use the addresses supplied by `WOLFHSM_CFG_BENCH_CUSTOM_DATA_IN_BUFFE` and `WOLFHSM_CFG_BENCH_CUSTOM_DATA_OUT_BUFF` Useful for placing buffers in a specific memory region (TCM, shared RAM, etc.).
`WOLFHSM_CFG_BENCH_CUSTOM_DATA_IN_BUFFER`	None	Address (cast to `void*`) of the user-provided input buffer. Consulted only when `WOLFHSM_CFG_BENCH_CUSTOM_DATA_BUFFERS` is defined.

Macro	Default	Description
WOLFHSM_CFG_BENCH_CUSTOM_DMA_OUT_BUFFER	None	Address of the user-provided output buffer; same conditions as above.
WOLFHSM_CFG_BENCH_CUSTOM_DMA_BUFFER	None	If defined, evaluates to the address of a user-supplied DMA buffer used in place of the static benchmark DMA buffer.
WOLFHSM_CFG_BENCH_INIT_DATA_BUFFERS	Undefined	If defined, benchmark modules re-initialize their input/output buffers with deterministic content before each measurement. Slightly slower per iteration, but produces more reproducible numbers.
WOLFHSM_CFG_BENCH_CUSTOM_PRINTF	Undefined	If defined, overrides the benchmark suite's print primitive with the named function or macro. Required on targets that do not provide a working `printf`. Independent of `WOLFHSM_CFG_PRINTF`.
WOLFHSM_CFG_BENCH_CUSTOM_TIME_FUNC	Undefined	If defined, overrides the benchmark suite's microsecond timer. The named function must have the signature `uint64_t func(void)`. Required on non-POSIX targets, where the suite has no portable way to obtain a wall-clock time.

9.19 Test Harness

These macros are consumed only by the wolfHSM test suite (`test/`) and its supporting infrastructure. They have no effect on a release build of the library. Most are toggled by `make` variables in `test/Makefile` (e.g. `make DMA=1 THREADSAFE=1 STRESS=1`).

Macro	Default	Description
WOLFHSM_CFG_TEST_POSIX	Undefined	If defined, compile tests and benchmark scaffolding that depend on POSIX APIs (pthreads, sockets, file-based flash, etc.). Set automatically by the POSIX test/benchmark builds.

Macro	Default	Description
WOLFHSM_CFG_TEST_CLIENT_ONLY	Undefined	If defined, build the unit tests as a client-only driver that expects to connect to an externally running server (e.g. one started by another process). Set by the CLIENT_ONLY=1 make variable.
WOLFHSM_CFG_TEST_CLIENT_ONLY_TCP	Undefined	Variant of the above that constrains the client-only build to the TCP transport, used by the integration harness that pairs a local client driver with a remote TCP server.
WOLFHSM_CFG_TEST_CLIENT_LARGE_DATA_DMA_ONLY	Undefined	If defined, tests that exercise large data payloads only run the DMA variant; the non-DMA equivalents are compiled out. Used on targets where the comm-buffer sizing cannot accommodate the large-data non-DMA path.
WOLFHSM_CFG_TEST_WOLFCRYPTTEST	Undefined	If defined, integrate the upstream wolfcrypt/test/test.c suite into the wolfHSM test driver and run it through the wolfHSM crypto callback. Set by the TESTWOLFCRYPT=1 make variable.
WOLFHSM_CFG_TEST_UNIT_NO_MAIN	Undefined	If defined, suppress the default main() provided by the test harness, allowing the test functions to be linked into an application that supplies its own entry point.
WOLFHSM_CFG_TEST_STRESS	Undefined	If defined, compile and run the POSIX thread-safety stress test (test/wh_test_posix_threadsafe_stress.c). Requires WOLFHSM_CFG_THREADSAFE and WOLFHSM_CFG_TEST_POSIX. Set by the STRESS=1 make variable.

Macro	Default	Description
WOLFHSM_CFG_TEST_STRESS_TSAN	Undefined	If defined, force the stress test to emit additional ThreadSanitizer annotations. Set automatically by the make TSAN=1 STRESS=1 combination.
WOLFHSM_CFG_TEST_STRESS_PHASE_MAX_ITERATIONS	8000	Number of iterations executed within a single phase of the threadsafe stress test.
WOLFHSM_CFG_TEST_STRESS_PHASE_TIMEOUT_SEC	Undefined (no timeout)	If defined, the stress test bails out of any single phase that has not finished within this many wall-clock seconds, reporting a timeout failure.
WOLFHSM_CFG_TEST_ASSERT_FUNC	Host libc assert()	If defined, overrides the macro used by the test harness to evaluate WH_TEST_ASSERT(...). Useful for redirecting assertion failures into a target-specific failure reporter.
WOLFHSM_CFG_TEST_ALLOW_PERSISTENT_NVM_ARTIFACTS	Undefined	If defined, NVM-touching tests are allowed to leave persistent artifacts behind (objects, counters, keys) between runs, which lets a separate test phase verify them. Otherwise NVM is reset between tests. Used by the POSIX test build.
WOLFHSM_CFG_TEST_CRYPTSVR_CFG	Implicitly defined	The crypto-server test pulls in its standard wolfCrypt test configuration unless NO_WOLFHSM_CFG_TEST_CRYPTSVR_CFG is defined at build time, in which case the integrator must supply their own configuration.
WOLFHSM_CFG_IS_TEST_SERVER	Undefined	If defined, the client-side unit tests assume they are talking to a server that is running additional test-only instrumentation (exercising edge cases that cannot be triggered against a standard server). Set automatically for the combined client/server POSIX test build; should not be defined outside of that harness.

9.20 Memory and Cache Porting Macros

For completeness, wolfHSM also relies on a small number of `X*` porting macros that are normally supplied alongside the `WOLFHSM_CFG_*` macros in `wolfhsm_cfg.h` even though they do not share the prefix:

- `XMEMFENCE()` — sequential memory fence (defaults to `__atomic_thread_fence(__ATOMIC_SEQ_CST)` on GCC/Clang; otherwise a no-op with a build warning).
- `XCACHELINE` — cache-line size in bytes (default 32).
- `XCACHEFLUSH(p)` / `XCACHEFLUSHBLK(p, n)` — flush one line / a range; defaults are no-op and `wh_Utils_CacheFlush` respectively.
- `XCACHEINVLD(p)` / `XCACHEINVLDBLK(p, n)` — invalidate one line / a range; defaults are no-op and `wh_Utils_CacheInvalidate` respectively.

These are the only knobs needed to make the shared-memory transports and DMA crypto paths safe on systems with separate I-cache/D-cache and on multicore SoCs without hardware cache coherency.

A Client API Reference

This chapter is the complete reference for the wolfHSM **client** API. It is generated directly from the documentation comments in the public client headers (`wolfhsm/wh_client.h`, `wolfhsm/wh_client_crypto.h`, and `wolfhsm/wh_client_she.h`), so it always tracks the source. For a conceptual, feature-oriented walkthrough of what these functions are for, see Features; this chapter documents the precise signatures, parameters, and return values.

- **Client API** — client context lifecycle, communication, NVM, keystore, certificate, image-manager, and counter operations (`wolfhsm/wh_client.h`).
- **Client Crypto API** — split-transaction, non-blocking crypto request/response calls (`wolfhsm/wh_client_crypt`
- **Client SHE API** — AUTOSAR SHE (Secure Hardware Extension) client interface: key update protocol (M1–M5), encrypted message handling, secure boot, deterministic PRNG, and status register access (`wolfhsm/wh_client_she.h`).

A.1 wolfhsm/wh_client.h

A.1.1 Types

	Name
enum	**wc_CipherType** { WC_CIPHER_NONE = 0}

A.1.2 Functions

	Name
int	**wh_Client_Init**(whClientContext * c, const whClientConfig * config)
int	**wh_Client_Cleanup**(whClientContext * c)Disconnects from the server and releases client context resources.
int	**wh_Client_SendRequest**(whClientContext * c, uint16_t group, uint16_t action, uint16_t data_size, const void * data)
int	**wh_Client_RecvResponse**(whClientContext * c, uint16_t * out_group, uint16_t * out_action, uint16_t * out_size, void * data)
int	**wh_Client_IsRequestPending**(const whClientContext * c)Reports whether a request has been sent whose matching response has not yet been consumed.
int	**wh_Client_CommInitRequest**(whClientContext * c)Sends a communication initialization request to the server.
int	**wh_Client_CommInitResponse**(whClientContext * c, uint32_t * out_clientid, uint32_t * out_serverid)Receives a communication initialization response from the server.
int	**wh_Client_CommInit**(whClientContext * c, uint32_t * out_clientid, uint32_t * out_serverid)Initializes communication with the server with a blocking call.

	Name
int	**wh_Client_CommInfoRequest**(whClientContext * c)Sends a communications information request to the server.
int	**wh_Client_CommInfoResponse**(whClientContext * c, uint8_t * out_version, uint8_t * out_build, uint32_t * out_cfg_comm_data_len, uint32_t * out_cfg_nvm_object_count, uint32_t * out_cfg_keycache_count, uint32_t * out_cfg_keycache_bufsize, uint32_t * out_cfg_keycache_bigcount, uint32_t * out_cfg_keycache_bigbufsize, uint32_t * out_cfg_customcb_count, uint32_t * out_cfg_dmaaddr_count, uint32_t * out_debug_state, uint32_t * out_boot_state, uint32_t * out_lifecycle_state, uint32_t * out_nvm_state)Receives a communication information response from the server.
int	**wh_Client_CommInfo**(whClientContext * c, uint8_t * out_version, uint8_t * out_build, uint32_t * out_cfg_comm_data_len, uint32_t * out_cfg_nvm_object_count, uint32_t * out_cfg_keycache_count, uint32_t * out_cfg_keycache_bufsize, uint32_t * out_cfg_keycache_bigcount, uint32_t * out_cfg_keycache_bigbufsize, uint32_t * out_cfg_customcb_count, uint32_t * out_cfg_dmaaddr_count, uint32_t * out_debug_state, uint32_t * out_boot_state, uint32_t * out_lifecycle_state, uint32_t * out_nvm_state)Retrieves server configuration and state with a blocking call.
int	**wh_Client_SetCryptoAffinity**(whClientContext * c, uint32_t affinity)Sets the crypto affinity on the client context.
int	**wh_Client_GetCryptoAffinity**(whClientContext * c, uint32_t * out_affinity)Gets the current crypto affinity from the client context.
int	**wh_Client_SetDmaMode**(whClientContext * c, int useDma)Turns the DMA path on or off for this client.
int	**wh_Client_GetDmaMode**(whClientContext * c, int * out_useDma)Gets the current DMA mode for this client.
int	**wh_Client_CommCloseRequest**(whClientContext * c)Sends a communication close request to the server.
int	**wh_Client_CommCloseResponse**(whClientContext * c)Receives a communication close response from the server.
int	**wh_Client_CommClose**(whClientContext * c)Closes communication with the server.

	Name
int	**wh_Client_EchoRequest**(whClientContext * c, uint16_t size, const void * data)Sends an echo request to the server.
int	**wh_Client_EchoResponse**(whClientContext * c, uint16_t * out_size, void * data)Receives an echo response from the server.
int	**wh_Client_Echo**(whClientContext * c, uint16_t snd_len, const void * snd_data, uint16_t * out_rcv_len, void * rcv_data)Sends an echo request to the server and receives the response.
int	**wh_Client_KeyCacheRequest_ex**(whClientContext * c, uint32_t flags, uint8_t * label, uint16_t labelSz, const uint8_t * in, uint16_t inSz, uint16_t keyId)Sends a key cache request to the server.
int	**wh_Client_KeyCacheRequest**(whClientContext * c, uint32_t flags, uint8_t * label, uint16_t labelSz, const uint8_t * in, uint16_t inSz)Sends a key cache request to the server.
int	**wh_Client_KeyCacheResponse**(whClientContext * c, uint16_t * keyId)Receives a key cache response from the server.
int	**wh_Client_KeyCache**(whClientContext * c, uint32_t flags, uint8_t * label, uint16_t labelSz, const uint8_t * in, uint16_t inSz, uint16_t * keyId)Sends a key cache request to the server and receives the response.
int	**wh_Client_KeyEvictRequest**(whClientContext * c, uint16_t keyId)Sends a key eviction request to the server.
int	**wh_Client_KeyEvictResponse**(whClientContext * c)Receives a key eviction response from the server.
int	**wh_Client_KeyEvict**(whClientContext * c, uint16_t keyId)Sends a key eviction request to the server and receives the response.
int	**wh_Client_KeyExportRequest**(whClientContext * c, uint16_t keyId)Sends a key export request to the server.
int	**wh_Client_KeyExportResponse**(whClientContext * c, uint8_t * label, uint16_t labelSz, uint8_t * out, uint16_t * outSz)Receives a key export response from the server.
int	**wh_Client_KeyExport**(whClientContext * c, uint16_t keyId, uint8_t * label, uint16_t labelSz, uint8_t * out, uint16_t * outSz)Sends a key export request to the server and receives the response.

	Name
int	**wh_Client_KeyExportPublicRequest**(whClientContext * c, whKeyId keyId, uint16_t algo)Sends a request to export only the public portion of a cached key.
int	**wh_Client_KeyExportPublicResponse**(whClientContext * c, uint8_t * label, uint16_t labelSz, uint8_t * out, uint16_t * outSz)Receives the public-key export response from the server.
int	**wh_Client_KeyExportPublic**(whClientContext * c, whKeyId keyId, uint16_t algo, uint8_t * label, uint16_t labelSz, uint8_t * out, uint16_t * outSz)Sends a public-key export request and receives the response.
int	**wh_Client_KeyCommitRequest**(whClientContext * c, whNvmId keyId)Sends a key commit request to the server.
int	**wh_Client_KeyCommitResponse**(whClientContext * c)Receives a key commit response from the server.
int	**wh_Client_KeyCommit**(whClientContext * c, whNvmId keyId)Sends a key commit request to the server and receives the response.
int	**wh_Client_KeyEraseRequest**(whClientContext * c, whNvmId keyId)Sends a key erase request to the server.
int	**wh_Client_KeyEraseResponse**(whClientContext * c)Receives a key erase response from the server.
int	**wh_Client_KeyErase**(whClientContext * c, whNvmId keyId)Sends a key erase request to the server and receives the response.
int	**wh_Client_KeyRevokeRequest**(whClientContext * c, whKeyId keyId)Sends a key revoke request to the server.
int	**wh_Client_KeyRevokeResponse**(whClientContext * c)Receives a key revoke response from the server.
int	**wh_Client_KeyRevoke**(whClientContext * c, whKeyId keyId)Sends a key revoke request to the server and receives the response.
int	**wh_Client_KeyCacheDmaRequest**(whClientContext * c, uint32_t flags, uint8_t * label, uint16_t labelSz, const void * keyAddr, uint16_t keySz, uint16_t keyId)Sends a key cache request using DMA to the server.
int	**wh_Client_KeyCacheDmaResponse**(whClientContext * c, uint16_t * keyId)Receives a key cache response for DMA from the server.

	Name
int	**wh_Client_KeyCacheDma**(whClientContext * c, uint32_t flags, uint8_t * label, uint16_t labelSz, const void * keyAddr, uint16_t keySz, uint16_t * keyId)Performs a complete key cache operation using DMA.
int	**wh_Client_KeyExportDmaRequest**(whClientContext * c, uint16_t keyId, const void * keyAddr, uint16_t keySz)Sends a key export request using DMA to the server.
int	**wh_Client_KeyExportDmaResponse**(whClientContext * c, uint8_t * label, uint16_t labelSz, uint16_t * outSz)Receives a key export response for DMA from the server.
int	**wh_Client_KeyExportDma**(whClientContext * c, uint16_t keyId, const void * keyAddr, uint16_t keySz, uint8_t * label, uint16_t labelSz, uint16_t * outSz)Performs a complete key export operation using DMA.
int	**wh_Client_KeyExportPublicDmaRequest**(whClientContext * c, whKeyId keyId, uint16_t algo, void * keyAddr, uint16_t keySz)Sends a public-key export DMA request to the server.
int	**wh_Client_KeyExportPublicDmaResponse**(whClientContext * c, uint8_t * label, uint16_t labelSz, uint16_t * outSz)Receives the public-key export DMA response from the server.
int	**wh_Client_KeyExportPublicDma**(whClientContext * c, whKeyId keyId, uint16_t algo, void * keyAddr, uint16_t keySz, uint8_t * label, uint16_t labelSz, uint16_t * outSz)Blocking wrapper around the public-key export DMA request/response.
int	**wh_Client_KeyWrap**(whClientContext * ctx, enum wc_CipherType cipherType, uint16_t serverKeyId, void * keyIn, uint16_t keySz, whNvmMetadata * metadataIn, void * wrappedKeyOut, uint16_t * wrappedKeyInOutSz)Sends a key wrap request to the server and receives the response.
int	**wh_Client_KeyWrapRequest**(whClientContext * ctx, enum wc_CipherType cipherType, uint16_t serverKeyId, void * key, uint16_t keySz, whNvmMetadata * metadata)Sends a key wrap request to the server.
int	**wh_Client_KeyWrapResponse**(whClientContext * ctx, enum wc_CipherType cipherType, void * wrappedKeyOut, uint16_t * wrappedKeyInOutSz)Receives a key wrap response from the server.

	Name
int	**wh_Client_KeyUnwrapAndExport**(whClientContext * ctx, enum wc_CipherType cipherType, uint16_t serverKeyId, void * wrappedKeyIn, uint16_t wrappedKeySz, whNvmMetadata * metadataOut, void * keyOut, uint16_t * keyInOutSz)Requests the server to unwrap and export a wrapped key and receives the response.
int	**wh_Client_KeyUnwrapAndExportRequest**(whClientContext * ctx, enum wc_CipherType cipherType, uint16_t serverKeyId, void * wrappedKeyIn, uint16_t wrappedKeySz)Requests the server to unwrap-and-export a wrapped key.
int	**wh_Client_KeyUnwrapAndExportResponse**(whClientContext * ctx, enum wc_CipherType cipherType, whNvmMetadata * metadataOut, void * keyOut, uint16_t * keyInOutSz)Receives an unwrap-and-export response from the server.
int	**wh_Client_KeyUnwrapAndCache**(whClientContext * ctx, enum wc_CipherType cipherType, uint16_t serverKeyId, void * wrappedKeyIn, uint16_t wrappedKeySz, uint16_t * keyIdOut)Requests the server to unwrap and cache a wrapped key and receives the response.
int	**wh_Client_KeyUnwrapAndCacheRequest**(whClientContext * ctx, enum wc_CipherType cipherType, uint16_t serverKeyId, void * wrappedKeyIn, uint16_t wrappedKeySz)Sends a key unwrap-and-cache request to the server.
int	**wh_Client_KeyUnwrapAndCacheResponse**(whClientContext * ctx, enum wc_CipherType cipherType, uint16_t * keyIdOut)Receives an unwrap-and-cache response from the server.
int	**wh_Client_DataWrap**(whClientContext * ctx, enum wc_CipherType cipherType, uint16_t serverKeyId, void * dataIn, uint32_t dataInSz, void * wrappedDataOut, uint32_t * wrappedDataInOutSz)Helper function to wrap a data object using a specified key.
int	**wh_Client_DataWrapRequest**(whClientContext * ctx, enum wc_CipherType cipherType, uint16_t serverKeyId, void * dataIn, uint32_t dataInSz)Sends a data wrap request to the server.
int	**wh_Client_DataWrapResponse**(whClientContext * ctx, enum wc_CipherType cipherType, void * wrappedDataOut, uint32_t * wrappedDataSz)Receives a data wrap response from the server.

	Name
int	**wh_Client_DataUnwrap**(whClientContext * ctx, enum wc_CipherType cipherType, uint16_t serverKeyId, void * wrappedDataIn, uint32_t wrappedDataInSz, void * dataOut, uint32_t * dataInOutSz)Helper function to unwrap a wrapped data object using a specified key.
int	**wh_Client_DataUnwrapRequest**(whClientContext * ctx, enum wc_CipherType cipherType, uint16_t serverKeyId, void * wrappedDataIn, uint32_t wrappedDataInSz)Sends a data unwrap request to the server.
int	**wh_Client_DataUnwrapResponse**(whClientContext * ctx, enum wc_CipherType cipherType, void * dataOut, uint32_t * dataSz)Receives a data unwrap response from the server.
int	**wh_Client_CounterInitRequest**(whClientContext * c, whNvmId counterId, uint32_t counter)
int	**wh_Client_CounterInitResponse**(whClientContext * c, uint32_t * counter)
int	**wh_Client_CounterInit**(whClientContext * c, whNvmId counterId, uint32_t * counter)Creates and initializes a counter with the value set in counter.
int	**wh_Client_CounterResetRequest**(whClientContext * c, whNvmId counterId)
int	**wh_Client_CounterResetResponse**(whClientContext * c, uint32_t * counter)
int	**wh_Client_CounterReset**(whClientContext * c, whNvmId counterId, uint32_t * counter)Creates and initializes a counter with to 0.
int	**wh_Client_CounterIncrementRequest**(whClientContext * c, whNvmId counterId)
int	**wh_Client_CounterIncrementResponse**(whClientContext * c, uint32_t * counter)
int	**wh_Client_CounterIncrement**(whClientContext * c, whNvmId counterId, uint32_t * counter)Increments a counter.
int	**wh_Client_CounterReadRequest**(whClientContext * c, whNvmId counterId)
int	**wh_Client_CounterReadResponse**(whClientContext * c, uint32_t * counter)
int	**wh_Client_CounterRead**(whClientContext * c, whNvmId counterId, uint32_t * counter)Read a counter.
int	**wh_Client_CounterDestroyRequest**(whClientContext * c, whNvmId counterId)
int	**wh_Client_CounterDestroyResponse**(whClientContext * c)
int	**wh_Client_CounterDestroy**(whClientContext * c, whNvmId counterId)Destroy a counter.

	Name
int	**wh_Client_NvmInitRequest**(whClientContext * c)Sends a non-volatile memory (NVM) initialization request to the server.
int	**wh_Client_NvmInitResponse**(whClientContext * c, int32_t * out_rc, uint32_t * out_clientnvm_id, uint32_t * out_servernvm_id)Receives a non-volatile memory (NVM) initialization response from the server.
int	**wh_Client_NvmInit**(whClientContext * c, int32_t * out_rc, uint32_t * out_clientnvm_id, uint32_t * out_servernvm_id)Sends a non-volatile memory (NVM) initialization request to the server and receives the response.
int	**wh_Client_NvmCleanupRequest**(whClientContext * c)Sends a non-volatile memory (NVM) cleanup request to the server.
int	**wh_Client_NvmCleanupResponse**(whClientContext * c, int32_t * out_rc)Receives a non-volatile memory (NVM) cleanup response from the server.
int	**wh_Client_NvmCleanup**(whClientContext * c, int32_t * out_rc)Sends a non-volatile memory (NVM) cleanup request to the server and receives the response.
int	**wh_Client_NvmGetAvailableRequest**(whClientContext * c)Sends a request to the server to get available non-volatile memory (NVM) information.
int	**wh_Client_NvmGetAvailableResponse**(whClientContext * c, int32_t * out_rc, uint32_t * out_avail_size, whNvmId * out_avail_objects, uint32_t * out_reclaim_size, whNvmId * out_reclaim_objects)Receives a response from the server with available non-volatile memory (NVM) information.
int	**wh_Client_NvmGetAvailable**(whClientContext * c, int32_t * out_rc, uint32_t * out_avail_size, whNvmId * out_avail_objects, uint32_t * out_reclaim_size, whNvmId * out_reclaim_objects)Sends a request to the server and receives a response with available non-volatile memory (NVM) information.
int	**wh_Client_NvmAddObjectRequest**(whClientContext * c, whNvmId id, whNvmAccess access, whNvmFlags flags, whNvmSize label_len, uint8_t * label, whNvmSize len, const uint8_t * data)Sends a request to the server to add an object to non-volatile memory (NVM).

	Name
int	**wh_Client_NvmAddObjectResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after attempting to add an object to non-volatile memory (NVM).
int	**wh_Client_NvmAddObject**(whClientContext * c, whNvmId id, whNvmAccess access, whNvmFlags flags, whNvmSize label_len, uint8_t * label, whNvmSize len, const uint8_t * data, int32_t * out_rc)Sends a request to the server and receives a response to add an object to non-volatile memory (NVM).
int	**wh_Client_NvmListRequest**(whClientContext * c, whNvmAccess access, whNvmFlags flags, whNvmId start_id)Sends a request to the server to list non-volatile memory (NVM) objects.
int	**wh_Client_NvmListResponse**(whClientContext * c, int32_t * out_rc, whNvmId * out_count, whNvmId * out_id)Receives a response from the server with a list of non-volatile memory (NVM) objects.
int	**wh_Client_NvmList**(whClientContext * c, whNvmAccess access, whNvmFlags flags, whNvmId start_id, int32_t * out_rc, whNvmId * out_count, whNvmId * out_id)Sends a request to the server and receives a response to list non-volatile memory (NVM) objects.
int	**wh_Client_NvmGetMetadataRequest**(whClientContext * c, whNvmId id)Sends a request to the server to get metadata of a non-volatile memory (NVM) object.
int	**wh_Client_NvmGetMetadataResponse**(whClientContext * c, int32_t * out_rc, whNvmId * out_id, whNvmAccess * out_access, whNvmFlags * out_flags, whNvmSize * out_len, whNvmSize label_len, uint8_t * label)Receives a response from the server with metadata of a non-volatile memory (NVM) object.
int	**wh_Client_NvmGetMetadata**(whClientContext * c, whNvmId id, int32_t * out_rc, whNvmId * out_id, whNvmAccess * out_access, whNvmFlags * out_flags, whNvmSize * out_len, whNvmSize label_len, uint8_t * label)Sends a request to the server and receives a response to get metadata of a non-volatile memory (NVM) object.
int	**wh_Client_NvmDestroyObjectsRequest**(whClientContext * c, whNvmId list_count, const whNvmId * id_list)Sends a request to the server to destroy non-volatile memory (NVM) objects.

	Name
int	**wh_Client_NvmDestroyObjectsResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after attempting to destroy non-volatile memory (NVM) objects.
int	**wh_Client_NvmDestroyObjects**(whClientContext * c, whNvmId list_count, const whNvmId * id_list, int32_t * out_rc)Sends a request to the server and receives a response to destroy non-volatile memory (NVM) objects.
int	**wh_Client_NvmReadRequest**(whClientContext * c, whNvmId id, whNvmSize offset, whNvmSize data_len)Sends a request to the server to read data from a non-volatile memory (NVM) object.
int	**wh_Client_NvmReadResponse**(whClientContext * c, int32_t * out_rc, whNvmSize * out_len, uint8_t * data)Receives a response from the server with NVM object data.
int	**wh_Client_NvmRead**(whClientContext * c, whNvmId id, whNvmSize offset, whNvmSize data_len, int32_t * out_rc, whNvmSize * out_len, uint8_t * data)Sends a request to the server and receives a response to read data from a non-volatile memory (NVM) object.
int	**wh_Client_NvmAddObjectDmaRequest**(whClientContext * c, whNvmMetadata * metadata, whNvmSize data_len, const uint8_t * data)Sends a request to the server to add an object to non-volatile memory (NVM) using DMA.
int	**wh_Client_NvmAddObjectDmaResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after attempting to add an object to non-volatile memory (NVM) using DMA.
int	**wh_Client_NvmAddObjectDma**(whClientContext * c, whNvmMetadata * metadata, whNvmSize data_len, const uint8_t * data, int32_t * out_rc)Sends a request to the server and receives a response to add an object to non-volatile memory (NVM) using DMA.
int	**wh_Client_NvmReadDmaRequest**(whClientContext * c, whNvmId id, whNvmSize offset, whNvmSize data_len, uint8_t * data)
int	**wh_Client_NvmReadDmaResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after attempting to read data from non-volatile memory (NVM) using DMA, with automatic detection of client address width (32-bit or 64-bit).

	Name
int	**wh_Client_NvmReadDma**(whClientContext * c, whNvmId id, whNvmSize offset, whNvmSize data_len, uint8_t * data, int32_t * out_rc)Sends a request to the server and receives a response to read data from non-volatile memory (NVM) using DMA, with automatic detection of client address width (32-bit or 64-bit).
int	**wh_Client_CustomCbRequest**(whClientContext * c, const whMessageCustomCb_Request * req)Sends a custom callback request to the server.
int	**wh_Client_CustomCbResponse**(whClientContext * c, whMessageCustomCb_Response * resp)Receives a response from the server after sending a custom callback request.
int	**wh_Client_CustomCheckRegisteredRequest**(whClientContext * c, uint32_t id)Sends a request to the server to check if a custom callback is registered.
int	**wh_Client_CustomCbCheckRegisteredResponse**(whClientCont * c, uint16_t * outId, int * responseError)Receives a response from the server after checking if a custom callback is registered.
int	**wh_Client_CustomCbCheckRegistered**(whClientContext * c, uint16_t id, int * responseError)Sends a request to the server and receives a response to check if a custom callback is registered.
int	**wh_Client_AuthLoginRequest**(whClientContext * c, whAuthMethod method, const char * username, const void * auth_data, uint16_t auth_data_len)Sends an authentication request to the server.
int	**wh_Client_AuthLoginResponse**(whClientContext * c, int32_t * out_rc, whUserId * out_user_id)Receives an authentication response from the server.
int	**wh_Client_AuthLogin**(whClientContext * c, whAuthMethod method, const char * username, const void * auth_data, uint16_t auth_data_len, int32_t * out_rc, whUserId * out_user_id)Authenticates a user with the server (blocking convenience wrapper).
int	**wh_Client_AuthLogoutRequest**(whClientContext * c, whUserId user_id)Sends a logout request to the server.
int	**wh_Client_AuthLogoutResponse**(whClientContext * c, int32_t * out_rc)Receives a logout response from the server.
int	**wh_Client_AuthLogout**(whClientContext * c, whUserId user_id, int32_t * out_rc)Logs out a user from the server (blocking convenience wrapper).

	Name
int	**wh_Client_AuthUserAddResponse**(whClientContext * c, int32_t * out_rc, whUserId * out_user_id)Receives a user add response from the server.
int	**wh_Client_AuthUserAddRequest**(whClientContext * c, const char * username, whAuthPermissions permissions, whAuthMethod method, const void * credentials, uint16_t credentials_len)Sends a user add request to the server.
int	**wh_Client_AuthUserAdd**(whClientContext * c, const char * username, whAuthPermissions permissions, whAuthMethod method, const void * credentials, uint16_t credentials_len, int32_t * out_rc, whUserId * out_user_id)Adds a new user to the server (blocking convenience wrapper).
int	**wh_Client_AuthUserGetRequest**(whClientContext * c, const char * username)Sends a user get request to the server.
int	**wh_Client_AuthUserGetResponse**(whClientContext * c, int32_t * out_rc, whUserId * out_user_id, whAuthPermissions * out_permissions)Receives a user get response from the server.
int	**wh_Client_AuthUserGet**(whClientContext * c, const char * username, int32_t * out_rc, whUserId * out_user_id, whAuthPermissions * out_permissions)Gets user information from the server (blocking convenience wrapper).
int	**wh_Client_AuthUserDeleteRequest**(whClientContext * c, whUserId user_id)Sends a user delete request to the server.
int	**wh_Client_AuthUserDeleteResponse**(whClientContext * c, int32_t * out_rc)Receives a user delete response from the server.
int	**wh_Client_AuthUserDelete**(whClientContext * c, whUserId user_id, int32_t * out_rc)Deletes a user from the server (blocking convenience wrapper).
int	**wh_Client_AuthUserSetPermissionsRequest**(whClientContext * c, whUserId user_id, whAuthPermissions permissions)Sends a user set permissions request to the server.
int	**wh_Client_AuthUserSetPermissionsResponse**(whClientContex * c, int32_t * out_rc)Receives a user set permissions response from the server.
int	**wh_Client_AuthUserSetPermissions**(whClientContext * c, whUserId user_id, whAuthPermissions permissions, int32_t * out_rc)Sets user permissions on the server (blocking convenience wrapper).

	Name
int	**wh_Client_AuthUserSetCredentialsRequest**(whClientContext * c, whUserId user_id, whAuthMethod method, const void * current_credentials, uint16_t current_credentials_len, const void * new_credentials, uint16_t new_credentials_len)Sends a user set credentials request to the server.
int	**wh_Client_AuthUserSetCredentialsResponse**(whClientContext * c, int32_t * out_rc)Receives a user set credentials response from the server.
int	**wh_Client_AuthUserSetCredentials**(whClientContext * c, whUserId user_id, whAuthMethod method, const void * current_credentials, uint16_t current_credentials_len, const void * new_credentials, uint16_t new_credentials_len, int32_t * out_rc)Sets user credentials on the server (blocking convenience wrapper).
int	**wh_Client_CertInitRequest**(whClientContext * c)Sends a request to initialize the certificate manager on the server.
int	**wh_Client_CertInitResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after initializing the certificate manager.
int	**wh_Client_CertInit**(whClientContext * c, int32_t * out_rc)Sends a request and receives a response to initialize the certificate manager.
int	**wh_Client_CertAddTrustedRequest**(whClientContext * c, whNvmId id, whNvmAccess access, whNvmFlags flags, uint8_t * label, whNvmSize label_len, const uint8_t * cert, uint32_t cert_len)Sends a request to add a trusted certificate to NVM storage.
int	**wh_Client_CertAddTrustedResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after adding a trusted certificate.
int	**wh_Client_CertAddTrusted**(whClientContext * c, whNvmId id, whNvmAccess access, whNvmFlags flags, uint8_t * label, whNvmSize label_len, const uint8_t * cert, uint32_t cert_len, int32_t * out_rc)Sends a request and receives a response to add a trusted certificate.
int	**wh_Client_CertEraseTrustedRequest**(whClientContext * c, whNvmId id)Sends a request to erase a trusted certificate from NVM storage.
int	**wh_Client_CertEraseTrustedResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after erasing a trusted certificate.
int	**wh_Client_CertEraseTrusted**(whClientContext * c, whNvmId id, int32_t * out_rc)Sends a request and receives a response to erase a trusted certificate.

	Name
int	**wh_Client_CertReadTrustedRequest**(whClientContext * c, whNvmId id, uint32_t cert_len)Sends a request to read a trusted certificate from NVM storage.
int	**wh_Client_CertReadTrustedResponse**(whClientContext * c, uint8_t * cert, uint32_t * cert_len, int32_t * out_rc)Receives a response from the server after getting a trusted certificate.
int	**wh_Client_CertReadTrusted**(whClientContext * c, whNvmId id, uint8_t * cert, uint32_t * cert_len, int32_t * out_rc)Sends a request and receives a response to read a trusted certificate.
int	**wh_Client_CertVerifyRequest**(whClientContext * c, const uint8_t * cert, uint32_t cert_len, whNvmId trustedRootNvmId)Sends a request to verify a certificate against trusted certificates.
int	**wh_Client_CertVerifyResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after verifying a certificate.
int	**wh_Client_CertVerify**(whClientContext * c, const uint8_t * cert, uint32_t cert_len, whNvmId trustedRootNvmId, int32_t * out_rc)Sends a request and receives a response to verify a certificate.
int	**wh_Client_CertVerifyAndCacheLeafPubKeyRequest**(whClientC * c, const uint8_t * cert, uint32_t cert_len, whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId keyId)Sends a request to verify a certificate and cache the leaf public key.
int	**wh_Client_CertVerifyAndCacheLeafPubKeyResponse**(whClient * c, whKeyId * out_keyId, int32_t * out_rc)Receives a response from the server after verifying a certificate and caching the leaf public key.
int	**wh_Client_CertVerifyAndCacheLeafPubKey**(whClientContext * c, const uint8_t * cert, uint32_t cert_len, whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId * inout_keyId, int32_t * out_rc)Sends a request and receives a response to verify a certificate, while also instructing the server to cache the public key of the leaf certificate.
int	**wh_Client_CertVerifyMultiRootRequest**(whClientContext * c, const uint8_t * cert, uint32_t cert_len, const whNvmId * trustedRootNvmIds, uint16_t numRoots)Sends a request to verify a certificate chain against a set of trusted root anchors.

	Name
int	**wh_Client_CertVerifyMultiRootResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after multi-root certificate verification.
int	**wh_Client_CertVerifyMultiRoot**(whClientContext * c, const uint8_t * cert, uint32_t cert_len, const whNvmId * trustedRootNvmIds, uint16_t numRoots, int32_t * out_rc)Sends a request and receives a response to verify a certificate chain against a set of trusted root anchors.
int	**wh_Client_CertVerifyMultiRootAndCacheLeafPubKeyRequest** * c, const uint8_t * cert, uint32_t cert_len, const whNvmId * trustedRootNvmIds, uint16_t numRoots, whNvmFlags cachedKeyFlags, whKeyId keyId)Sends a request to verify a certificate chain against a set of trusted root anchors and cache the leaf certificate public key.
int	**wh_Client_CertVerifyMultiRootAndCacheLeafPubKeyRespons** * c, whKeyId * out_keyId, int32_t * out_rc)Receives a response from the server after multi-root certificate verification with leaf public key caching.
int	**wh_Client_CertVerifyMultiRootAndCacheLeafPubKey**(whClien * c, const uint8_t * cert, uint32_t cert_len, const whNvmId * trustedRootNvmIds, uint16_t numRoots, whNvmFlags cachedKeyFlags, whKeyId * inout_keyId, int32_t * out_rc)Sends a request and receives a response to verify a certificate chain against a set of trusted root anchors and cache the leaf certificate public key.
int	**wh_Client_CertVerifyCacheClearRequest**(whClientContext * c)Send a request to clear the server's trusted certificate verify cache.
int	**wh_Client_CertVerifyCacheClearResponse**(whClientContext * c, int32_t * out_rc)Receive the response to a verify-cache clear request.
int	**wh_Client_CertVerifyCacheClear**(whClientContext * c, int32_t * out_rc)Synchronous helper to clear the server's trusted certificate verify cache.
int	**wh_Client_CertVerifyCacheSetEnabledRequest**(whClientConte * c, uint8_t enable)Send a request to enable or disable the server's trusted certificate verify cache at runtime.
int	**wh_Client_CertVerifyCacheSetEnabledResponse**(whClientCont * c, int32_t * out_rc)Receive the response to a verify-cache enable/disable request.
int	**wh_Client_CertVerifyCacheSetEnabled**(whClientContext * c, uint8_t enable, int32_t * out_rc)Synchronous helper to enable or disable the server's trusted certificate verify cache.

	Name
int	**wh_Client_CertAddTrustedDmaRequest**(whClientContext * c, whNvmId id, whNvmAccess access, whNvmFlags flags, uint8_t * label, whNvmSize label_len, const void * cert, uint32_t cert_len)Sends a request to add a trusted certificate to NVM storage using DMA.
int	**wh_Client_CertAddTrustedDmaResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after adding a trusted certificate using DMA.
int	**wh_Client_CertAddTrustedDma**(whClientContext * c, whNvmId id, whNvmAccess access, whNvmFlags flags, uint8_t * label, whNvmSize label_len, const void * cert, uint32_t cert_len, int32_t * out_rc)Sends a request and receives a response to add a trusted certificate using DMA.
int	**wh_Client_CertReadTrustedDmaRequest**(whClientContext * c, whNvmId id, void * cert, uint32_t cert_len)Sends a request to read a trusted certificate from NVM storage using DMA.
int	**wh_Client_CertReadTrustedDmaResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after reading a trusted certificate using DMA.
int	**wh_Client_CertReadTrustedDma**(whClientContext * c, whNvmId id, void * cert, uint32_t cert_len, int32_t * out_rc)Sends a request and receives a response to read trusted certificate using DMA.
int	**wh_Client_CertVerifyDmaRequest**(whClientContext * c, const void * cert, uint32_t cert_len, whNvmId trustedRootNvmId)Sends a request to verify a certificate using DMA.
int	**wh_Client_CertVerifyDmaResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after verifying a certificate using DMA.
int	**wh_Client_CertVerifyDma**(whClientContext * c, const void * cert, uint32_t cert_len, whNvmId trustedRootNvmId, int32_t * out_rc)Sends a request and receives a response to verify a certificate using DMA.
int	**wh_Client_CertVerifyDmaAndCacheLeafPubKeyRequest**(whCl * c, const void * cert, uint32_t cert_len, whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId keyId)Sends a request to verify a certificate using DMA and cache the leaf certificate public key.

	Name
int	**wh_Client_CertVerifyDmaAndCacheLeafPubKeyResponse**(wh * c, whKeyId * out_keyId, int32_t * out_rc)Receives a response from the server after verifying a certificate using DMA and caching the leaf public key.
int	**wh_Client_CertVerifyDmaAndCacheLeafPubKey**(whClientCont * c, const void * cert, uint32_t cert_len, whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId * inout_keyId, int32_t * out_rc)Sends a request and receives a response to verify a certificate using DMA and cache the leaf certificate public key.
int	**wh_Client_CertVerifyMultiRootDmaRequest**(whClientContext * c, const void * cert, uint32_t cert_len, const whNvmId * trustedRootNvmIds, uint16_t numRoots)Sends a DMA request to verify a certificate chain against a set of trusted root anchors.
int	**wh_Client_CertVerifyMultiRootDmaResponse**(whClientContex * c, int32_t * out_rc)Receives a response from the server after multi-root DMA certificate verification.
int	**wh_Client_CertVerifyMultiRootDma**(whClientContext * c, const void * cert, uint32_t cert_len, const whNvmId * trustedRootNvmIds, uint16_t numRoots, int32_t * out_rc)Sends a request and receives a response to verify a certificate chain via DMA against a set of trusted root anchors.
int	**wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKeyRe** * c, const void * cert, uint32_t cert_len, const whNvmId * trustedRootNvmIds, uint16_t numRoots, whNvmFlags cachedKeyFlags, whKeyId keyId)Sends a DMA request to verify a certificate chain against a set of trusted root anchors and cache the leaf certificate public key.
int	**wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKeyRes** * c, whKeyId * out_keyId, int32_t * out_rc)Receives a response from the server after multi-root DMA certificate verification with leaf public key caching.
int	**wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKey**(wh * c, const void * cert, uint32_t cert_len, const whNvmId * trustedRootNvmIds, uint16_t numRoots, whNvmFlags cachedKeyFlags, whKeyId * inout_keyId, int32_t * out_rc)Sends a request and receives a response to verify a certificate chain via DMA against a set of trusted root anchors and cache the leaf certificate public key.

	Name
int	**wh_Client_CertVerifyAcertRequest**(whClientContext * c, const void * cert, uint32_t cert_len, whNvmId trustedRootNvmId)Sends a request to verify an attribute certificate.
int	**wh_Client_CertVerifyAcertResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after verifying an attribute certificate.
int	**wh_Client_CertVerifyAcert**(whClientContext * c, const void * cert, uint32_t cert_len, whNvmId trustedRootNvmId, int32_t * out_rc)Sends a request and receives a response to verify an attribute certificate.
int	**wh_Client_CertVerifyAcertDmaRequest**(whClientContext * c, const void * cert, uint32_t cert_len, whNvmId trustedRootNvmId)Prepares and sends a DMA request to verify an attribute certificate.
int	**wh_Client_CertVerifyAcertDmaResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after verifying an attribute certificate using DMA.
int	**wh_Client_DmaRegisterAllowList**(struct whClientContext_t * client, const whDmaAddrAllowList * allowlist)Registers a DMA address allowlist for client-side validation.
int	**wh_Client_DmaRegisterCb**(struct whClientContext_t * client, whClientDmaClientMemCb cb)Registers a custom client DMA callback.
int	**wh_Client_DmaProcessClientAddress**(struct whClientContext_t * client, uintptr_t clientAddr, void ** serverPtr, size_t len, whDmaOper oper, whDmaFlags flags)Processes a client address for DMA operations, using the native pointer size of the system.
int	**wh_Client_DmaAsyncPre**(struct whClientContext_t * client, whClientDmaAsyncBuf * buf, uintptr_t clientAddr, uint64_t len, whDmaOper preOper, uintptr_t * outXformedAddr)Runs the PRE half of a DMA buffer mapping and stashes it (INTERNAL).
int	**wh_Client_DmaAsyncPost**(struct whClientContext_t * client, whClientDmaAsyncBuf * buf)Runs the POST half of a stashed DMA buffer mapping (INTERNAL).

	Name
int	**wh_Client_CertVerifyAcertDma**(whClientContext * c, const void * cert, uint32_t cert_len, whNvmId trustedRootNvmId, int32_t * out_rc)Sends a DMA request and receives a response to verify an attribute certificate.

A.1.3 Types Documentation

A.1.3.1 enum wc_CipherType

Enumerator	Value	Description
WC_CIPHER_NONE	0	

A.1.4 Functions Documentation

A.1.4.1 function wh_Client_Init

```
int wh_Client_Init(
    whClientContext * c,
    const whClientConfig * config
)
```

Parameters:

- **c** The pointer to the whClientContext object to be initialized.
- **config** The pointer to the whClientConfig object containing the configuration settings.

Return: Returns 0 on success, or a negative value indicating an error.

Context initialization and shutdown functions Initializes a whClientContext object with the provided configuration.

A.1.4.2 function wh_Client_Cleanup

```
int wh_Client_Cleanup(
    whClientContext * c
)
```

Disconnects from the server and releases client context resources.

Parameters:

- **c** A pointer to the whClientContext structure to be cleaned up.

Return: Returns 0 on success, or a negative value on failure.

This function frees any resources allocated during the initialization of the whClientContext. It should be called when the client is no longer needed

A.1.4.3 function wh_Client_SendRequest

```
int wh_Client_SendRequest(
    whClientContext * c,
    uint16_t group,
    uint16_t action,
```

```
    uint16_t data_size,
    const void * data
)
```

Parameters:

- **c** The client context.
- **group** The group identifier.
- **action** The action identifier.
- **data_size** The size of the data to be sent. Zero is allowed in the case of NULL data.
- **data** A pointer to the data to be sent. NULL is allowed in the case of zero-sized data.

Return: Returns 0 on success, or a negative value on failure.

Generic request/response functions Sends a request to the server using the specified client context.

A.1.4.4 function wh_Client_RecvResponse

```
int wh_Client_RecvResponse(
    whClientContext * c,
    uint16_t * out_group,
    uint16_t * out_action,
    uint16_t * out_size,
    void * data
)
```

Parameters:

- **c** The client context.
- **out_group** Pointer to store the received group value.
- **out_action** Pointer to store the received action value.
- **out_size** Pointer to store the received size value.
- **data** Pointer to store the received data.

Return: 0 if successful, a negative value if an error occurred.

Receives a response from the server and extracts the group, action, size, and data.

A.1.4.5 function wh_Client_IsRequestPending

```
int wh_Client_IsRequestPending(
    const whClientContext * c
)
```

Reports whether a request has been sent whose matching response has not yet been consumed.

Parameters:

- **c** The client context.

Return: 1 if a request is outstanding, 0 if idle, WH_ERROR_BADARGS on NULL or uninitialized context.

Does not mutate any context state. Intended to be polled from within client API *Request functions so they can fail fast before doing any payload construction or pre-send side effects (e.g. DMA address translation).

A.1.4.6 function wh_Client_CommInitRequest

```
int wh_Client_CommInitRequest(
    whClientContext * c
)
```

Sends a communication initialization request to the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

Comm component functions

This function prepares and sends a communication initialization request message to the server. It populates the message with the client's ID (initialized from the config struct at client initialization) and sends it using the communication context.

A.1.4.7 function wh_Client_CommInitResponse

```
int wh_Client_CommInitResponse(
    whClientContext * c,
    uint32_t * out_clientid,
    uint32_t * out_serverid
)
```

Receives a communication initialization response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_clientid** Pointer to store the client ID from the response.
- **out_serverid** Pointer to store the server ID from the response.

Return: int Returns 0 on success, or a negative error code on failure.

This function waits for and processes a communication initialization response message from the server. It validates the response and extracts the client and server IDs from the message.

A.1.4.8 function wh_Client_CommInit

```
int wh_Client_CommInit(
    whClientContext * c,
    uint32_t * out_clientid,
    uint32_t * out_serverid
)
```

Initializes communication with the server with a blocking call.

Parameters:

- **c** Pointer to the client context.
- **out_clientid** Pointer to store the client ID from the response.
- **out_serverid** Pointer to store the server ID from the response.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of initializing communication with the server. It sends an initialization request and waits for a valid response, extracting the client and server IDs from the response.

A.1.4.9 function wh_Client_CommInfoRequest

```
int wh_Client_CommInfoRequest(
    whClientContext * c
)
```

Sends a communications information request to the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a communication information request message to the server.

A.1.4.10 function wh_Client_CommInfoResponse

```
int wh_Client_CommInfoResponse(
    whClientContext * c,
    uint8_t * out_version,
    uint8_t * out_build,
    uint32_t * out_cfg_comm_data_len,
    uint32_t * out_cfg_nvm_object_count,
    uint32_t * out_cfg_keycache_count,
    uint32_t * out_cfg_keycache_bufsize,
    uint32_t * out_cfg_keycache_bigcount,
    uint32_t * out_cfg_keycache_bigbufsize,
    uint32_t * out_cfg_customcb_count,
    uint32_t * out_cfg_dmaaddr_count,
    uint32_t * out_debug_state,
    uint32_t * out_boot_state,
    uint32_t * out_lifecycle_state,
    uint32_t * out_nvm_state
)
```

Receives a communication information response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_version** Pointer to store the server version string (8 bytes)
- **out_build** Pointer to store the server build string (8 bytes)
- **out_cfg_comm_data_len** Pointer to store the server's maximum data len for any request or response
- **out_cfg_nvm_object_count** Pointer to store the server's maximum number of NVM objects
- **out_cfg_keycache_count** Pointer to store the server's number of keys in the server RAM
- **out_cfg_keycache_bufsize** Pointer to store the server's maximum size of each key in server RAM
- **out_cfg_keycache_bigcount** Pointer to store the server's number of big keys in the server RAM
- **out_cfg_keycache_bigbufsize** bufsize Pointer to store the server's maximum size of each big key in server RAM
- **out_cfg_customcb_count** Pointer to store the server's number of custom callbacks
- **out_cfg_dmaaddr_count** Pointer to store the server's number of dmaaddr regions Growth:
- **out_debug_state** Pointer to store the server's current debug state
- **out_boot_state** Pointer to store the server's current boot state
- **out_lifecycle_state** Pointer to store the server's lifecycle state
- **out_nvm_state** Pointer to store the server's current nvm state

Return: int Returns 0 on success, or a negative error code on failure.

This function waits for and processes a communication information response message from the server. It validates the response and extracts the server configuration data from the message.

A.1.4.11 function wh_Client_CommInfo

```
int wh_Client_CommInfo(
    whClientContext * c,
    uint8_t * out_version,
    uint8_t * out_build,
    uint32_t * out_cfg_comm_data_len,
    uint32_t * out_cfg_nvm_object_count,
    uint32_t * out_cfg_keycache_count,
    uint32_t * out_cfg_keycache_bufsize,
    uint32_t * out_cfg_keycache_bigcount,
    uint32_t * out_cfg_keycache_bigbufsize,
    uint32_t * out_cfg_customcb_count,
    uint32_t * out_cfg_dmaaddr_count,
    uint32_t * out_debug_state,
    uint32_t * out_boot_state,
    uint32_t * out_lifecycle_state,
    uint32_t * out_nvm_state
)
```

Retrieves server configuration and state with a blocking call.

Parameters:

- **c** Pointer to the client context.
- **out_version** Pointer to store the server version string (8 bytes)
- **out_build** Pointer to store the server build string (8 bytes)
- **out_cfg_comm_data_len** Pointer to store the server's maximum data len for any request or response
- **out_cfg_nvm_object_count** Pointer to store the server's maximum number of NVM objects
- **out_cfg_keycache_count** Pointer to store the server's number of keys in the server RAM
- **out_cfg_keycache_bufsize** Pointer to store the server's maximum size of each key in server RAM
- **out_cfg_keycache_bigcount** Pointer to store the server's number of keys in the server RAM
- **out_cfg_keycache_bigbufsize** Pointer to store the server's maximum size of each key in server RAM
- **out_cfg_customcb_count** Pointer to store the server's number of custom callbacks
- **out_cfg_dmaaddr_count** Pointer to store the server's number of dmaaddr regions Growth:
- **out_debug_state** Pointer to store the server's current debug state
- **out_boot_state** Pointer to store the server's current boot state
- **out_lifecycle_state** Pointer to store the server's lifecycle state
- **out_nvm_state** Pointer to store the server's current nvm state

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending communication info request and parsting the response from the server by busy polling for a valid response.

A.1.4.12 function wh_Client_SetCryptoAffinity

```
int wh_Client_SetCryptoAffinity(
    whClientContext * c,
    uint32_t affinity
)
```

Sets the crypto affinity on the client context.

Parameters:

- **c** Pointer to the client context.
- **affinity** Requested crypto affinity (WH_CRYPTO_AFFINITY_SW or WH_CRYPTO_AFFINITY_HW).

Return: int Returns 0 on success, or WH_ERROR_BADARGS on invalid input.

Affinity is stored locally and transmitted per-message in every crypto request. No round-trip to the server is required.

A.1.4.13 function wh_Client_GetCryptoAffinity

```
int wh_Client_GetCryptoAffinity(
    whClientContext * c,
    uint32_t * out_affinity
)
```

Gets the current crypto affinity from the client context.

Parameters:

- **c** Pointer to the client context.
- **out_affinity** Pointer to store the current crypto affinity.

Return: int Returns 0 on success, or WH_ERROR_BADARGS on invalid input.

A.1.4.14 function wh_Client_SetDmaMode

```
int wh_Client_SetDmaMode(
    whClientContext * c,
    int useDma
)
```

Turns the DMA path on or off for this client.

Parameters:

- **c** Pointer to the client context.
- **useDma** Nonzero to use DMA where supported, zero for the normal path.

Return: int 0 on success, or WH_ERROR_BADARGS on invalid input.

When on, operations that support DMA use it and the rest fall back to the normal path. You can change this at any time; it is stored locally and does not contact the server.

Always available: without WOLFHSM_CFG_DMA it does nothing and returns success, so you need not guard the call with #ifdef.

A.1.4.15 function wh_Client_GetDmaMode

```
int wh_Client_GetDmaMode(
    whClientContext * c,
    int * out_useDma
)
```

Gets the current DMA mode for this client.

Parameters:

- **c** Pointer to the client context.
- **out_useDma** Set to the current mode (0 or 1; always 0 without WOLFHSM_CFG_DMA).

Return: int 0 on success, or WH_ERROR_BADARGS on invalid input.

A.1.4.16 function wh_Client_CommCloseRequest

```
int wh_Client_CommCloseRequest(
    whClientContext * c
)
```

Sends a communication close request to the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a communication close request message to the server. It signals the server to close the communication channel with the client.

A.1.4.17 function wh_Client_CommCloseResponse

```
int wh_Client_CommCloseResponse(
    whClientContext * c
)
```

Receives a communication close response from the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

This function checks for and processes a communication close response message from the server.

A.1.4.18 function wh_Client_CommClose

```
int wh_Client_CommClose(
    whClientContext * c
)
```

Closes communication with the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of closing communication with the server. It sends a close request and waits for a valid response to confirm that the communication channel has been closed.

A.1.4.19 function wh_Client_EchoRequest

```
int wh_Client_EchoRequest(
    whClientContext * c,
    uint16_t size,
    const void * data
)
```

Sends an echo request to the server.

Parameters:

- **c** Pointer to the client context.
- **size** Size of the data payload.

- **data** Pointer to the data payload.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends an echo request message to the server. The message contains a data payload of the specified size. This function does not block; it returns immediately after sending the request.

A.1.4.20 function wh_Client_EchoResponse

```
int wh_Client_EchoResponse(
    whClientContext * c,
    uint16_t * out_size,
    void * data
)
```

Receives an echo response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_size** Pointer to store the size of the received data payload.
- **data** Pointer to store the received data payload.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process an echo response message from the server. It validates the response and extracts the data payload. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.21 function wh_Client_Echo

```
int wh_Client_Echo(
    whClientContext * c,
    uint16_t snd_len,
    const void * snd_data,
    uint16_t * out_rcv_len,
    void * rcv_data
)
```

Sends an echo request to the server and receives the response.

Parameters:

- **c** Pointer to the client context.
- **snd_len** Size of the data payload to send.
- **snd_data** Pointer to the data payload to send.
- **out_rcv_len** Pointer to store the size of the received data payload.
- **rcv_data** Pointer to store the received data payload.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending an echo request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response, extracting the data payload from the response once received. This function blocks until the entire operation is complete or an error occurs.

A.1.4.22 function wh_Client_KeyCacheRequest_ex

```
int wh_Client_KeyCacheRequest_ex(
    whClientContext * c,
    uint32_t flags,
    uint8_t * label,
    uint16_t labelSz,
    const uint8_t * in,
    uint16_t inSz,
    uint16_t keyId
)
```

Sends a key cache request to the server.

Parameters:

- **c** Pointer to the client context.
- **flags** Flags for the key cache request.
- **label** Pointer to the label associated with the key.
- **labelSz** Size of the label.
- **in** Pointer to the key data to be cached.
- **inSz** Size of the key data.
- **keyId** Key ID to be used for caching. If set to WH_KEYID_ERASED, a new ID will be generated.

Return: int Returns 0 on success, or a negative error code on failure.

Key functions

For client-side key data to be used, it must first be brought into the key cache (RAM) of the HSM server. Key cache requests instruct the server to transfer key data from client memory and allocate space in the HSM server RAM to hold this key. Key eviction requests instruct the HSM server to remove the key from the cache so that the RAM may be reused. Key export requests instruct the server to send back the cached key data to client RAM. Key commit requests instruct the HSM server to write the cached key into the HSM NVM. Key erase requests instruct the HSM server to remove a previously committed key from NVM.

This function prepares and sends a key cache request message to the server. The message contains the specified flags, label, and input data. This function does not block; it returns immediately after sending the request.

A.1.4.23 function wh_Client_KeyCacheRequest

```
int wh_Client_KeyCacheRequest(
    whClientContext * c,
    uint32_t flags,
    uint8_t * label,
    uint16_t labelSz,
    const uint8_t * in,
    uint16_t inSz
)
```

Sends a key cache request to the server.

Parameters:

- **c** Pointer to the client context.
- **flags** Flags for the key cache request.
- **label** Pointer to the label associated with the key.
- **labelSz** Size of the label.
- **in** Pointer to the key data to be cached.

- **inSz** Size of the key data.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a key cache request message to the server. The message contains the specified flags, label, and input data. This function does not block; it returns immediately after sending the request.

A.1.4.24 function wh_Client_KeyCacheResponse

```
int wh_Client_KeyCacheResponse(
    whClientContext * c,
    uint16_t * keyId
)
```

Receives a key cache response from the server.

Parameters:

- **c** Pointer to the client context.
- **keyId** Pointer to store the key ID assigned by the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a key cache response message from the server. It validates the response and extracts the key ID. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.25 function wh_Client_KeyCache

```
int wh_Client_KeyCache(
    whClientContext * c,
    uint32_t flags,
    uint8_t * label,
    uint16_t labelSz,
    const uint8_t * in,
    uint16_t inSz,
    uint16_t * keyId
)
```

Sends a key cache request to the server and receives the response.

Parameters:

- **c** Pointer to the client context.
- **flags** Flags for the key cache request.
- **label** Pointer to the label associated with the key.
- **labelSz** Size of the label.
- **in** Pointer to the key data to be cached.
- **inSz** Size of the key data.
- **keyId** Pointer to store the key ID assigned by the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a key cache request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response, extracting the key ID from the response once received. This function blocks until the entire operation is complete or an error occurs.

A.1.4.26 function wh_Client_KeyEvictRequest

```
int wh_Client_KeyEvictRequest(
    whClientContext * c,
    uint16_t keyId
)
```

Sends a key eviction request to the server.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to be evicted.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a key eviction request message to the server. The message contains the specified key ID. This function does not block; it returns immediately after sending the request.

A.1.4.27 function wh_Client_KeyEvictResponse

```
int wh_Client_KeyEvictResponse(
    whClientContext * c
)
```

Receives a key eviction response from the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a key eviction response message from the server. It validates the response. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.28 function wh_Client_KeyEvict

```
int wh_Client_KeyEvict(
    whClientContext * c,
    uint16_t keyId
)
```

Sends a key eviction request to the server and receives the response.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to be evicted.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a key eviction request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.29 function wh_Client_KeyExportRequest

```
int wh_Client_KeyExportRequest(
    whClientContext * c,
    uint16_t keyId
)
```

Sends a key export request to the server.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to be exported.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a key export request message to the server. The message contains the specified key ID. This function does not block; it returns immediately after sending the request.

A.1.4.30 function wh_Client_KeyExportResponse

```
int wh_Client_KeyExportResponse(
    whClientContext * c,
    uint8_t * label,
    uint16_t labelSz,
    uint8_t * out,
    uint16_t * outSz
)
```

Receives a key export response from the server.

Parameters:

- **c** Pointer to the client context.
- **label** Pointer to store the label associated with the key.
- **labelSz** Size of the label buffer.
- **out** Pointer to store the exported key data.
- **outSz** Pointer to store the size of the exported key data.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a key export response message from the server. It validates the response and extracts the label and key data. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.31 function wh_Client_KeyExport

```
int wh_Client_KeyExport(
    whClientContext * c,
    uint16_t keyId,
    uint8_t * label,
    uint16_t labelSz,
    uint8_t * out,
    uint16_t * outSz
)
```

Sends a key export request to the server and receives the response.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to be exported.

- **label** Pointer to store the label associated with the key.
- **labelSz** Size of the label buffer.
- **out** Pointer to store the exported key data.
- **outSz** Pointer to store the size of the exported key data.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a key export request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response, extracting the label and key data from the response once received. This function blocks until the entire operation is complete or an error occurs.

A.1.4.32 function wh_Client_KeyExportPublicRequest

```
int wh_Client_KeyExportPublicRequest(
    whClientContext * c,
    whKeyId keyId,
    uint16_t algo
)
```

Sends a request to export only the public portion of a cached key.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to export the public key of.
- **algo** Algorithm selector (WH_KEY_ALGO_*).

Return: int Returns 0 on success, or a negative error code on failure.

Unlike wh_Client_KeyExport(), which returns the raw cached DER (potentially including private material), this function instructs the server to re-emit only the public half of the cached public-key object. The private key stays inside the HSM. The algo selector identifies how the cached bytes should be interpreted (see WH_KEY_ALGO_ENUM in wh_common.h).

Non-blocking: returns immediately after sending.

A.1.4.33 function wh_Client_KeyExportPublicResponse

```
int wh_Client_KeyExportPublicResponse(
    whClientContext * c,
    uint8_t * label,
    uint16_t labelSz,
    uint8_t * out,
    uint16_t * outSz
)
```

Receives the public-key export response from the server.

Parameters:

- **c** Pointer to the client context.
- **label** Optional buffer to receive the cached key's label.
- **labelSz** Size of label buffer.
- **out** Buffer receiving the public-only DER.
- **outSz** In: size of out. Out: bytes written.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

Non-blocking: may return WH_ERROR_NOTREADY; callers should retry in that case. On success, writes the public DER into out and the associated label (if any) into label.

A.1.4.34 function wh_Client_KeyExportPublic

```
int wh_Client_KeyExportPublic(
    whClientContext * c,
    whKeyId keyId,
    uint16_t algo,
    uint8_t * label,
    uint16_t labelSz,
    uint8_t * out,
    uint16_t * outSz
)
```

Sends a public-key export request and receives the response.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to export the public key of.
- **algo** Algorithm selector (WH_KEY_ALGO_*).
- **label** Optional buffer to receive the cached key's label.
- **labelSz** Size of label buffer.
- **out** Buffer receiving the public-only DER.
- **outSz** In: size of out. Out: bytes written.

Return: int Returns 0 on success, or a negative error code on failure.

Blocking wrapper around wh_Client_KeyExportPublicRequest/Response.

A.1.4.35 function wh_Client_KeyCommitRequest

```
int wh_Client_KeyCommitRequest(
    whClientContext * c,
    whNvmId keyId
)
```

Sends a key commit request to the server.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to be committed. Committing a key means making it persistent in non-volatile memory.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a key commit request message to the server. The message contains the specified key ID. This function does not block; it returns immediately after sending the request.

A.1.4.36 function wh_Client_KeyCommitResponse

```
int wh_Client_KeyCommitResponse(
    whClientContext * c
)
```

Receives a key commit response from the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a key commit response message from the server. It validates the response. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.37 function wh_Client_KeyCommit

```
int wh_Client_KeyCommit(
    whClientContext * c,
    whNvmId keyId
)
```

Sends a key commit request to the server and receives the response.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to be committed. Committing a key means making it persistent in non-volatile memory.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a key commit request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.38 function wh_Client_KeyEraseRequest

```
int wh_Client_KeyEraseRequest(
    whClientContext * c,
    whNvmId keyId
)
```

Sends a key erase request to the server.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to be erased.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a key erase request message to the server. The message contains the specified key ID. This function does not block; it returns immediately after sending the request.

A.1.4.39 function wh_Client_KeyEraseResponse

```
int wh_Client_KeyEraseResponse(
    whClientContext * c
)
```

Receives a key erase response from the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a key erase response message from the server. It validates the response. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.40 function wh_Client_KeyErase

```
int wh_Client_KeyErase(
    whClientContext * c,
    whNvmId keyId
)
```

Sends a key erase request to the server and receives the response.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to be erased.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a key erase request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.41 function wh_Client_KeyRevokeRequest

```
int wh_Client_KeyRevokeRequest(
    whClientContext * c,
    whKeyId keyId
)
```

Sends a key revoke request to the server.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to be revoked.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a key revoke request message to the server. The message contains the specified key ID. This function does not block; it returns immediately after sending the request.

A.1.4.42 function wh_Client_KeyRevokeResponse

```
int wh_Client_KeyRevokeResponse(
    whClientContext * c
)
```

Receives a key revoke response from the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a key revoke response message from the server. It validates the response. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.43 function wh_Client_KeyRevoke

```
int wh_Client_KeyRevoke(
    whClientContext * c,
    whKeyId keyId
)
```

Sends a key revoke request to the server and receives the response.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to be revoked.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a key revoke request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.44 function wh_Client_KeyCacheDmaRequest

```
int wh_Client_KeyCacheDmaRequest(
    whClientContext * c,
    uint32_t flags,
    uint8_t * label,
    uint16_t labelSz,
    const void * keyAddr,
    uint16_t keySz,
    uint16_t keyId
)
```

Sends a key cache request using DMA to the server.

Parameters:

- **c** Pointer to the client context.
- **flags** Key flags.
- **label** Optional label for the key.
- **labelSz** Size of the label in bytes.
- **keyAddr** DMA address of the key data.
- **keySz** Size of the key in bytes.
- **keyId** Key ID to be associated with the cached key.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a key cache request message using DMA addressing to the server. The message contains the key data and metadata. This function does not block; it returns immediately after sending the request.

A.1.4.45 function wh_Client_KeyCacheDmaResponse

```
int wh_Client_KeyCacheDmaResponse(
    whClientContext * c,
```

```
    uint16_t * keyId
)
```

Receives a key cache response for DMA from the server.

Parameters:

- **c** Pointer to the client context.
- **keyId** Pointer to store the assigned key ID.

Return: int Returns 0 on success, or a negative error code on failure.

This function processes a key cache response message for a DMA operation from the server. It validates the response and returns the assigned key ID.

A.1.4.46 function wh_Client_KeyCacheDma

```
int wh_Client_KeyCacheDma(
    whClientContext * c,
    uint32_t flags,
    uint8_t * label,
    uint16_t labelSz,
    const void * keyAddr,
    uint16_t keySz,
    uint16_t * keyId
)
```

Performs a complete key cache operation using DMA.

Parameters:

- **c** Pointer to the client context.
- **flags** Key flags.
- **label** Optional label for the key.
- **labelSz** Size of the label in bytes.
- **keyAddr** DMA address of the key data.
- **keySz** Size of the key in bytes.
- **keyId** Pointer to store the assigned key ID.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of caching a key using DMA, including sending the request and receiving the response.

A.1.4.47 function wh_Client_KeyExportDmaRequest

```
int wh_Client_KeyExportDmaRequest(
    whClientContext * c,
    uint16_t keyId,
    const void * keyAddr,
    uint16_t keySz
)
```

Sends a key export request using DMA to the server.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to export.
- **keyAddr** DMA address where the key should be exported.
- **keySz** Size of the key buffer in bytes.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a key export request message using DMA addressing to the server.

A.1.4.48 function wh_Client_KeyExportDmaResponse

```
int wh_Client_KeyExportDmaResponse(
    whClientContext * c,
    uint8_t * label,
    uint16_t labelSz,
    uint16_t * outSz
)
```

Receives a key export response for DMA from the server.

Parameters:

- **c** Pointer to the client context.
- **label** Buffer to store the key's label.
- **labelSz** Size of the label buffer.
- **outSz** Pointer to store the actual size of the exported key.

Return: int Returns 0 on success, or a negative error code on failure.

This function processes a key export response message for a DMA operation from the server.

A.1.4.49 function wh_Client_KeyExportDma

```
int wh_Client_KeyExportDma(
    whClientContext * c,
    uint16_t keyId,
    const void * keyAddr,
    uint16_t keySz,
    uint8_t * label,
    uint16_t labelSz,
    uint16_t * outSz
)
```

Performs a complete key export operation using DMA.

Parameters:

- **c** Pointer to the client context.
- **keyId** Key ID to export.
- **keyAddr** DMA address where the key should be exported.
- **keySz** Size of the key buffer in bytes.
- **label** Buffer to store the key's label.
- **labelSz** Size of the label buffer.
- **outSz** Pointer to store the actual size of the exported key.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of exporting a key using DMA, including sending the request and receiving the response.

A.1.4.50 function wh_Client_KeyExportPublicDmaRequest

```
int wh_Client_KeyExportPublicDmaRequest(
    whClientContext * c,
    whKeyId keyId,
```

```
    uint16_t algo,
    void * keyAddr,
    uint16_t keySz
)
```

Sends a public-key export DMA request to the server.

Parameters:

- **c** Pointer to the client context.
- **keyId** Server key ID whose public key should be exported.
- **algo** Algorithm selector (WH_KEY_ALGO_*).
- **keyAddr** Client buffer that will receive the public DER.
- **keySz** Size of the keyAddr buffer in bytes.

Return: int Returns 0 on success, or a negative error code on failure.

DMA counterpart to wh_Client_KeyExportPublicRequest. The server decodes the cached key according to the algo selector, emits the public-only portion as DER, and DMAs that DER into the client buffer at keyAddr.

A.1.4.51 function wh_Client_KeyExportPublicDmaResponse

```
int wh_Client_KeyExportPublicDmaResponse(
    whClientContext * c,
    uint8_t * label,
    uint16_t labelSz,
    uint16_t * outSz
)
```

Receives the public-key export DMA response from the server.

Parameters:

- **c** Pointer to the client context.
- **label** Optional buffer to receive the cached key's label.
- **labelSz** Size of label buffer.
- **outSz** Receives the number of public-DER bytes the server wrote into the client buffer.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if the response is not yet available, or a negative error code on failure.

A.1.4.52 function wh_Client_KeyExportPublicDma

```
int wh_Client_KeyExportPublicDma(
    whClientContext * c,
    whKeyId keyId,
    uint16_t algo,
    void * keyAddr,
    uint16_t keySz,
    uint8_t * label,
    uint16_t labelSz,
    uint16_t * outSz
)
```

Blocking wrapper around the public-key export DMA request/response.

Parameters:

- **c** Pointer to the client context.

- **keyId** Server key ID whose public key should be exported.
- **algo** Algorithm selector (WH_KEY_ALGO_*).
- **keyAddr** Client buffer that will receive the public DER.
- **keySz** Size of the keyAddr buffer in bytes.
- **label** Optional buffer to receive the cached key's label.
- **labelSz** Size of label buffer.
- **outSz** Receives the number of public-DER bytes written.

Return: int Returns 0 on success, or a negative error code on failure.

A.1.4.53 function wh_Client_KeyWrap

```
int wh_Client_KeyWrap(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    uint16_t serverKeyId,
    void * keyIn,
    uint16_t keySz,
    whNvmMetadata * metadataIn,
    void * wrappedKeyOut,
    uint16_t * wrappedKeyInOutSz
)
```

Sends a key wrap request to the server and receives the response.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used to wrap the key.
- **serverKeyId** Key ID of the key encryption key on the server.
- **keyIn** Pointer to the key material to wrap.
- **keySz** The size in bytes of the key material to wrap.
- **metadataIn** Pointer to the metadata for the wrapped key.
- **wrappedKeyOut** Pointer to store the wrapped key.
- **[in/out]** wrappedKeyInOutSz IN: Size of wrappedKeyOut in bytes. OUT: Size of the total wrapped key object returned by the server. OUT may be less than IN.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a key wrap request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response, extracting the wrapped key from the response data once received. This function will block until the entire operation completes or an error occurs.

A.1.4.54 function wh_Client_KeyWrapRequest

```
int wh_Client_KeyWrapRequest(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    uint16_t serverKeyId,
    void * key,
    uint16_t keySz,
    whNvmMetadata * metadata
)
```

Sends a key wrap request to the server.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used to wrap the key.
- **serverKeyId** Key ID of the key encryption key on the server.
- **key** Pointer to the key material to wrap.
- **keySz** The size in bytes of the key material to wrap.
- **metadataIn** Pointer to the metadata for the wrapped key.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a key wrap request to the server. The request data contains the key data and metadata to be wrapped. This function does not block; it returns immediately after sending the request.

A.1.4.55 function wh_Client_KeyWrapResponse

```
int wh_Client_KeyWrapResponse(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    void * wrappedKeyOut,
    uint16_t * wrappedKeyInOutSz
)
```

Receives a key wrap response from the server.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used to wrap the key.
- **wrappedKeyOut** Pointer to store the wrapped key.
- **[in/out]** wrappedKeyInOutSz IN: Size of the wrappedKeyOut buffer. OUT: Size of the wrapped key object. OUT may be less than IN

Return: int Returns 0 on success, or a negative error code on failure.

This function attempts to process a key wrap response message from the server. It will validate the response and extract the wrapped key from the response data. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.56 function wh_Client_KeyUnwrapAndExport

```
int wh_Client_KeyUnwrapAndExport(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    uint16_t serverKeyId,
    void * wrappedKeyIn,
    uint16_t wrappedKeySz,
    whNvmMetadata * metadataOut,
    void * keyOut,
    uint16_t * keyInOutSz
)
```

Requests the server to unwrap and export a wrapped key and receives the response.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when for unwrapping the key.
- **serverKeyId** Key ID to be used for unwrapping the key.
- **wrappedKeyIn** Pointer to the wrapped key data.

- **wrappedKeySz** The size in bytes of the wrapped key data.
- **metadataOut** Pointer to store the unwrapped key metadata.
- **keyOut** Pointer to store the unwrapped key.
- **[in/out]** keyInOutSz IN: Size of the keyOut buffer. OUT: Size of the exported key returned by the server. OUT may be less than IN.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a unwrap key and export request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response, extracting the unwrapped key and metadata from the response data once received. This function will block until the entire operation completes or an error occurs.

A.1.4.57 function wh_Client_KeyUnwrapAndExportRequest

```
int wh_Client_KeyUnwrapAndExportRequest(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    uint16_t serverKeyId,
    void * wrappedKeyIn,
    uint16_t wrappedKeySz
)
```

Requests the server to unwrap-and-export a wrapped key.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when for unwrapping the key.
- **serverKeyId** Key ID to be used for unwrapping the key.
- **wrappedKeyIn** Pointer to the wrapped key data.
- **wrappedKeySz** The size in bytes of the wrapped key data.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a key unwrap-and-export request to the server. The request data contains the wrapped key for the server to unwrap. This function does not block; it returns immediately after sending the request.

A.1.4.58 function wh_Client_KeyUnwrapAndExportResponse

```
int wh_Client_KeyUnwrapAndExportResponse(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    whNvmMetadata * metadataOut,
    void * keyOut,
    uint16_t * keyInOutSz
)
```

Receives an unwrap-and-export response from the server.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when for unwrapping the key.
- **metadataOut** Pointer to store the unwrapped key metadata.
- **keyOut** Pointer to store the unwrapped key.
- **[in/out]** keyInOutSz IN: Size of the keyOut buffer. OUT: Size of the exported key returned by the server. OUT may be less than IN.

Return: int Returns 0 on success, or a negative error code on failure.

This function attempts to process an unwrap-and-export response message from the server. It will validate the response and extract the metadata and unwrapped key from from the response data. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.59 function wh_Client_KeyUnwrapAndCache

```
int wh_Client_KeyUnwrapAndCache(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    uint16_t serverKeyId,
    void * wrappedKeyIn,
    uint16_t wrappedKeySz,
    uint16_t * keyIdOut
)
```

Requests the server to unwrap and cache a wrapped key and receives the response.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when unwrapping the key.
- **serverKeyId** Key ID to be used for unwrapping the key.
- **wrappedKeyIn** Pointer to the wrapped key data.
- **wrappedKeySz** The size in bytes of the wrapped key data.
- **keyIdOut** Pointer to store the server-assigned ID of the cached key.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a unwrap-and-cache request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response, extracting the server-assigned key ID for the unwrapped key once received. This function will block until the entire operation completes or an error occurs.

A.1.4.60 function wh_Client_KeyUnwrapAndCacheRequest

```
int wh_Client_KeyUnwrapAndCacheRequest(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    uint16_t serverKeyId,
    void * wrappedKeyIn,
    uint16_t wrappedKeySz
)
```

Sends a key unwrap-and-cache request to the server.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when unwrapping the key.
- **serverKeyId** Key ID to be used for unwrapping the key.
- **wrappedKeyIn** Pointer to the wrapped key data.
- **wrappedKeySz** The size in bytes of the wrapped key data.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a key unwrap-and-cache request to the server. The request data contains the wrapped key for the server to unwrap and cache. This function does not block; it returns immediately after sending the request.

A.1.4.61 function wh_Client_KeyUnwrapAndCacheResponse

```
int wh_Client_KeyUnwrapAndCacheResponse(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    uint16_t * keyIdOut
)
```

Receives an unwrap-and-cache response from the server.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when unwrapping the key.
- **keyIdOut** Pointer to store the server-assigned ID of the cached key.

Return: int Returns 0 on success, or a negative error code on failure.

This function attempts to process an unwrap-and-cache response message from the server. It will validate the response and extract the server-assigned key ID for the cached key. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.62 function wh_Client_DataWrap

```
int wh_Client_DataWrap(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    uint16_t serverKeyId,
    void * dataIn,
    uint32_t dataInSz,
    void * wrappedDataOut,
    uint32_t * wrappedDataInOutSz
)
```

Helper function to wrap a data object using a specified key.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when wrapping the data.
- **serverKeyId** Key ID to be used for wrapping the data.
- **dataIn** Pointer to the plaintext data you want to wrap.
- **dataInSz** The size in bytes of the plaintext data.
- **wrappedDataOut** The pointer to the buffer that stores the resulting wrapped data.
- **[in/out]** wrappedDataInOutSz IN: The size in bytes of wrappedDataOut buffer. OUT: The size of the wrapped data object returned from the server. OUT may be less than IN.

Return: int Returns 0 on success, or a negative error code on failure.

This helper function uses existing calls in wolfHSM and wolfCrypt to construct a wrapped data object using a specified cipher and key id

A.1.4.63 function wh_Client_DataWrapRequest

```
int wh_Client_DataWrapRequest(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    uint16_t serverKeyId,
    void * dataIn,
```

```
    uint32_t dataInSz
)
```

Sends a data wrap request to the server.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when wrapping the data.
- **serverKeyId** Key ID to be used for wrapping the data.
- **dataIn** Pointer to the plaintext data you want to wrap.
- **dataInSz** The size in bytes of the plaintext data.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a data wrap request to the server. The request data contains the plaintext data for the server to wrap. This function does not block; it returns immediately after sending the request.

A.1.4.64 function wh_Client_DataWrapResponse

```
int wh_Client_DataWrapResponse(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    void * wrappedDataOut,
    uint32_t * wrappedDataSz
)
```

Receives a data wrap response from the server.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when wrapping the data.
- **wrappedDataOut** The pointer to the buffer that stores the resulting wrapped data.
- **[in/out]** wrappedDataSz IN: The size in bytes of wrappedDataOut buffer. OUT: The size of the wrapped data object returned from the server. OUT may be less than IN.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a data wrap response message from the server. It will validate the response and extract the wrapped data from the response data. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.65 function wh_Client_DataUnwrap

```
int wh_Client_DataUnwrap(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    uint16_t serverKeyId,
    void * wrappedDataIn,
    uint32_t wrappedDataInSz,
    void * dataOut,
    uint32_t * dataInOutSz
)
```

Helper function to unwrap a wrapped data object using a specified key.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when unwrapping the data.
- **serverKeyId** Key ID to be used for wrapping the data.
- **wrappedDataIn** Pointer to the wrapped data object you want to unwrap.
- **wrappedDataInSz** The size in bytes of the wrapped data object.
- **dataOut** The pointer to the buffer that stores the resulting unwrapped data.
- **[in/out]** dataInOutSz IN: The size in bytes of dataOut. OUT: The size of the unwrapped data object return by the server. OUT may be less than IN.

Return: int Returns 0 on success, or a negative error code on failure.

This helper function uses existing calls in wolfHSM and wolfCrypt to unwrap a wrapped data object using a specified cipher and key id

A.1.4.66 function wh_Client_DataUnwrapRequest

```
int wh_Client_DataUnwrapRequest(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    uint16_t serverKeyId,
    void * wrappedDataIn,
    uint32_t wrappedDataInSz
)
```

Sends a data unwrap request to the server.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when unwrapping the data.
- **serverKeyId** Key ID to be used for unwrapping the data.
- **wrappedDataIn** Pointer to the wrapped data object you want to unwrap.
- **wrappedDataInSz** The size in bytes of the wrapped data object.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a data unwrap request to the server. The request data contains the wrapped data object for the server to unwrap. This function does not block; it returns immediately after sending the request.

A.1.4.67 function wh_Client_DataUnwrapResponse

```
int wh_Client_DataUnwrapResponse(
    whClientContext * ctx,
    enum wc_CipherType cipherType,
    void * dataOut,
    uint32_t * dataSz
)
```

Receives a data unwrap response from the server.

Parameters:

- **ctx** Pointer to the client context.
- **cipherType** Cipher used when unwrapping the data.
- **dataOut** The pointer to the buffer that stores the resulting unwrapped data.
- **[in/out]** dataSz IN: The size in bytes of dataOut. OUT: The size of the unwrapped data object returned by the server. OUT may be less than IN.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a data unwrap response message from the server. It will validate the response and extract the unwrapped data from the response data. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.68 function wh_Client_CounterInitRequest

```
int wh_Client_CounterInitRequest(
    whClientContext * c,
    whNvmId counterId,
    uint32_t counter
)
```

A.1.4.69 function wh_Client_CounterInitResponse

```
int wh_Client_CounterInitResponse(
    whClientContext * c,
    uint32_t * counter
)
```

A.1.4.70 function wh_Client_CounterInit

```
int wh_Client_CounterInit(
    whClientContext * c,
    whNvmId counterId,
    uint32_t * counter
)
```

Creates and initializes a counter with the value set in counter.

Parameters:

- **c** Pointer to the whClientContext structure.
- **counterId** counter ID to be associated with the counter.
- **counter** Value to initialize the counter with, returns with the value set by the HSM for confirmation.

Return: int Returns 0 on success or a negative error code on failure.

This function creates/resets a counter with the supplied counterId and gives it the value stored in counter at the start of the call.

A.1.4.71 function wh_Client_CounterResetRequest

```
int wh_Client_CounterResetRequest(
    whClientContext * c,
    whNvmId counterId
)
```

A.1.4.72 function wh_Client_CounterResetResponse

```
int wh_Client_CounterResetResponse(
    whClientContext * c,
    uint32_t * counter
)
```

A.1.4.73 function wh_Client_CounterReset

```
int wh_Client_CounterReset(
    whClientContext * c,
    whNvmId counterId,
    uint32_t * counter
)
```

Creates and initializes a counter with to 0.

Parameters:

- **c** Pointer to the whClientContext structure.
- **counterId** Counter ID to be associated with the counter.
- **counter** Value set by the HSM for confirmation.

Return: int Returns 0 on success or a negative error code on failure.

This function creates/resets a counter with the supplied counterId and gives it the value of 0.

A.1.4.74 function wh_Client_CounterIncrementRequest

```
int wh_Client_CounterIncrementRequest(
    whClientContext * c,
    whNvmId counterId
)
```

A.1.4.75 function wh_Client_CounterIncrementResponse

```
int wh_Client_CounterIncrementResponse(
    whClientContext * c,
    uint32_t * counter
)
```

A.1.4.76 function wh_Client_CounterIncrement

```
int wh_Client_CounterIncrement(
    whClientContext * c,
    whNvmId counterId,
    uint32_t * counter
)
```

Increments a counter.

Parameters:

- **c** Pointer to the whClientContext structure.
- **counterId** Counter ID to be associated with the counter.
- **counter** Value set by the HSM for confirmation.

Return: int Returns 0 on success or a negative error code on failure.

This function increments a counter created previously. If the counter would roll over the HSM will saturate the value, keeping it at the uint32_t max.

A.1.4.77 function wh_Client_CounterReadRequest

```
int wh_Client_CounterReadRequest(
    whClientContext * c,
```

```
    whNvmId counterId
)
```

A.1.4.78 function wh_Client_CounterReadResponse

```
int wh_Client_CounterReadResponse(
    whClientContext * c,
    uint32_t * counter
)
```

A.1.4.79 function wh_Client_CounterRead

```
int wh_Client_CounterRead(
    whClientContext * c,
    whNvmId counterId,
    uint32_t * counter
)
```

Read a counter.

Parameters:

- **c** Pointer to the whClientContext structure.
- **counterId** Counter ID to be associated with the counter.
- **counter** Value set by the HSM.

Return: int Returns 0 on success or a negative error code on failure.

This function read a counter created previously.

A.1.4.80 function wh_Client_CounterDestroyRequest

```
int wh_Client_CounterDestroyRequest(
    whClientContext * c,
    whNvmId counterId
)
```

A.1.4.81 function wh_Client_CounterDestroyResponse

```
int wh_Client_CounterDestroyResponse(
    whClientContext * c
)
```

A.1.4.82 function wh_Client_CounterDestroy

```
int wh_Client_CounterDestroy(
    whClientContext * c,
    whNvmId counterId
)
```

Destroy a counter.

Parameters:

- **c** Pointer to the whClientContext structure.
- **counterId** Counter ID to be associated with the counter.

Return: int Returns 0 on success or a negative error code on failure.

This function destroys an NVM counter created previously.

A.1.4.83 function wh_Client_NvmInitRequest

```
int wh_Client_NvmInitRequest(
    whClientContext * c
)
```

Sends a non-volatile memory (NVM) initialization request to the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

NVM functions

This function prepares and sends an NVM initialization request message to the server. The message contains the client NVM ID. This function does not block; it returns immediately after sending the request.

A.1.4.84 function wh_Client_NvmInitResponse

```
int wh_Client_NvmInitResponse(
    whClientContext * c,
    int32_t * out_rc,
    uint32_t * out_clientnvm_id,
    uint32_t * out_servernvm_id
)
```

Receives a non-volatile memory (NVM) initialization response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_clientnvm_id** Pointer to store the client NVM ID assigned by the server.
- **out_servernvm_id** Pointer to store the server NVM ID assigned by the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process an NVM initialization response message from the server. It validates the response and extracts the client and server NVM IDs. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.85 function wh_Client_NvmInit

```
int wh_Client_NvmInit(
    whClientContext * c,
    int32_t * out_rc,
    uint32_t * out_clientnvm_id,
    uint32_t * out_servernvm_id
)
```

Sends a non-volatile memory (NVM) initialization request to the server and receives the response.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_clientnvm_id** Pointer to store the client NVM ID assigned by the server.
- **out_servernvm_id** Pointer to store the server NVM ID assigned by the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending an NVM initialization request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response, extracting the client and server NVM IDs from the response once received. This function blocks until the entire operation is complete or an error occurs.

A.1.4.86 function wh_Client_NvmCleanupRequest

```
int wh_Client_NvmCleanupRequest(
    whClientContext * c
)
```

Sends a non-volatile memory (NVM) cleanup request to the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends an NVM cleanup request message to the server. This function does not block; it returns immediately after sending the request.

A.1.4.87 function wh_Client_NvmCleanupResponse

```
int wh_Client_NvmCleanupResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a non-volatile memory (NVM) cleanup response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process an NVM cleanup response message from the server. It validates the response. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.88 function wh_Client_NvmCleanup

```
int wh_Client_NvmCleanup(
    whClientContext * c,
    int32_t * out_rc
)
```

Sends a non-volatile memory (NVM) cleanup request to the server and receives the response.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending an NVM cleanup request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.89 function wh_Client_NvmGetAvailableRequest

```
int wh_Client_NvmGetAvailableRequest(
    whClientContext * c
)
```

Sends a request to the server to get available non-volatile memory (NVM) information.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to the server to retrieve information about the available and reclaimable NVM space and objects. This function does not block; it returns immediately after sending the request.

A.1.4.90 function wh_Client_NvmGetAvailableResponse

```
int wh_Client_NvmGetAvailableResponse(
    whClientContext * c,
    int32_t * out_rc,
    uint32_t * out_avail_size,
    whNvmId * out_avail_objects,
    uint32_t * out_reclaim_size,
    whNvmId * out_reclaim_objects
)
```

Receives a response from the server with available non-volatile memory (NVM) information.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_avail_size** Pointer to store the available NVM size.
- **out_avail_objects** Pointer to store the available NVM objects.
- **out_reclaim_size** Pointer to store the reclaimable NVM size.
- **out_reclaim_objects** Pointer to store the reclaimable NVM objects.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a response message from the server containing information about the available and reclaimable NVM space and objects. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.91 function wh_Client_NvmGetAvailable

```
int wh_Client_NvmGetAvailable(
    whClientContext * c,
    int32_t * out_rc,
    uint32_t * out_avail_size,
    whNvmId * out_avail_objects,
    uint32_t * out_reclaim_size,
```

```
    whNvmId * out_reclaim_objects
)
```

Sends a request to the server and receives a response with available non-volatile memory (NVM) information.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_avail_size** Pointer to store the available NVM size.
- **out_avail_objects** Pointer to store the available NVM objects.
- **out_reclaim_size** Pointer to store the reclaimable NVM size.
- **out_reclaim_objects** Pointer to store the reclaimable NVM objects.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to the server to retrieve information about the available and reclaimable NVM space and objects, and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.92 function wh_Client_NvmAddObjectRequest

```
int wh_Client_NvmAddObjectRequest(
    whClientContext * c,
    whNvmId id,
    whNvmAccess access,
    whNvmFlags flags,
    whNvmSize label_len,
    uint8_t * label,
    whNvmSize len,
    const uint8_t * data
)
```

Sends a request to the server to add an object to non-volatile memory (NVM).

Parameters:

- **c** Pointer to the client context.
- **id** The ID of the NVM object to add.
- **access** The access permissions for the NVM object.
- **flags** Flags associated with the NVM object.
- **label_len** The length of the label.
- **label** Pointer to the label data.
- **len** The length of the data.
- **data** Pointer to the data to be added.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to the server to add an object to the NVM. The request includes the object ID, access permissions, flags, label, and data. This function does not block; it returns immediately after sending the request.

A.1.4.93 function wh_Client_NvmAddObjectResponse

```
int wh_Client_NvmAddObjectResponse(
    whClientContext * c,
```

```
    int32_t * out_rc
)
```

Receives a response from the server after attempting to add an object to non-volatile memory (NVM).

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a response message from the server after an add object request. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.94 function wh_Client_NvmAddObject

```
int wh_Client_NvmAddObject(
    whClientContext * c,
    whNvmId id,
    whNvmAccess access,
    whNvmFlags flags,
    whNvmSize label_len,
    uint8_t * label,
    whNvmSize len,
    const uint8_t * data,
    int32_t * out_rc
)
```

Sends a request to the server and receives a response to add an object to non-volatile memory (NVM).

Parameters:

- **c** Pointer to the client context.
- **id** The ID of the NVM object to add.
- **access** The access permissions for the NVM object.
- **flags** Flags associated with the NVM object.
- **label_len** The length of the label.
- **label** Pointer to the label data.
- **len** The length of the data.
- **data** Pointer to the data to be added.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to the server to add an object to the NVM and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.95 function wh_Client_NvmListRequest

```
int wh_Client_NvmListRequest(
    whClientContext * c,
    whNvmAccess access,
    whNvmFlags flags,
    whNvmId start_id
)
```

Sends a request to the server to list non-volatile memory (NVM) objects.

Parameters:

- **c** Pointer to the client context.
- **access** The access permissions for the NVM objects to list.
- **flags** Flags associated with the NVM objects to list.
- **start_id** The starting ID of the NVM objects to list.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to the server to list NVM objects. The request includes the access permissions, flags, and the starting object ID. This function does not block; it returns immediately after sending the request.

A.1.4.96 function wh_Client_NvmListResponse

```
int wh_Client_NvmListResponse(
    whClientContext * c,
    int32_t * out_rc,
    whNvmId * out_count,
    whNvmId * out_id
)
```

Receives a response from the server with a list of non-volatile memory (NVM) objects.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_count** Pointer to store the count of NVM objects that match the criteria.
- **out_id** Pointer to store the ID of the first matching NVM object.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a response message from the server containing a list of NVM objects. It validates the response and extracts the return code, count of objects, and the object IDs. The count is the number of objects that match the flags/access pattern starting at start_id. The out_id is the first matching object ID. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.97 function wh_Client_NvmList

```
int wh_Client_NvmList(
    whClientContext * c,
    whNvmAccess access,
    whNvmFlags flags,
    whNvmId start_id,
    int32_t * out_rc,
    whNvmId * out_count,
    whNvmId * out_id
)
```

Sends a request to the server and receives a response to list non-volatile memory (NVM) objects.

Parameters:

- **c** Pointer to the client context.
- **access** The access permissions for the NVM objects to list.

- **flags** Flags associated with the NVM objects to list.
- **start_id** The starting ID of the NVM objects to list.
- **out_rc** Pointer to store the return code from the server.
- **out_count** Pointer to store the count of NVM objects that match the criteria.
- **out_id** Pointer to store the ID of the first matching NVM object.

Return: int Returns 0 on success, or a negative error code on failure.

Note: Enumerating all objects requires calling this function in a loop with successive start_id values. In thread-safe builds, concurrent NVM modifications between calls may result in inconsistent results.

This function handles the complete process of sending a request to the server to list NVM objects and receiving the response. It sends the request and repeatedly attempts to receive a valid response. The count is the number of objects that match the flags/access pattern starting at start_id. The out_id is the first matching object ID. This function blocks until the entire operation is complete or an error occurs.

A.1.4.98 function wh_Client_NvmGetMetadataRequest

```
int wh_Client_NvmGetMetadataRequest(
    whClientContext * c,
    whNvmId id
)
```

Sends a request to the server to get metadata of a non-volatile memory (NVM) object.

Parameters:

- **c** Pointer to the client context.
- **id** The ID of the NVM object for which metadata is requested.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to the server to retrieve metadata for a specific NVM object. The request includes the object ID. This function does not block; it returns immediately after sending the request.

A.1.4.99 function wh_Client_NvmGetMetadataResponse

```
int wh_Client_NvmGetMetadataResponse(
    whClientContext * c,
    int32_t * out_rc,
    whNvmId * out_id,
    whNvmAccess * out_access,
    whNvmFlags * out_flags,
    whNvmSize * out_len,
    whNvmSize label_len,
    uint8_t * label
)
```

Receives a response from the server with metadata of a non-volatile memory (NVM) object.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_id** Pointer to store the ID of the NVM object.
- **out_access** Pointer to store the access permissions of the NVM object.
- **out_flags** Pointer to store the flags of the NVM object.
- **out_len** Pointer to store the length of the data.

- **label_len** The length of the label buffer.
- **label** Pointer to store the label data.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a response message from the server containing metadata of an NVM object. It validates the response and extracts the return code, object ID, access permissions, flags, data length, and label. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.100 function wh_Client_NvmGetMetadata

```
int wh_Client_NvmGetMetadata(
    whClientContext * c,
    whNvmId id,
    int32_t * out_rc,
    whNvmId * out_id,
    whNvmAccess * out_access,
    whNvmFlags * out_flags,
    whNvmSize * out_len,
    whNvmSize label_len,
    uint8_t * label
)
```

Sends a request to the server and receives a response to get metadata of a non-volatile memory (NVM) object.

Parameters:

- **c** Pointer to the client context.
- **id** The ID of the NVM object for which metadata is requested.
- **out_rc** Pointer to store the return code from the server.
- **out_id** Pointer to store the ID of the NVM object.
- **out_access** Pointer to store the access permissions of the NVM object.
- **out_flags** Pointer to store the flags of the NVM object.
- **out_len** Pointer to store the length of the data.
- **label_len** The length of the label buffer.
- **label** Pointer to store the label data.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to the server to get metadata of an NVM object and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.101 function wh_Client_NvmDestroyObjectsRequest

```
int wh_Client_NvmDestroyObjectsRequest(
    whClientContext * c,
    whNvmId list_count,
    const whNvmId * id_list
)
```

Sends a request to the server to destroy non-volatile memory (NVM) objects.

Parameters:

- **c** Pointer to the client context.

- **list_count** The number of NVM objects to destroy.
- **id_list** Pointer to an array of IDs of the NVM objects to destroy.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to the server to destroy a list of NVM objects. The request includes the count of objects and their IDs. This function does not block; it returns immediately after sending the request.

A.1.4.102 function wh_Client_NvmDestroyObjectsResponse

```
int wh_Client_NvmDestroyObjectsResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after attempting to destroy non-volatile memory (NVM) objects.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a response message from the server after attempting to destroy NVM objects. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.103 function wh_Client_NvmDestroyObjects

```
int wh_Client_NvmDestroyObjects(
    whClientContext * c,
    whNvmId list_count,
    const whNvmId * id_list,
    int32_t * out_rc
)
```

Sends a request to the server and receives a response to destroy non-volatile memory (NVM) objects.

Parameters:

- **c** Pointer to the client context.
- **list_count** The number of NVM objects to destroy.
- **id_list** Pointer to an array of IDs of the NVM objects to destroy.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to the server to destroy NVM objects and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.104 function wh_Client_NvmReadRequest

```
int wh_Client_NvmReadRequest(
    whClientContext * c,
    whNvmId id,
    whNvmSize offset,
```

```
    whNvmSize data_len
)
```

Sends a request to the server to read data from a non-volatile memory (NVM) object.

Parameters:

- **c** Pointer to the client context.
- **id** The ID of the NVM object to read from.
- **offset** The offset within the NVM object data to start reading from.
- **data_len** The length of data to read.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to the server to read data from a specific NVM object. The request includes the object ID, the offset within the NVM object data to start reading from, and the length of data to read. This function does not block; it returns immediately after sending the request.

A.1.4.105 function wh_Client_NvmReadResponse

```
int wh_Client_NvmReadResponse(
    whClientContext * c,
    int32_t * out_rc,
    whNvmSize * out_len,
    uint8_t * data
)
```

Receives a response from the server with NVM object data.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_len** Pointer to store the length of the data read.
- **data** Pointer to store the NVM object data.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a response message from the server containing NVM object data. It validates the response and extracts the return code, the length of the data read, and the data itself. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.106 function wh_Client_NvmRead

```
int wh_Client_NvmRead(
    whClientContext * c,
    whNvmId id,
    whNvmSize offset,
    whNvmSize data_len,
    int32_t * out_rc,
    whNvmSize * out_len,
    uint8_t * data
)
```

Sends a request to the server and receives a response to read data from a non-volatile memory (NVM) object.

Parameters:

- **c** Pointer to the client context.

- **id** The ID of the NVM object to read from.
- **offset** The offset within the NVM object data to start reading from.
- **data_len** The length of data to read.
- **out_rc** Pointer to store the return code from the server.
- **out_len** Pointer to store the length of the data read.
- **data** Pointer to store the NVM object data.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to the server to read data from an NVM object and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.107 function wh_Client_NvmAddObjectDmaRequest

```
int wh_Client_NvmAddObjectDmaRequest(
    whClientContext * c,
    whNvmMetadata * metadata,
    whNvmSize data_len,
    const uint8_t * data
)
```

Sends a request to the server to add an object to non-volatile memory (NVM) using DMA.

Parameters:

- **c** Pointer to the client context.
- **metadata** Pointer to the metadata.
- **data_len** The length of the data to be added.
- **data** Pointer to the data to be added.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to the server to add an object to NVM using DMA. The request includes the metadata client address, the length of the data, and the data client address. This function does not block; it returns immediately after sending the request.

A.1.4.108 function wh_Client_NvmAddObjectDmaResponse

```
int wh_Client_NvmAddObjectDmaResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after attempting to add an object to non-volatile memory (NVM) using DMA.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a response message from the server after attempting to add an object to NVM using DMA. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.109 function wh_Client_NvmAddObjectDma

```
int wh_Client_NvmAddObjectDma(
    whClientContext * c,
    whNvmMetadata * metadata,
    whNvmSize data_len,
    const uint8_t * data,
    int32_t * out_rc
)
```

Sends a request to the server and receives a response to add an object to non-volatile memory (NVM) using DMA.

Parameters:

- **c** Pointer to the client context.
- **metadata** Pointer to the metadata.
- **data_len** The length of the data to be added.
- **data** Pointer to the data to be added.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to the server to add an object to NVM using DMA and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.110 function wh_Client_NvmReadDmaRequest

```
int wh_Client_NvmReadDmaRequest(
    whClientContext * c,
    whNvmId id,
    whNvmSize offset,
    whNvmSize data_len,
    uint8_t * data
)
```

A.1.4.111 function wh_Client_NvmReadDmaResponse

```
int wh_Client_NvmReadDmaResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after attempting to read data from non-volatile memory (NVM) using DMA, with automatic detection of client address width (32-bit or 64-bit).

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a response message from the server after attempting to read data from NVM using DMA. The client address width (32-bit or 64-bit) is automatically detected. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.112 function wh_Client_NvmReadDma

```
int wh_Client_NvmReadDma(
    whClientContext * c,
    whNvmId id,
    whNvmSize offset,
    whNvmSize data_len,
    uint8_t * data,
    int32_t * out_rc
)
```

Sends a request to the server and receives a response to read data from non-volatile memory (NVM) using DMA, with automatic detection of client address width (32-bit or 64-bit).

Parameters:

- **c** Pointer to the client context.
- **id** The NVM ID of the object to read.
- **offset** The offset within the object to start reading from.
- **data_len** The length of the data to be read.
- **data** Pointer to the data buffer where the data will be read into.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to the server to read data from NVM using DMA and receiving the response. The client address width (32-bit or 64-bit) is automatically detected. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.113 function wh_Client_CustomCbRequest

```
int wh_Client_CustomCbRequest(
    whClientContext * c,
    const whMessageCustomCb_Request * req
)
```

Sends a custom callback request to the server.

Parameters:

- **c** Pointer to the client context.
- **req** Pointer to the custom callback request structure.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a custom callback request to the server. The request includes the custom callback request structure. This function does not block; it returns immediately after sending the request.

A.1.4.114 function wh_Client_CustomCbResponse

```
int wh_Client_CustomCbResponse(
    whClientContext * c,
    whMessageCustomCb_Response * resp
)
```

Receives a response from the server after sending a custom callback request.

Parameters:

- **c** Pointer to the client context.
- **resp** Pointer to store the custom callback response from the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a response message from the server after sending a custom callback request. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.115 function wh_Client_CustomCheckRegisteredRequest

```
int wh_Client_CustomCheckRegisteredRequest(
    whClientContext * c,
    uint32_t id
)
```

Sends a request to the server to check if a custom callback is registered.

Parameters:

- **c** Pointer to the client context.
- **id** The ID of the custom callback to check.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to the server to check if a custom callback is registered. The request includes the callback ID. This function does not block; it returns immediately after sending the request.

A.1.4.116 function wh_Client_CustomCbCheckRegisteredResponse

```
int wh_Client_CustomCbCheckRegisteredResponse(
    whClientContext * c,
    uint16_t * outId,
    int * responseError
)
```

Receives a response from the server after checking if a custom callback is registered.

Parameters:

- **c** Pointer to the client context.
- **outId** Pointer to store the callback ID from the server.
- **responseError** Pointer to store the response error code from the server.

Return: int Returns 0 if the callback is registered, WH_ERROR_NOHANDLER if it is not registered, or a negative error code on failure.

This function attempts to process a response message from the server after checking if a custom callback is registered. It validates the response and extracts the return code and callback ID. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.117 function wh_Client_CustomCbCheckRegistered

```
int wh_Client_CustomCbCheckRegistered(
    whClientContext * c,
    uint16_t id,
    int * responseError
)
```

Sends a request to the server and receives a response to check if a custom callback is registered.

Parameters:

- **c** Pointer to the client context.
- **id** The ID of the custom callback to check.
- **responseError** Pointer to store the response error code from the server.

Return: int Returns 0 if the callback is registered, WH_ERROR_NOHANDLER if it is not registered, or a negative error code on failure.

This function handles the complete process of sending a request to the server to check if a custom callback is registered and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.118 function wh_Client_AuthLoginRequest

```
int wh_Client_AuthLoginRequest(
    whClientContext * c,
    whAuthMethod method,
    const char * username,
    const void * auth_data,
    uint16_t auth_data_len
)
```

Sends an authentication request to the server.

Parameters:

- **c** Pointer to the client context.
- **method** The authentication method to use (e.g., WH_AUTH_METHOD_PIN).
- **username** The user name to login (null-terminated C string).
- **auth_data** Pointer to the authentication data.
- **auth_data_len** Length of the authentication data.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends an authentication request message to the server. The request includes the authentication method and authentication data (e.g., PIN). This function does not block; it returns immediately after sending the request.

A.1.4.119 function wh_Client_AuthLoginResponse

```
int wh_Client_AuthLoginResponse(
    whClientContext * c,
    int32_t * out_rc,
    whUserId * out_user_id
)
```

Receives an authentication response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the authenticated user ID.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process an authentication response message from the server. It validates the response and extracts the return code and user ID. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.120 function wh_Client_AuthLogin

```
int wh_Client_AuthLogin(
    whClientContext * c,
    whAuthMethod method,
    const char * username,
    const void * auth_data,
    uint16_t auth_data_len,
    int32_t * out_rc,
    whUserId * out_user_id
)
```

Authenticates a user with the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **method** The authentication method to use (e.g., WH_AUTH_METHOD_PIN).
- **username** The user name to login (null-terminated C string).
- **auth_data** Pointer to the authentication data.
- **auth_data_len** Length of the authentication data.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the authenticated user ID.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending an authentication request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.121 function wh_Client_AuthLogoutRequest

```
int wh_Client_AuthLogoutRequest(
    whClientContext * c,
    whUserId user_id
)
```

Sends a logout request to the server.

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to logout.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a logout request message to the server. This function does not block; it returns immediately after sending the request.

A.1.4.122 function wh_Client_AuthLogoutResponse

```
int wh_Client_AuthLogoutResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a logout response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a logout response message from the server. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.123 function wh_Client_AuthLogout

```
int wh_Client_AuthLogout(
    whClientContext * c,
    whUserId user_id,
    int32_t * out_rc
)
```

Logs out a user from the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to logout.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a logout request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.124 function wh_Client_AuthUserAddResponse

```
int wh_Client_AuthUserAddResponse(
    whClientContext * c,
    int32_t * out_rc,
    whUserId * out_user_id
)
```

Receives a user add response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the new user ID.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a user add response message from the server. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.125 function wh_Client_AuthUserAddRequest

```
int wh_Client_AuthUserAddRequest(
    whClientContext * c,
```

```
    const char * username,
    whAuthPermissions permissions,
    whAuthMethod method,
    const void * credentials,
    uint16_t credentials_len
)
```

Sends a user add request to the server.

Parameters:

- **c** Pointer to the client context.
- **username** The username for the new user.
- **permissions** The permissions for the new user.
- **method** The authentication method for the new user.
- **credentials** Pointer to the credentials data.
- **credentials_len** Length of the credentials data.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a user add request message to the server. This function does not block; it returns immediately after sending the request.

A.1.4.126 function wh_Client_AuthUserAdd

```
int wh_Client_AuthUserAdd(
    whClientContext * c,
    const char * username,
    whAuthPermissions permissions,
    whAuthMethod method,
    const void * credentials,
    uint16_t credentials_len,
    int32_t * out_rc,
    whUserId * out_user_id
)
```

Adds a new user to the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **username** The username for the new user.
- **permissions** The permissions for the new user.
- **method** The authentication method for the new user.
- **credentials** Pointer to the credentials data.
- **credentials_len** Length of the credentials data.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the new user ID.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a user add request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.127 function wh_Client_AuthUserGetRequest

```
int wh_Client_AuthUserGetRequest(
    whClientContext * c,
```

```
    const char * username
)
```

Sends a user get request to the server.

Parameters:

- **c** Pointer to the client context.
- **username** The username to look up.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a user get request message to the server. This function does not block; it returns immediately after sending the request.

A.1.4.128 function wh_Client_AuthUserGetResponse

```
int wh_Client_AuthUserGetResponse(
    whClientContext * c,
    int32_t * out_rc,
    whUserId * out_user_id,
    whAuthPermissions * out_permissions
)
```

Receives a user get response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the user ID.
- **out_permissions** Pointer to store the user permissions.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a user get response message from the server. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.129 function wh_Client_AuthUserGet

```
int wh_Client_AuthUserGet(
    whClientContext * c,
    const char * username,
    int32_t * out_rc,
    whUserId * out_user_id,
    whAuthPermissions * out_permissions
)
```

Gets user information from the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **username** The username to look up.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the user ID.
- **out_permissions** Pointer to store the user permissions.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a user get request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.130 function wh_Client_AuthUserDeleteRequest

```
int wh_Client_AuthUserDeleteRequest(
    whClientContext * c,
    whUserId user_id
)
```

Sends a user delete request to the server.

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to delete.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a user delete request message to the server. This function does not block; it returns immediately after sending the request.

A.1.4.131 function wh_Client_AuthUserDeleteResponse

```
int wh_Client_AuthUserDeleteResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a user delete response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a user delete response message from the server. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.132 function wh_Client_AuthUserDelete

```
int wh_Client_AuthUserDelete(
    whClientContext * c,
    whUserId user_id,
    int32_t * out_rc
)
```

Deletes a user from the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to delete.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a user delete request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.133 function wh_Client_AuthUserSetPermissionsRequest

```
int wh_Client_AuthUserSetPermissionsRequest(
    whClientContext * c,
    whUserId user_id,
    whAuthPermissions permissions
)
```

Sends a user set permissions request to the server.

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to set permissions for.
- **permissions** The new permissions to set.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a user set permissions request message to the server. This function does not block; it returns immediately after sending the request.

A.1.4.134 function wh_Client_AuthUserSetPermissionsResponse

```
int wh_Client_AuthUserSetPermissionsResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a user set permissions response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a user set permissions response message from the server. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.135 function wh_Client_AuthUserSetPermissions

```
int wh_Client_AuthUserSetPermissions(
    whClientContext * c,
    whUserId user_id,
    whAuthPermissions permissions,
    int32_t * out_rc
)
```

Sets user permissions on the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.

- **user_id** The user ID to set permissions for.
- **permissions** The new permissions to set.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a user set permissions request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.136 function wh_Client_AuthUserSetCredentialsRequest

```
int wh_Client_AuthUserSetCredentialsRequest(
    whClientContext * c,
    whUserId user_id,
    whAuthMethod method,
    const void * current_credentials,
    uint16_t current_credentials_len,
    const void * new_credentials,
    uint16_t new_credentials_len
)
```

Sends a user set credentials request to the server.

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to set credentials for.
- **method** The authentication method.
- **current_credentials** Pointer to the current credentials data.
- **current_credentials_len** Length of the current credentials data.
- **new_credentials** Pointer to the new credentials data.
- **new_credentials_len** Length of the new credentials data.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a user set credentials request message to the server. This function does not block; it returns immediately after sending the request.

A.1.4.137 function wh_Client_AuthUserSetCredentialsResponse

```
int wh_Client_AuthUserSetCredentialsResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a user set credentials response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available, or a negative error code on failure.

This function attempts to process a user set credentials response message from the server. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.138 function wh_Client_AuthUserSetCredentials

```
int wh_Client_AuthUserSetCredentials(
    whClientContext * c,
    whUserId user_id,
    whAuthMethod method,
    const void * current_credentials,
    uint16_t current_credentials_len,
    const void * new_credentials,
    uint16_t new_credentials_len,
    int32_t * out_rc
)
```

Sets user credentials on the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to set credentials for.
- **method** The authentication method.
- **current_credentials** Pointer to the current credentials data.
- **current_credentials_len** Length of the current credentials data.
- **new_credentials** Pointer to the new credentials data.
- **new_credentials_len** Length of the new credentials data.
- **out_rc** Pointer to store the return code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a user set credentials request to the server and receiving the response. It sends the request and repeatedly attempts to receive a valid response. This function blocks until the entire operation is complete or an error occurs.

A.1.4.139 function wh_Client_CertInitRequest

```
int wh_Client_CertInitRequest(
    whClientContext * c
)
```

Sends a request to initialize the certificate manager on the server.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to initialize the certificate manager on the server. This function does not block; it returns immediately after sending the request.

A.1.4.140 function wh_Client_CertInitResponse

```
int wh_Client_CertInitResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after initializing the certificate manager.

Parameters:

- **c** Pointer to the client context.

- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function attempts to process a response message from the server after initializing the certificate manager. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.141 function wh_Client_CertInit

```
int wh_Client_CertInit(
    whClientContext * c,
    int32_t * out_rc
)
```

Sends a request and receives a response to initialize the certificate manager.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to initialize the certificate manager and receiving the response. It blocks until the entire operation is complete or an error occurs.

A.1.4.142 function wh_Client_CertAddTrustedRequest

```
int wh_Client_CertAddTrustedRequest(
    whClientContext * c,
    whNvmId id,
    whNvmAccess access,
    whNvmFlags flags,
    uint8_t * label,
    whNvmSize label_len,
    const uint8_t * cert,
    uint32_t cert_len
)
```

Sends a request to add a trusted certificate to NVM storage.

Parameters:

- **c** Pointer to the client context.
- **id** The NVM ID to store the certificate.
- **cert** Pointer to the certificate data.
- **cert_len** Length of the certificate data.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to add a trusted certificate to NVM storage. This function does not block; it returns immediately after sending the request.

A.1.4.143 function wh_Client_CertAddTrustedResponse

```
int wh_Client_CertAddTrustedResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after adding a trusted certificate.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function attempts to process a response message from the server after adding a trusted certificate. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.144 function wh_Client_CertAddTrusted

```
int wh_Client_CertAddTrusted(
    whClientContext * c,
    whNvmId id,
    whNvmAccess access,
    whNvmFlags flags,
    uint8_t * label,
    whNvmSize label_len,
    const uint8_t * cert,
    uint32_t cert_len,
    int32_t * out_rc
)
```

Sends a request and receives a response to add a trusted certificate.

Parameters:

- **c** Pointer to the client context.
- **id** The NVM ID to store the certificate.
- **cert** Pointer to the certificate data.
- **cert_len** Length of the certificate data.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to add a trusted certificate and receiving the response. It blocks until the entire operation is complete or an error occurs.

A.1.4.145 function wh_Client_CertEraseTrustedRequest

```
int wh_Client_CertEraseTrustedRequest(
    whClientContext * c,
    whNvmId id
)
```

Sends a request to erase a trusted certificate from NVM storage.

Parameters:

- **c** Pointer to the client context.
- **id** The NVM ID of the certificate to delete.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to erase a trusted certificate from NVM storage. This function does not block; it returns immediately after sending the request.

A.1.4.146 function wh_Client_CertEraseTrustedResponse

```
int wh_Client_CertEraseTrustedResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after erasing a trusted certificate.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function attempts to process a response message from the server after erasing a trusted certificate. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.147 function wh_Client_CertEraseTrusted

```
int wh_Client_CertEraseTrusted(
    whClientContext * c,
    whNvmId id,
    int32_t * out_rc
)
```

Sends a request and receives a response to erase a trusted certificate.

Parameters:

- **c** Pointer to the client context.
- **id** The NVM ID of the certificate to delete.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to erase a trusted certificate and receiving the response. It blocks until the entire operation is complete or an error occurs.

A.1.4.148 function wh_Client_CertReadTrustedRequest

```
int wh_Client_CertReadTrustedRequest(
    whClientContext * c,
    whNvmId id,
    uint32_t cert_len
)
```

Sends a request to read a trusted certificate from NVM storage.

Parameters:

- **c** Pointer to the client context.
- **id** The NVM ID of the certificate to retrieve.
- **cert_len** Maximum length of the certificate buffer.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to read a trusted certificate from NVM storage. This function does not block; it returns immediately after sending the request.

A.1.4.149 function wh_Client_CertReadTrustedResponse

```
int wh_Client_CertReadTrustedResponse(
    whClientContext * c,
    uint8_t * cert,
    uint32_t * cert_len,
    int32_t * out_rc
)
```

Receives a response from the server after getting a trusted certificate.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to store the certificate data.
- **cert_len** Pointer to the maximum length of the certificate buffer. On output, contains the actual length of the certificate.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function attempts to process a response message from the server after getting a trusted certificate. It validates the response, extracts the certificate data, and updates the certificate length. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.150 function wh_Client_CertReadTrusted

```
int wh_Client_CertReadTrusted(
    whClientContext * c,
    whNvmId id,
    uint8_t * cert,
    uint32_t * cert_len,
    int32_t * out_rc
)
```

Sends a request and receives a response to read a trusted certificate.

Parameters:

- **c** Pointer to the client context.
- **id** The NVM ID of the certificate to retrieve.
- **cert** Pointer to store the certificate data.
- **cert_len** Pointer to the maximum length of the certificate buffer. On output, contains the actual length of the certificate.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to read a trusted certificate and receiving the response. It blocks until the entire operation is complete or an error occurs.

A.1.4.151 function wh_Client_CertVerifyRequest

```
int wh_Client_CertVerifyRequest(
    whClientContext * c,
    const uint8_t * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId
)
```

Sends a request to verify a certificate against trusted certificates.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to verify a certificate against trusted certificates. This function does not block; it returns immediately after sending the request.

A.1.4.152 function wh_Client_CertVerifyResponse

```
int wh_Client_CertVerifyResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after verifying a certificate.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function attempts to process a response message from the server after verifying a certificate. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.153 function wh_Client_CertVerify

```
int wh_Client_CertVerify(
    whClientContext * c,
    const uint8_t * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId,
    int32_t * out_rc
)
```

Sends a request and receives a response to verify a certificate.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function handles the complete process of sending a request to verify a certificate and receiving the response. It blocks until the entire operation is complete or an error occurs.

Both results must be checked, e.g.:

```
int32_t verifyResult = 0;
int rc = wh_Client_CertVerify(c, cert, cert_len, rootId, &verifyResult);
if (rc != 0) {
    // transport/protocol failure - verdict unknown
}
else if (verifyResult != WH_ERROR_OK) {
    // round-trip ok, but the certificate did NOT verify
}
else {
    // certificate is valid
}
```

A.1.4.154 function wh_Client_CertVerifyAndCacheLeafPubKeyRequest

```
int wh_Client_CertVerifyAndCacheLeafPubKeyRequest(
    whClientContext * c,
    const uint8_t * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId,
    whNvmFlags cachedKeyFlags,
    whKeyId keyId
)
```

Sends a request to verify a certificate and cache the leaf public key.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.
- **cachedKeyFlags** NVM usage flags to apply when caching the leaf public key (e.g., WH_NVM_FLAGS_USAGE_VERIF WH_NVM_FLAGS_USAGE_SIGN).
- **keyId** The keyId to cache the leaf public key in. If set to WH_KEYID_ERASED, the server will pick a keyId.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to verify a certificate and also instructs the server to cache the public key of the leaf certificate. This function does not block; it returns immediately after sending the request.

A.1.4.155 function wh_Client_CertVerifyAndCacheLeafPubKeyResponse

```
int wh_Client_CertVerifyAndCacheLeafPubKeyResponse(
    whClientContext * c,
    whKeyId * out_keyId,
    int32_t * out_rc
)
```

Receives a response from the server after verifying a certificate and caching the leaf public key.

Parameters:

- **c** Pointer to the client context.
- **out_keyId** Pointer to store the key ID of the cached leaf public key.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function attempts to process a response message from the server after verifying a certificate and caching the leaf public key. It validates the response and extracts the return code and key ID. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.156 function wh_Client_CertVerifyAndCacheLeafPubKey

```
int wh_Client_CertVerifyAndCacheLeafPubKey(
    whClientContext * c,
    const uint8_t * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId,
    whNvmFlags cachedKeyFlags,
    whKeyId * inout_keyId,
    int32_t * out_rc
)
```

Sends a request and receives a response to verify a certificate, while also instructing the server to cache the public key of the leaf certificate.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.
- **cachedKeyFlags** NVM usage flags to apply when caching the leaf public key (e.g., WH_NVM_FLAGS_USAGE_VERIF WH_NVM_FLAGS_USAGE_SIGN).
- **inout_keyId** Pointer to the desired key ID of the cached leaf public key. If set to WH_KEYID_ERASED, the server will pick a keyId. On output, contains the keyId of the cached leaf public key.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function handles the complete process of sending a request to verify a certificate and cache the leaf public key, and receiving the response. It blocks until the entire operation is complete or an error occurs.

A.1.4.157 function wh_Client_CertVerifyMultiRootRequest

```
int wh_Client_CertVerifyMultiRootRequest(
    whClientContext * c,
    const uint8_t * cert,
    uint32_t cert_len,
    const whNvmId * trustedRootNvmIds,
    uint16_t numRoots
)
```

Sends a request to verify a certificate chain against a set of trusted root anchors.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmIds** Array of NVM IDs of trusted root certificates.
- **numRoots** Number of entries in trustedRootNvmIds (1.. WOLFHSM_CFG_CERT_MAX_VERIFY_ROOTS).

Return: int Returns 0 on success, or a negative error code on failure.

Generalizes wh_Client_CertVerifyRequest to accept an ordered list of trusted root NVM IDs. Verification succeeds if the chain anchors to any of the supplied roots. This function does not block; it returns immediately after sending the request.

A.1.4.158 function wh_Client_CertVerifyMultiRootResponse

```
int wh_Client_CertVerifyMultiRootResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after multi-root certificate verification.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

A.1.4.159 function wh_Client_CertVerifyMultiRoot

```
int wh_Client_CertVerifyMultiRoot(
    whClientContext * c,
    const uint8_t * cert,
    uint32_t cert_len,
    const whNvmId * trustedRootNvmIds,
    uint16_t numRoots,
    int32_t * out_rc
)
```

Sends a request and receives a response to verify a certificate chain against a set of trusted root anchors.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmIds** Array of NVM IDs of trusted root certificates.
- **numRoots** Number of entries in trustedRootNvmIds.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

A.1.4.160 function wh_Client_CertVerifyMultiRootAndCacheLeafPubKeyRequest

```
int wh_Client_CertVerifyMultiRootAndCacheLeafPubKeyRequest(
    whClientContext * c,
    const uint8_t * cert,
    uint32_t cert_len,
    const whNvmId * trustedRootNvmIds,
    uint16_t numRoots,
    whNvmFlags cachedKeyFlags,
    whKeyId keyId
)
```

Sends a request to verify a certificate chain against a set of trusted root anchors and cache the leaf certificate public key.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmIds** Array of NVM IDs of trusted root certificates.
- **numRoots** Number of entries in trustedRootNvmIds.
- **cachedKeyFlags** NVM usage flags for the cached leaf public key.
- **keyId** The keyId to cache the leaf public key in. If set to WH_KEYID_ERASED, the server will pick a keyId.

Return: int Returns 0 on success, or a negative error code on failure.

A.1.4.161 function wh_Client_CertVerifyMultiRootAndCacheLeafPubKeyResponse

```
int wh_Client_CertVerifyMultiRootAndCacheLeafPubKeyResponse(
    whClientContext * c,
    whKeyId * out_keyId,
    int32_t * out_rc
)
```

Receives a response from the server after multi-root certificate verification with leaf public key caching.

Parameters:

- **c** Pointer to the client context.
- **out_keyId** Pointer to store the key ID of the cached leaf public key.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

A.1.4.162 function wh_Client_CertVerifyMultiRootAndCacheLeafPubKey

```
int wh_Client_CertVerifyMultiRootAndCacheLeafPubKey(
    whClientContext * c,
    const uint8_t * cert,
```

```
    uint32_t cert_len,
    const whNvmId * trustedRootNvmIds,
    uint16_t numRoots,
    whNvmFlags cachedKeyFlags,
    whKeyId * inout_keyId,
    int32_t * out_rc
)
```

Sends a request and receives a response to verify a certificate chain against a set of trusted root anchors and cache the leaf certificate public key.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmIds** Array of NVM IDs of trusted root certificates.
- **numRoots** Number of entries in trustedRootNvmIds.
- **cachedKeyFlags** NVM usage flags for the cached leaf public key.
- **inout_keyId** Pointer to the desired key ID (in) / cached key ID (out).
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

A.1.4.163 function wh_Client_CertVerifyCacheClearRequest

```
int wh_Client_CertVerifyCacheClearRequest(
    whClientContext * c
)
```

Send a request to clear the server's trusted certificate verify cache.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

Subsequent verification of any certificate will re-run the public-key signature check until that cert is verified again and re-cached.

A.1.4.164 function wh_Client_CertVerifyCacheClearResponse

```
int wh_Client_CertVerifyCacheClearResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receive the response to a verify-cache clear request.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

A.1.4.165 function wh_Client_CertVerifyCacheClear

```
int wh_Client_CertVerifyCacheClear(
    whClientContext * c,
    int32_t * out_rc
)
```

Synchronous helper to clear the server's trusted certificate verify cache.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

A.1.4.166 function wh_Client_CertVerifyCacheSetEnabledRequest

```
int wh_Client_CertVerifyCacheSetEnabledRequest(
    whClientContext * c,
    uint8_t enable
)
```

Send a request to enable or disable the server's trusted certificate verify cache at runtime.

Parameters:

- **c** Pointer to the client context.
- **enable** Non-zero to enable caching, zero to disable.

Return: int Returns 0 on success, or a negative error code on failure.

Disabling clears all existing cache entries and suppresses both subsequent lookups and inserts until the cache is re-enabled. Enabling resumes normal caching from an empty state. The cache defaults to enabled at server init so this call is only needed to opt out (or re-enable after opting out).

In per-client cache mode the toggle is scoped to this client's server. With WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE_ the toggle is shared across all clients connected to the same NVM context.

A.1.4.167 function wh_Client_CertVerifyCacheSetEnabledResponse

```
int wh_Client_CertVerifyCacheSetEnabledResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receive the response to a verify-cache enable/disable request.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

A.1.4.168 function wh_Client_CertVerifyCacheSetEnabled

```
int wh_Client_CertVerifyCacheSetEnabled(
    whClientContext * c,
    uint8_t enable,
    int32_t * out_rc
)
```

Synchronous helper to enable or disable the server's trusted certificate verify cache.

Parameters:

- **c** Pointer to the client context.
- **enable** Non-zero to enable caching, zero to disable.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

A.1.4.169 function wh_Client_CertAddTrustedDmaRequest

```
int wh_Client_CertAddTrustedDmaRequest(
    whClientContext * c,
    whNvmId id,
    whNvmAccess access,
    whNvmFlags flags,
    uint8_t * label,
    whNvmSize label_len,
    const void * cert,
    uint32_t cert_len
)
```

Sends a request to add a trusted certificate to NVM storage using DMA.

Parameters:

- **c** Pointer to the client context.
- **id** NVM ID to store the trusted certificate.
- **cert** Pointer to the certificate data to add.
- **cert_len** Length of the certificate data.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to add a trusted certificate to NVM storage using DMA. This function does not block; it returns immediately after sending the request.

A.1.4.170 function wh_Client_CertAddTrustedDmaResponse

```
int wh_Client_CertAddTrustedDmaResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after adding a trusted certificate using DMA.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function attempts to process a response message from the server after adding a trusted certificate using DMA. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.171 function wh_Client_CertAddTrustedDma

```
int wh_Client_CertAddTrustedDma(
    whClientContext * c,
    whNvmId id,
    whNvmAccess access,
    whNvmFlags flags,
    uint8_t * label,
    whNvmSize label_len,
    const void * cert,
    uint32_t cert_len,
    int32_t * out_rc
)
```

Sends a request and receives a response to add a trusted certificate using DMA.

Parameters:

- **c** Pointer to the client context.
- **id** NVM ID to store the trusted certificate.
- **cert** Pointer to the certificate data to add.
- **cert_len** Length of the certificate data.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to add a trusted certificate using DMA and receiving the response. It blocks until the entire operation is complete or an error occurs.

A.1.4.172 function wh_Client_CertReadTrustedDmaRequest

```
int wh_Client_CertReadTrustedDmaRequest(
    whClientContext * c,
    whNvmId id,
    void * cert,
    uint32_t cert_len
)
```

Sends a request to read a trusted certificate from NVM storage using DMA.

Parameters:

- **c** Pointer to the client context.
- **id** NVM ID of the trusted certificate to get.
- **cert** Pointer to buffer to store the certificate data.
- **cert_len** Length of the certificate buffer.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to read a trusted certificate from NVM storage using DMA. This function does not block; it returns immediately after sending the request.

A.1.4.173 function wh_Client_CertReadTrustedDmaResponse

```
int wh_Client_CertReadTrustedDmaResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after reading a trusted certificate using DMA.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function attempts to process a response message from the server after reading a trusted certificate using DMA. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.174 function wh_Client_CertReadTrustedDma

```
int wh_Client_CertReadTrustedDma(
    whClientContext * c,
    whNvmId id,
    void * cert,
    uint32_t cert_len,
    int32_t * out_rc
)
```

Sends a request and receives a response to read trusted certificate using DMA.

Parameters:

- **c** Pointer to the client context.
- **id** NVM ID of the trusted certificate to get.
- **cert** Pointer to buffer to store the certificate data.
- **cert_len** Length of the certificate buffer.
- **out_rc** Pointer to store the response code from the server.

Return: int Returns 0 on success, or a negative error code on failure.

This function handles the complete process of sending a request to read a trusted certificate using DMA and receiving the response. It blocks until the entire operation is complete or an error occurs.

A.1.4.175 function wh_Client_CertVerifyDmaRequest

```
int wh_Client_CertVerifyDmaRequest(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId
)
```

Sends a request to verify a certificate using DMA.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to verify a certificate using DMA. This function does not block; it returns immediately after sending the request.

A.1.4.176 function wh_Client_CertVerifyDmaResponse

```
int wh_Client_CertVerifyDmaResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after verifying a certificate using DMA.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function attempts to process a response message from the server after verifying a certificate using DMA. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.177 function wh_Client_CertVerifyDma

```
int wh_Client_CertVerifyDma(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId,
    int32_t * out_rc
)
```

Sends a request and receives a response to verify a certificate using DMA.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function handles the complete process of sending a request to verify a certificate using DMA and receiving the response. It blocks until the entire operation is complete or an error occurs.

A.1.4.178 function wh_Client_CertVerifyDmaAndCacheLeafPubKeyRequest

```
int wh_Client_CertVerifyDmaAndCacheLeafPubKeyRequest(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId,
    whNvmFlags cachedKeyFlags,
    whKeyId keyId
)
```

Sends a request to verify a certificate using DMA and cache the leaf certificate public key.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.
- **cachedKeyFlags** NVM usage flags to apply when caching the leaf public key (e.g., WH_NVM_FLAGS_USAGE_VERIF WH_NVM_FLAGS_USAGE_SIGN).
- **keyId** The keyId to cache the leaf public key in. If set to WH_KEYID_ERASED, the server will pick a keyId.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to verify a certificate using DMA and also instructs the server to cache the public key of the leaf certificate. This function does not block; it returns immediately after sending the request.

A.1.4.179 function wh_Client_CertVerifyDmaAndCacheLeafPubKeyResponse

```
int wh_Client_CertVerifyDmaAndCacheLeafPubKeyResponse(
    whClientContext * c,
    whKeyId * out_keyId,
    int32_t * out_rc
)
```

Receives a response from the server after verifying a certificate using DMA and caching the leaf public key.

Parameters:

- **c** Pointer to the client context.
- **out_keyId** Pointer to store the key ID of the cached leaf public key.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function attempts to process a response message from the server after verifying a certificate using DMA and caching the leaf public key. It validates the response and extracts the return code and key ID. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.180 function wh_Client_CertVerifyDmaAndCacheLeafPubKey

```
int wh_Client_CertVerifyDmaAndCacheLeafPubKey(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId,
    whNvmFlags cachedKeyFlags,
    whKeyId * inout_keyId,
    int32_t * out_rc
)
```

Sends a request and receives a response to verify a certificate using DMA and cache the leaf certificate public key.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.
- **cachedKeyFlags** NVM usage flags to apply when caching the leaf public key (e.g., WH_NVM_FLAGS_USAGE_VERIF WH_NVM_FLAGS_USAGE_SIGN).
- **inout_keyId** Pointer to the desired key ID of the cached leaf public key. If set to WH_KEYID_ERASED, the server will pick a keyId. On output, contains the keyId of the cached leaf public key.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function handles the complete process of sending a request to verify a certificate using DMA and cache the leaf certificate public key, and receiving the response. It blocks until the entire operation is complete or an error occurs.

A.1.4.181 function wh_Client_CertVerifyMultiRootDmaRequest

```
int wh_Client_CertVerifyMultiRootDmaRequest(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    const whNvmId * trustedRootNvmIds,
    uint16_t numRoots
)
```

Sends a DMA request to verify a certificate chain against a set of trusted root anchors.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmIds** Array of NVM IDs of trusted root certificates.
- **numRoots** Number of entries in trustedRootNvmIds (1.. WOLFHSM_CFG_CERT_MAX_VERIFY_ROOTS).

Return: int Returns 0 on success, or a negative error code on failure.

Generalizes wh_Client_CertVerifyDmaRequest to accept an ordered list of trusted root NVM IDs. Verification succeeds if the chain anchors to any of the supplied roots. This function does not block; it returns immediately after sending the request.

A.1.4.182 function wh_Client_CertVerifyMultiRootDmaResponse

```
int wh_Client_CertVerifyMultiRootDmaResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after multi-root DMA certificate verification.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

A.1.4.183 function wh_Client_CertVerifyMultiRootDma

```
int wh_Client_CertVerifyMultiRootDma(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    const whNvmId * trustedRootNvmIds,
    uint16_t numRoots,
    int32_t * out_rc
)
```

Sends a request and receives a response to verify a certificate chain via DMA against a set of trusted root anchors.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmIds** Array of NVM IDs of trusted root certificates.
- **numRoots** Number of entries in trustedRootNvmIds.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

A.1.4.184 function wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKeyRequest

```
int wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKeyRequest(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    const whNvmId * trustedRootNvmIds,
    uint16_t numRoots,
    whNvmFlags cachedKeyFlags,
    whKeyId keyId
)
```

Sends a DMA request to verify a certificate chain against a set of trusted root anchors and cache the leaf certificate public key.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmIds** Array of NVM IDs of trusted root certificates.
- **numRoots** Number of entries in trustedRootNvmIds.

- **cachedKeyFlags** NVM usage flags for the cached leaf public key.
- **keyId** The keyId to cache the leaf public key in. If set to WH_KEYID_ERASED, the server will pick a keyId.

Return: int Returns 0 on success, or a negative error code on failure.

A.1.4.185 function wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKeyResponse

```
int wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKeyResponse(
    whClientContext * c,
    whKeyId * out_keyId,
    int32_t * out_rc
)
```

Receives a response from the server after multi-root DMA certificate verification with leaf public key caching.

Parameters:

- **c** Pointer to the client context.
- **out_keyId** Pointer to store the key ID of the cached leaf public key.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

A.1.4.186 function wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKey

```
int wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKey(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    const whNvmId * trustedRootNvmIds,
    uint16_t numRoots,
    whNvmFlags cachedKeyFlags,
    whKeyId * inout_keyId,
    int32_t * out_rc
)
```

Sends a request and receives a response to verify a certificate chain via DMA against a set of trusted root anchors and cache the leaf certificate public key.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the certificate data to verify.
- **cert_len** Length of the certificate data.
- **trustedRootNvmIds** Array of NVM IDs of trusted root certificates.
- **numRoots** Number of entries in trustedRootNvmIds.
- **cachedKeyFlags** NVM usage flags for the cached leaf public key.
- **inout_keyId** Pointer to the desired key ID (in) / cached key ID (out).
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

A.1.4.187 function wh_Client_CertVerifyAcertRequest

```
int wh_Client_CertVerifyAcertRequest(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId
)
```

Sends a request to verify an attribute certificate.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the attribute certificate data to verify.
- **cert_len** Length of the attribute certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a request to verify an attribute certificate against a trusted root certificate. This function does not block; it returns immediately after sending the request.

A.1.4.188 function wh_Client_CertVerifyAcertResponse

```
int wh_Client_CertVerifyAcertResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after verifying an attribute certificate.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function attempts to process a response message from the server after verifying an attribute certificate. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.189 function wh_Client_CertVerifyAcert

```
int wh_Client_CertVerifyAcert(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId,
    int32_t * out_rc
)
```

Sends a request and receives a response to verify an attribute certificate.

Parameters:

- **c** Pointer to the client context.

- **cert** Pointer to the attribute certificate data to verify.
- **cert_len** Length of the attribute certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function handles the complete process of sending a request to verify an attribute certificate and receiving the response. It blocks until the entire operation is complete or an error occurs.

A.1.4.190 function wh_Client_CertVerifyAcertDmaRequest

```
int wh_Client_CertVerifyAcertDmaRequest(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId
)
```

Prepares and sends a DMA request to verify an attribute certificate.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the attribute certificate data to verify.
- **cert_len** Length of the attribute certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.

Return: int Returns 0 on success, or a negative error code on failure.

This function prepares and sends a DMA request to verify an attribute certificate against a trusted root certificate. This function does not block; it returns immediately after sending the request.

A.1.4.191 function wh_Client_CertVerifyAcertDmaResponse

```
int wh_Client_CertVerifyAcertDmaResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a response from the server after verifying an attribute certificate using DMA.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function attempts to process a response message from the server after verifying an attribute certificate using DMA. It validates the response and extracts the return code. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

A.1.4.192 function wh_Client_DmaRegisterAllowList

```
int wh_Client_DmaRegisterAllowList(
    struct whClientContext_t * client,
    const whDmaAddrAllowList * allowlist
)
```

Registers a DMA address allowlist for client-side validation.

Parameters:

- **client** Pointer to the client context.
- **allowlist** Pointer to the DMA address allowlist structure.

Return: int Returns WH_ERROR_OK on success, or WH_ERROR_BADARGS if the arguments are invalid.

This function allows the client to register an allowlist of valid DMA addresses. The allowlist will be checked during DMA operations to ensure addresses are within allowed ranges.

A.1.4.193 function wh_Client_DmaRegisterCb

```
int wh_Client_DmaRegisterCb(
    struct whClientContext_t * client,
    whClientDmaClientMemCb cb
)
```

Registers a custom client DMA callback.

Parameters:

- **client** Pointer to the client context.
- **cb** The custom DMA callback handler to register.

Return: int Returns WH_ERROR_OK on success, or WH_ERROR_BADARGS if the arguments are invalid.

This function allows the client to register a custom callback handler for processing memory operations. The callback will be invoked during DMA operations to transform client addresses, manipulate caches, etc.

A.1.4.194 function wh_Client_DmaProcessClientAddress

```
int wh_Client_DmaProcessClientAddress(
    struct whClientContext_t * client,
    uintptr_t clientAddr,
    void ** serverPtr,
    size_t len,
    whDmaOper oper,
    whDmaFlags flags
)
```

Processes a client address for DMA operations, using the native pointer size of the system.

Parameters:

- **client** Pointer to the client context.
- **clientAddr** The client address to be processed.
- **serverPtr** Pointer to store the transformed server address.
- **len** The length of the memory operation.
- **oper** The DMA operation type (e.g., read or write).
- **flags** Flags for the DMA operation.

Return: int Returns WH_ERROR_OK on success, WH_ERROR_BADARGS if the arguments are invalid, or a negative error code on failure.

This function transforms a client address for DMA operations. It performs user-supplied address transformations, cache manipulations, and checks the transformed address against the client's allowlist if registered.

A.1.4.195 function wh_Client_DmaAsyncPre

```
int wh_Client_DmaAsyncPre(
    struct whClientContext_t * client,
    whClientDmaAsyncBuf * buf,
    uintptr_t clientAddr,
    uint64_t len,
    whDmaOper preOper,
    uintptr_t * outXformedAddr
)
```

Runs the PRE half of a DMA buffer mapping and stashes it (INTERNAL).

Parameters:

- **client** Pointer to the client context.
- **buf** The slot to clear and populate with the mapping.
- **clientAddr** Original client address to translate.
- **len** Length of the buffer (0 to skip / clear only).
- **preOper** Client PRE operation (READ_PRE or WRITE_PRE).
- **outXformedAddr** Receives the translated address (0 when skipped).

Return: WH_ERROR_OK on success or skip, WH_ERROR_BADARGS on NULL args or a non-PRE operation, or the port PRE callback's error.

Shared between the client *Dma source files; not port-facing. Clears the slot, translates clientAddr for preOper, and on success stashes the mapping (translated address, client address, length, and the matching POST operation) so the matching Response can run wh_Client_DmaAsyncPost(). The POST direction is derived from preOper (READ_PRE -> READ_POST, WRITE_PRE -> WRITE_POST), so the caller cannot pair them inconsistently. The translated address is returned in outXformedAddr for the caller to place in the request message. When len is 0 (e.g. an optional buffer that is absent) it is a no-op: the slot stays cleared and outXformedAddr is set to 0.

A.1.4.196 function wh_Client_DmaAsyncPost

```
int wh_Client_DmaAsyncPost(
    struct whClientContext_t * client,
    whClientDmaAsyncBuf * buf
)
```

Runs the POST half of a stashed DMA buffer mapping (INTERNAL).

Parameters:

- **client** Pointer to the client context.
- **buf** The stashed single-buffer mapping to clean up.

Return: WH_ERROR_OK on success or when there is nothing to clean up (buf->sz == 0), WH_ERROR_BADARGS on NULL client/buf, or the port POST callback's error (e.g. a failed unmap or copy-back).

Shared between the client *Dma source files; not port-facing. Releases (and, for a server-write buffer, copies back) a mapping stashed by the matching Request, using buf->postOper for the direction. A

no-op (WH_ERROR_OK) when buf->sz is 0; clears the whole slot (even on failure) so a later Response cannot re-run it.

A.1.4.197 function wh_Client_CertVerifyAcertDma

```
int wh_Client_CertVerifyAcertDma(
    whClientContext * c,
    const void * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId,
    int32_t * out_rc
)
```

Sends a DMA request and receives a response to verify an attribute certificate.

Parameters:

- **c** Pointer to the client context.
- **cert** Pointer to the attribute certificate data to verify.
- **cert_len** Length of the attribute certificate data.
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against.
- **out_rc** Required (non-NULL); receives the verification verdict (WH_ERROR_OK if the certificate is valid). Passing NULL returns WH_ERROR_BADARGS.

Return: int Returns 0 when the request/response round-trip completes. This alone does NOT mean the certificate is valid: the caller MUST also check that *out_rc == WH_ERROR_OK. Returns a negative error code on transport failure.

This function handles the complete process of sending a DMA request to verify an attribute certificate and receiving the response. It blocks until the entire operation is complete or an error occurs.

A.1.5 Source code

```
/*
 * Copyright (C) 2024 wolfSSL Inc.
 *
 * This file is part of wolfHSM.
 *
 * wolfHSM is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
 * wolfHSM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with wolfHSM.  If not, see <http://www.gnu.org/licenses/>.
 */
/*
 * wolfhsm/wh_client.h
 *
 * Base WolfHSM Client Library API
 *
 * The WolfHSM Client provides a single context and connection to a
```

```
 * WolfHSM Server.  All communications and state are internally managed by
 * registering a crypto callback function to be invoked synchronously when
 * wolfCrypt functions are called.  In order to specify to use the WolfHSM
 * Server for cryptographic operations, pass the client's devId into any of
 * the wolfCrypt init functions. Set the devId in whClientConfig.devId (0 picks
 * the default WH_DEV_ID) and read it back with WH_CLIENT_DEVID(c). With a
 * single client you can just pass the global WH_DEV_ID (or WH_DEV_ID_DMA for
 * DMA only). With more than one client, give each its own devId. See
 * WH_CLIENT_DEVID and WH_DEV_ID below.
 *
 * In addition to the offload of cryptographic functions, the WolfHSM Client
 * also exposes WolfHSM Server key management, non-volatile memory, and
 ↪  protocol
 * functions.
 *
 */

#ifndef WOLFHSM_WH_CLIENT_H_
#define WOLFHSM_WH_CLIENT_H_

/* Pick up compile-time configuration */
#include "wolfhsm/wh_settings.h"

/* System libraries */
#include <stdint.h>

/* Common WolfHSM types and defines shared with the server */
#include "wolfhsm/wh_common.h"

/* Component includes */
#include "wolfhsm/wh_comm.h"
#include "wolfhsm/wh_message_customcb.h"
#ifdef WOLFHSM_CFG_DMA
#include "wolfhsm/wh_dma.h"
#endif /* WOLFHSM_CFG_DMA */
#include "wolfhsm/wh_keyid.h"
#include "wolfhsm/wh_auth.h"


/* Forward declaration of the client structure so its elements can reference
 * itself  (e.g. server argument to custom callback) */
typedef struct whClientContext_t whClientContext;

#ifndef WOLFHSM_CFG_NO_CRYPTO

/* WolfCrypt types and defines */
#include "wolfssl/wolfcrypt/types.h"

/* Default devId for wolfHSM offload; you can override the value. Every
 * wh_Client_Init() registers it, so it works like any per-client devId
 * (including DMA mode, see wh_Client_SetDmaMode). It is also the devId used
 * when a client config leaves devId 0. Calls on it always go to the most
 * recently initialized client, and any client's wh_Client_Cleanup()
 * unregisters it, so only pass it to wolfCrypt directly when there is one
```

```
 * client; with more, give each its own devId. New code should not use this
 * directly; it is kept only for backwards compatibility. */
#ifndef WH_DEV_ID
#define WH_DEV_ID 0x5748534D /* "WHSM" */
#endif

#if WH_DEV_ID <= 0
#error "WH_DEV_ID must be a positive, nonzero value"
#endif

#ifdef WOLFHSM_CFG_DMA

/* Default devId for DMA-only wolfHSM offload; you can override the value. It
 * is reserved, so it is not valid as whClientConfig.devId. Like WH_DEV_ID it
 * goes to the most recently initialized client and any wh_Client_Cleanup()
 * unregisters it, so it is for single-client use only. New code should not use
 * this directly; it is kept only for backwards compatibility. */
#ifndef WH_DEV_ID_DMA
#define WH_DEV_ID_DMA 0x57444D41 /* "WDMA" */
#endif /* WH_DEV_ID_DMA */

#if WH_DEV_ID_DMA <= 0
#error "WH_DEV_ID_DMA must be a positive, nonzero value"
#endif
#if WH_DEV_ID == WH_DEV_ID_DMA
#error "WH_DEV_ID and WH_DEV_ID_DMA must be distinct"
#endif

#endif /* WOLFHSM_CFG_DMA */

#else /* WOLFHSM_CFG_NO_CRYPTO */

/*  for compile purpose */
#define WH_DEV_ID -2 /* invalid ID */
/* cipher types */
enum wc_CipherType {
    WC_CIPHER_NONE = 0,
};

#endif /* !WOLFHSM_CFG_NO_CRYPTO*/

/* Get the devId for a client context (c is a whClientContext*). This is the
 * devId set at wh_Client_Init(): whClientConfig.devId, or WH_DEV_ID if that
 * was 0. Only valid after a successful init. */
#define WH_CLIENT_DEVID(c) ((c)->devId)

#ifdef WOLFHSM_CFG_DMA
typedef int (*whClientDmaClientMemCb)(struct whClientContext_t* client,
                                      uintptr_t clientAddr, void** ptr,
                                      size_t len, whDmaOper oper,
                                      whDmaFlags flags);

/* Common DMA callback types and structures */
typedef struct {
```

```
    whClientDmaClientMemCb    cb;
    const whDmaAddrAllowList* dmaAddrAllowList; /* allowed addresses */
    /* nonzero to prefer the DMA path */
    uint32_t preferDma;
} whClientDmaConfig;

/* Per-operation async DMA context: stores translated input DMA address
 * that must survive across the Request/Response boundary for POST cleanup.
 * State is now passed inline (not via DMA), so only input tracking is needed.
 * ioAddr:      translated DMA address for input POST
 * clientAddr:  original client address for POST
 * ioSz:        DMA'd size for POST */
typedef struct {
    uintptr_t ioAddr;
    uintptr_t clientAddr;
    uint64_t  ioSz;
} whClientDmaAsyncSha;

/* Per-operation async DMA context for RNG: stores the translated output DMA
 * address that must survive across the Request/Response boundary for POST
 * cleanup of the client's output buffer. */
typedef struct {
    uintptr_t outAddr;
    uintptr_t clientAddr;
    uint64_t  outSz;
} whClientDmaAsyncRng;

/* Per-operation async DMA context for AES modes (ECB/CBC/CTR/GCM): stores
 * translated DMA addresses for the input, output, and optional AAD buffers
 * so the matching *Response can run POST cleanup.
 *
 * For non-GCM modes only input and output are used; aadSz stays 0 and aad
 * cleanup is skipped. A sz of 0 means "no mapping held" and the Response
 * will skip POST cleanup for that buffer. */
typedef struct {
    uintptr_t inAddr; /* translated input DMA address   (READ) */
    uintptr_t inClientAddr;
    uint64_t  inSz;
    uintptr_t outAddr; /* translated output DMA address  (WRITE) */
    uintptr_t outClientAddr;
    uint64_t  outSz;
    uintptr_t aadAddr; /* GCM AAD DMA address            (READ) */
    uintptr_t aadClientAddr;
    uint64_t  aadSz;
} whClientDmaAsyncAes;

/* Per-operation async DMA context for CMAC: same shape as SHA since CMAC
 * DMA only transfers the input buffer. Stashed across Request/Response for
 * POST cleanup. inSz == 0 means "nothing to clean up". */
typedef struct {
    uintptr_t inAddr;
    uintptr_t clientAddr;
    uint64_t  inSz;
} whClientDmaAsyncCmac;
```

```
/* One client buffer mapped across a DMA Request/Response boundary. The Request
 * stashes the translated address, original client address, length, and POST
 * direction; the Response runs wh_Client_DmaAsyncPost(). sz == 0 means nothing
 * to clean up. postOper keeps the POST direction-correct when this shared union
 * member is used by different ops; it does not make a mispaired
 ↪  Request/Response
 * safe (the one-in-flight, self-paired invariant still applies). */
typedef struct {
    uintptr_t xformedAddr;
    uintptr_t clientAddr;
    uint64_t  sz;
    whDmaOper postOper;
} whClientDmaAsyncBuf;

/* Two buffers mapped together for NvmAddObjectDma (metadata + optional data).
 ↪  */
typedef struct {
    whClientDmaAsyncBuf meta;
    whClientDmaAsyncBuf data;
} whClientDmaAsyncNvmAdd;

/* Async DMA context union; only one DMA request is in flight at a time. The
 * crypto members (sha/rng/cmac/aes) are bespoke and predate the generic
 ↪  holder.
 * Key/NVM ops use `buf` (single-buffer) and `nvmAdd` (two-buffer). */
typedef union {
    whClientDmaAsyncSha    sha;
    whClientDmaAsyncRng    rng;
    whClientDmaAsyncAes    aes;
    whClientDmaAsyncCmac   cmac;
    whClientDmaAsyncBuf    buf;
    whClientDmaAsyncNvmAdd nvmAdd;
} whClientDmaAsyncCtx;

typedef struct {
    whClientDmaClientMemCb    cb;
    const whDmaAddrAllowList* dmaAddrAllowList; /* allowed addresses */
    void* heap; /* heap hint for using static memory (or other allocator) */
    whClientDmaAsyncCtx asyncCtx;
    /* nonzero to prefer the DMA path */
    uint32_t preferDma;
} whClientDmaContext;
#endif /* WOLFHSM_CFG_DMA */

/* Client context */
struct whClientContext_t {
    uint16_t     last_req_id;
    uint16_t     last_req_kind;
    uint32_t     cryptoAffinity;
    /* devId registered at init (see WH_CLIENT_DEVID). Nonzero only after a
     * successful init. */
    int devId;
    /* Nonzero once this context has called wolfCrypt_Init(), so
```

```
     * wh_Client_Cleanup() calls wolfCrypt_Cleanup() once to match and not when
     * init failed before wolfCrypt_Init() ran (e.g. comm init failed). */
    int cryptoInitialized;
#ifdef WOLFHSM_CFG_DMA
    whClientDmaContext dma;
#endif /* WOLFHSM_CFG_DMA */
    whCommClient comm[1];
};

struct whClientConfig_t {
    whCommClientConfig* comm;
    /* devId to register for this client. 0 picks the default WH_DEV_ID.
     * Otherwise it must be positive, and (with WOLFHSM_CFG_DMA) not
     * WH_DEV_ID_DMA, which is reserved. Ignored when WOLFHSM_CFG_NO_CRYPTO. */
    int devId;
#ifdef WOLFHSM_CFG_DMA
    whClientDmaConfig* dmaConfig;
#endif /* WOLFHSM_CFG_DMA */
};
typedef struct whClientConfig_t whClientConfig;


int wh_Client_Init(whClientContext* c, const whClientConfig* config);

int wh_Client_Cleanup(whClientContext* c);

int wh_Client_SendRequest(whClientContext* c, uint16_t group, uint16_t action,
                          uint16_t data_size, const void* data);
int wh_Client_RecvResponse(whClientContext* c, uint16_t* out_group,
                           uint16_t* out_action, uint16_t* out_size,
                           void* data);

int wh_Client_IsRequestPending(const whClientContext* c);

int wh_Client_CommInitRequest(whClientContext* c);

int wh_Client_CommInitResponse(whClientContext* c, uint32_t* out_clientid,
                               uint32_t* out_serverid);

int wh_Client_CommInit(whClientContext* c, uint32_t* out_clientid,
                       uint32_t* out_serverid);


int wh_Client_CommInfoRequest(whClientContext* c);

int wh_Client_CommInfoResponse(whClientContext* c,
        uint8_t* out_version,
        uint8_t* out_build,
        uint32_t *out_cfg_comm_data_len,
        uint32_t *out_cfg_nvm_object_count,
        uint32_t *out_cfg_keycache_count,
        uint32_t *out_cfg_keycache_bufsize,
        uint32_t *out_cfg_keycache_bigcount,
        uint32_t *out_cfg_keycache_bigbufsize,
```

```
        uint32_t *out_cfg_customcb_count,
        uint32_t *out_cfg_dmaaddr_count,
        uint32_t *out_debug_state,
        uint32_t *out_boot_state,
        uint32_t *out_lifecycle_state,
        uint32_t *out_nvm_state);

int wh_Client_CommInfo(whClientContext* c,
        uint8_t* out_version,
        uint8_t* out_build,
        uint32_t *out_cfg_comm_data_len,
        uint32_t *out_cfg_nvm_object_count,
        uint32_t *out_cfg_keycache_count,
        uint32_t *out_cfg_keycache_bufsize,
        uint32_t *out_cfg_keycache_bigcount,
        uint32_t *out_cfg_keycache_bigbufsize,
        uint32_t *out_cfg_customcb_count,
        uint32_t *out_cfg_dmaaddr_count,
        uint32_t *out_debug_state,
        uint32_t *out_boot_state,
        uint32_t *out_lifecycle_state,
        uint32_t *out_nvm_state);

int wh_Client_SetCryptoAffinity(whClientContext* c, uint32_t affinity);

int wh_Client_GetCryptoAffinity(whClientContext* c, uint32_t* out_affinity);

int wh_Client_SetDmaMode(whClientContext* c, int useDma);

int wh_Client_GetDmaMode(whClientContext* c, int* out_useDma);

int wh_Client_CommCloseRequest(whClientContext* c);

int wh_Client_CommCloseResponse(whClientContext* c);

int wh_Client_CommClose(whClientContext* c);

int wh_Client_EchoRequest(whClientContext* c, uint16_t size, const void* data);

int wh_Client_EchoResponse(whClientContext* c, uint16_t* out_size, void* data);

int wh_Client_Echo(whClientContext* c, uint16_t snd_len, const void* snd_data,
                   uint16_t* out_rcv_len, void* rcv_data);

int wh_Client_KeyCacheRequest_ex(whClientContext* c, uint32_t flags,
                                 uint8_t* label, uint16_t labelSz, const
                                  ↪  uint8_t* in,
                                 uint16_t inSz, uint16_t keyId);

int wh_Client_KeyCacheRequest(whClientContext* c, uint32_t flags,
                              uint8_t* label, uint16_t labelSz, const uint8_t* in,
                                 uint16_t inSz);

int wh_Client_KeyCacheResponse(whClientContext* c, uint16_t* keyId);
```

```
int wh_Client_KeyCache(whClientContext* c, uint32_t flags, uint8_t* label,
                       uint16_t labelSz, const uint8_t* in, uint16_t inSz,
                       uint16_t* keyId);

int wh_Client_KeyEvictRequest(whClientContext* c, uint16_t keyId);

int wh_Client_KeyEvictResponse(whClientContext* c);

int wh_Client_KeyEvict(whClientContext* c, uint16_t keyId);

int wh_Client_KeyExportRequest(whClientContext* c, uint16_t keyId);

int wh_Client_KeyExportResponse(whClientContext* c, uint8_t* label,
                                uint16_t labelSz, uint8_t* out,
                                uint16_t* outSz);

int wh_Client_KeyExport(whClientContext* c, uint16_t keyId, uint8_t* label,
                        uint16_t labelSz, uint8_t* out, uint16_t* outSz);

int wh_Client_KeyExportPublicRequest(whClientContext* c, whKeyId keyId,
                                     uint16_t algo);

int wh_Client_KeyExportPublicResponse(whClientContext* c, uint8_t* label,
                                      uint16_t labelSz, uint8_t* out,
                                      uint16_t* outSz);

int wh_Client_KeyExportPublic(whClientContext* c, whKeyId keyId, uint16_t algo,
                              uint8_t* label, uint16_t labelSz, uint8_t* out,
                              uint16_t* outSz);

int wh_Client_KeyCommitRequest(whClientContext* c, whNvmId keyId);

int wh_Client_KeyCommitResponse(whClientContext* c);

int wh_Client_KeyCommit(whClientContext* c, whNvmId keyId);

int wh_Client_KeyEraseRequest(whClientContext* c, whNvmId keyId);

int wh_Client_KeyEraseResponse(whClientContext* c);

int wh_Client_KeyErase(whClientContext* c, whNvmId keyId);

int wh_Client_KeyRevokeRequest(whClientContext* c, whKeyId keyId);

int wh_Client_KeyRevokeResponse(whClientContext* c);

int wh_Client_KeyRevoke(whClientContext* c, whKeyId keyId);

#ifdef WOLFHSM_CFG_DMA

int wh_Client_KeyCacheDmaRequest(whClientContext* c, uint32_t flags,
                                 uint8_t* label, uint16_t labelSz,
                                 const void* keyAddr, uint16_t keySz,
```

```
                                uint16_t keyId);

int wh_Client_KeyCacheDmaResponse(whClientContext* c, uint16_t* keyId);

int wh_Client_KeyCacheDma(whClientContext* c, uint32_t flags, uint8_t* label,
                          uint16_t labelSz, const void* keyAddr, uint16_t keySz,
                           uint16_t* keyId);

int wh_Client_KeyExportDmaRequest(whClientContext* c, uint16_t keyId,
                                  const void* keyAddr, uint16_t keySz);

int wh_Client_KeyExportDmaResponse(whClientContext* c, uint8_t* label,
                                   uint16_t labelSz, uint16_t* outSz);

int wh_Client_KeyExportDma(whClientContext* c, uint16_t keyId,
                           const void* keyAddr, uint16_t keySz, uint8_t* label,
                           uint16_t labelSz, uint16_t* outSz);

int wh_Client_KeyExportPublicDmaRequest(whClientContext* c, whKeyId keyId,
                                        uint16_t algo, void* keyAddr,
                                        uint16_t keySz);

int wh_Client_KeyExportPublicDmaResponse(whClientContext* c, uint8_t* label,
                                         uint16_t labelSz, uint16_t* outSz);

int wh_Client_KeyExportPublicDma(whClientContext* c, whKeyId keyId,
                                 uint16_t algo, void* keyAddr,
                                 uint16_t keySz, uint8_t* label,
                                 uint16_t labelSz, uint16_t* outSz);
#endif /* WOLFHSM_CFG_DMA */

int wh_Client_KeyWrap(whClientContext* ctx, enum wc_CipherType cipherType,
                      uint16_t serverKeyId, void* keyIn, uint16_t keySz,
                      whNvmMetadata* metadataIn, void* wrappedKeyOut,
                      uint16_t* wrappedKeyInOutSz);

int wh_Client_KeyWrapRequest(whClientContext*   ctx,
                             enum wc_CipherType cipherType,
                             uint16_t serverKeyId, void* key, uint16_t keySz,
                             whNvmMetadata* metadata);

int wh_Client_KeyWrapResponse(whClientContext*   ctx,
                              enum wc_CipherType cipherType,
                             void* wrappedKeyOut, uint16_t* wrappedKeyInOutSz);

int wh_Client_KeyUnwrapAndExport(whClientContext*   ctx,
                                 enum wc_CipherType cipherType,
                                 uint16_t serverKeyId, void* wrappedKeyIn,
                                 uint16_t       wrappedKeySz,
                                 whNvmMetadata* metadataOut, void* keyOut,
                                 uint16_t* keyInOutSz);

int wh_Client_KeyUnwrapAndExportRequest(whClientContext*   ctx,
                                        enum wc_CipherType cipherType,
```

```
                                        uint16_t           serverKeyId,
                                        void*              wrappedKeyIn,
                                        uint16_t           wrappedKeySz);

int wh_Client_KeyUnwrapAndExportResponse(whClientContext*   ctx,
                                         enum wc_CipherType cipherType,
                                         whNvmMetadata*     metadataOut,
                                         void* keyOut, uint16_t* keyInOutSz);

int wh_Client_KeyUnwrapAndCache(whClientContext*   ctx,
                                enum wc_CipherType cipherType,
                                uint16_t serverKeyId, void* wrappedKeyIn,
                                uint16_t wrappedKeySz, uint16_t* keyIdOut);
int wh_Client_KeyUnwrapAndCacheRequest(whClientContext*   ctx,
                                       enum wc_CipherType cipherType,
                                      uint16_t serverKeyId, void* wrappedKeyIn,
                                       uint16_t wrappedKeySz);
int wh_Client_KeyUnwrapAndCacheResponse(whClientContext*   ctx,
                                        enum wc_CipherType cipherType,
                                        uint16_t*          keyIdOut);

int wh_Client_DataWrap(whClientContext* ctx, enum wc_CipherType cipherType,
                       uint16_t serverKeyId, void* dataIn, uint32_t dataInSz,
                       void* wrappedDataOut, uint32_t* wrappedDataInOutSz);

int wh_Client_DataWrapRequest(whClientContext*   ctx,
                              enum wc_CipherType cipherType,
                              uint16_t serverKeyId, void* dataIn,
                              uint32_t dataInSz);

int wh_Client_DataWrapResponse(whClientContext*   ctx,
                               enum wc_CipherType cipherType,
                               void* wrappedDataOut, uint32_t* wrappedDataSz);

int wh_Client_DataUnwrap(whClientContext* ctx, enum wc_CipherType cipherType,
                         uint16_t serverKeyId, void* wrappedDataIn,
                         uint32_t wrappedDataInSz, void* dataOut,
                         uint32_t* dataInOutSz);

int wh_Client_DataUnwrapRequest(whClientContext*   ctx,
                                enum wc_CipherType cipherType,
                                uint16_t serverKeyId, void* wrappedDataIn,
                                uint32_t wrappedDataInSz);

int wh_Client_DataUnwrapResponse(whClientContext*   ctx,
                                 enum wc_CipherType cipherType, void* dataOut,
                                 uint32_t* dataSz);

/* Counter functions */
int wh_Client_CounterInitRequest(whClientContext* c, whNvmId counterId,
    uint32_t counter);
int wh_Client_CounterInitResponse(whClientContext* c, uint32_t* counter);
int wh_Client_CounterInit(whClientContext* c, whNvmId counterId,
    uint32_t* counter);
```

```
int wh_Client_CounterResetRequest(whClientContext* c, whNvmId counterId);
int wh_Client_CounterResetResponse(whClientContext* c, uint32_t* counter);
int wh_Client_CounterReset(whClientContext* c, whNvmId counterId,
    uint32_t* counter);

int wh_Client_CounterIncrementRequest(whClientContext* c, whNvmId counterId);
int wh_Client_CounterIncrementResponse(whClientContext* c, uint32_t* counter);
int wh_Client_CounterIncrement(whClientContext* c, whNvmId counterId,
    uint32_t* counter);

int wh_Client_CounterReadRequest(whClientContext* c, whNvmId counterId);
int wh_Client_CounterReadResponse(whClientContext* c, uint32_t* counter);
int wh_Client_CounterRead(whClientContext* c, whNvmId counterId,
    uint32_t* counter);

int wh_Client_CounterDestroyRequest(whClientContext* c, whNvmId counterId);
int wh_Client_CounterDestroyResponse(whClientContext* c);
int wh_Client_CounterDestroy(whClientContext* c, whNvmId counterId);

int wh_Client_NvmInitRequest(whClientContext* c);

int wh_Client_NvmInitResponse(whClientContext* c, int32_t* out_rc,
                              uint32_t* out_clientnvm_id,
                              uint32_t* out_servernvm_id);

int wh_Client_NvmInit(whClientContext* c, int32_t* out_rc,
                      uint32_t* out_clientnvm_id, uint32_t* out_servernvm_id);

int wh_Client_NvmCleanupRequest(whClientContext* c);

int wh_Client_NvmCleanupResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_NvmCleanup(whClientContext* c, int32_t* out_rc);

int wh_Client_NvmGetAvailableRequest(whClientContext* c);

int wh_Client_NvmGetAvailableResponse(whClientContext* c, int32_t* out_rc,
                                      uint32_t* out_avail_size,
                                      whNvmId*  out_avail_objects,
                                      uint32_t* out_reclaim_size,
                                      whNvmId*  out_reclaim_objects);

int wh_Client_NvmGetAvailable(whClientContext* c, int32_t* out_rc,
                              uint32_t* out_avail_size,
                              whNvmId*  out_avail_objects,
                              uint32_t* out_reclaim_size,
                              whNvmId*  out_reclaim_objects);

int wh_Client_NvmAddObjectRequest(whClientContext* c, whNvmId id,
                                  whNvmAccess access, whNvmFlags flags,
                                  whNvmSize label_len, uint8_t* label,
                                  whNvmSize len, const uint8_t* data);
```

```
int wh_Client_NvmAddObjectResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_NvmAddObject(whClientContext* c, whNvmId id, whNvmAccess access,
                           whNvmFlags flags, whNvmSize label_len,
                           uint8_t* label, whNvmSize len, const uint8_t* data,
                           int32_t* out_rc);

int wh_Client_NvmListRequest(whClientContext* c, whNvmAccess access,
                             whNvmFlags flags, whNvmId start_id);

int wh_Client_NvmListResponse(whClientContext* c, int32_t* out_rc,
                              whNvmId* out_count, whNvmId* out_id);

int wh_Client_NvmList(whClientContext* c, whNvmAccess access, whNvmFlags flags,
                      whNvmId start_id, int32_t* out_rc, whNvmId* out_count,
                      whNvmId* out_id);

int wh_Client_NvmGetMetadataRequest(whClientContext* c, whNvmId id);

int wh_Client_NvmGetMetadataResponse(whClientContext* c, int32_t* out_rc,
                                     whNvmId* out_id, whNvmAccess* out_access,
                                     whNvmFlags* out_flags, whNvmSize* out_len,
                                     whNvmSize label_len, uint8_t* label);

int wh_Client_NvmGetMetadata(whClientContext* c, whNvmId id, int32_t* out_rc,
                             whNvmId* out_id, whNvmAccess* out_access,
                             whNvmFlags* out_flags, whNvmSize* out_len,
                             whNvmSize label_len, uint8_t* label);

int wh_Client_NvmDestroyObjectsRequest(whClientContext* c, whNvmId list_count,
                                       const whNvmId* id_list);

int wh_Client_NvmDestroyObjectsResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_NvmDestroyObjects(whClientContext* c, whNvmId list_count,
                                const whNvmId* id_list, int32_t* out_rc);

int wh_Client_NvmReadRequest(whClientContext* c, whNvmId id, whNvmSize offset,
                             whNvmSize data_len);

int wh_Client_NvmReadResponse(whClientContext* c, int32_t* out_rc,
                              whNvmSize* out_len, uint8_t* data);

int wh_Client_NvmRead(whClientContext* c, whNvmId id, whNvmSize offset,
                      whNvmSize data_len, int32_t* out_rc, whNvmSize* out_len,
                      uint8_t* data);


int wh_Client_NvmAddObjectDmaRequest(whClientContext* c,
                                     whNvmMetadata*   metadata,
                                     whNvmSize data_len, const uint8_t* data);

int wh_Client_NvmAddObjectDmaResponse(whClientContext* c, int32_t* out_rc);
```

```
int wh_Client_NvmAddObjectDma(whClientContext* c, whNvmMetadata* metadata,
                              whNvmSize data_len, const uint8_t* data,
                              int32_t* out_rc);

/*
 * @brief Sends a request to the server to read data from non-volatile memory
 * (NVM) using DMA, with automatic detection of client address width (32-bit or
 * 64-bit).
 *
 * This function prepares and sends a request to the server to read data from
 * NVM using DMA. The client address width (32-bit or 64-bit) is automatically
 * detected. The request includes the NVM ID, offset, length of the data, and
 * the data client address. This function does not block; it returns immediately
 * after sending the request.
 *
 * @param[in] c Pointer to the client context.
 * @param[in] id The NVM ID of the object to read.
 * @param[in] offset The offset within the object to start reading from.
 * @param[in] data_len The length of the data to be read.
 * @param[in] data Pointer to the data buffer where the data will be read into.
 * @return int Returns 0 on success, or a negative error code on failure.
 */
int wh_Client_NvmReadDmaRequest(whClientContext* c, whNvmId id,
                                whNvmSize offset, whNvmSize data_len,
                                uint8_t* data);

int wh_Client_NvmReadDmaResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_NvmReadDma(whClientContext* c, whNvmId id, whNvmSize offset,
                         whNvmSize data_len, uint8_t* data, int32_t* out_rc);

/* Client custom-callback support */

int wh_Client_CustomCbRequest(whClientContext*                 c,
                              const whMessageCustomCb_Request* req);

int wh_Client_CustomCbResponse(whClientContext*            c,
                               whMessageCustomCb_Response* resp);

int wh_Client_CustomCheckRegisteredRequest(whClientContext* c, uint32_t id);

int wh_Client_CustomCbCheckRegisteredResponse(whClientContext* c,
                                              uint16_t*        outId,
                                              int*             responseError);

int wh_Client_CustomCbCheckRegistered(whClientContext* c, uint16_t id,
                                      int* responseError);

/* Auth Manager functions */

int wh_Client_AuthLoginRequest(whClientContext* c, whAuthMethod method,
                               const char* username, const void* auth_data,
                               uint16_t auth_data_len);
```

```
int wh_Client_AuthLoginResponse(whClientContext* c, int32_t* out_rc,
                                whUserId* out_user_id);

int wh_Client_AuthLogin(whClientContext* c, whAuthMethod method,
                        const char* username, const void* auth_data,
                        uint16_t auth_data_len, int32_t* out_rc,
                        whUserId* out_user_id);

int wh_Client_AuthLogoutRequest(whClientContext* c, whUserId user_id);

int wh_Client_AuthLogoutResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_AuthLogout(whClientContext* c, whUserId user_id, int32_t*
 ↪  out_rc);

int wh_Client_AuthUserAddResponse(whClientContext* c, int32_t* out_rc,
                                  whUserId* out_user_id);

int wh_Client_AuthUserAddRequest(whClientContext* c, const char* username,
                                 whAuthPermissions permissions,
                                 whAuthMethod method, const void* credentials,
                                 uint16_t credentials_len);

int wh_Client_AuthUserAdd(whClientContext* c, const char* username,
                          whAuthPermissions permissions, whAuthMethod method,
                          const void* credentials, uint16_t credentials_len,
                          int32_t* out_rc, whUserId* out_user_id);

int wh_Client_AuthUserGetRequest(whClientContext* c, const char* username);

int wh_Client_AuthUserGetResponse(whClientContext* c, int32_t* out_rc,
                                  whUserId*          out_user_id,
                                  whAuthPermissions* out_permissions);

int wh_Client_AuthUserGet(whClientContext* c, const char* username,
                          int32_t* out_rc, whUserId* out_user_id,
                          whAuthPermissions* out_permissions);

int wh_Client_AuthUserDeleteRequest(whClientContext* c, whUserId user_id);

int wh_Client_AuthUserDeleteResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_AuthUserDelete(whClientContext* c, whUserId user_id,
                             int32_t* out_rc);

int wh_Client_AuthUserSetPermissionsRequest(whClientContext*  c,
                                            whUserId          user_id,
                                            whAuthPermissions permissions);

int wh_Client_AuthUserSetPermissionsResponse(whClientContext* c,
                                             int32_t*         out_rc);

int wh_Client_AuthUserSetPermissions(whClientContext* c, whUserId user_id,
                                     whAuthPermissions permissions,
```

```
                                    int32_t*          out_rc);

int wh_Client_AuthUserSetCredentialsRequest(
    whClientContext* c, whUserId user_id, whAuthMethod method,
    const void* current_credentials, uint16_t current_credentials_len,
    const void* new_credentials, uint16_t new_credentials_len);

int wh_Client_AuthUserSetCredentialsResponse(whClientContext* c,
                                             int32_t*         out_rc);

int wh_Client_AuthUserSetCredentials(
    whClientContext* c, whUserId user_id, whAuthMethod method,
    const void* current_credentials, uint16_t current_credentials_len,
    const void* new_credentials, uint16_t new_credentials_len, int32_t*
    ↪  out_rc);
/* Certificate functions */

int wh_Client_CertInitRequest(whClientContext* c);

int wh_Client_CertInitResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertInit(whClientContext* c, int32_t* out_rc);

int wh_Client_CertAddTrustedRequest(whClientContext* c, whNvmId id,
                                    whNvmAccess access, whNvmFlags flags,
                                    uint8_t* label, whNvmSize label_len,
                                    const uint8_t* cert, uint32_t cert_len);

int wh_Client_CertAddTrustedResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertAddTrusted(whClientContext* c, whNvmId id, whNvmAccess
 ↪  access,
                             whNvmFlags flags, uint8_t* label,
                             whNvmSize label_len, const uint8_t* cert,
                             uint32_t cert_len, int32_t* out_rc);

int wh_Client_CertEraseTrustedRequest(whClientContext* c, whNvmId id);

int wh_Client_CertEraseTrustedResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertEraseTrusted(whClientContext* c, whNvmId id, int32_t*
 ↪  out_rc);

int wh_Client_CertReadTrustedRequest(whClientContext* c, whNvmId id,
                                     uint32_t cert_len);

int wh_Client_CertReadTrustedResponse(whClientContext* c, uint8_t* cert,
                                      uint32_t* cert_len, int32_t* out_rc);

int wh_Client_CertReadTrusted(whClientContext* c, whNvmId id, uint8_t* cert,
                              uint32_t* cert_len, int32_t* out_rc);

int wh_Client_CertVerifyRequest(whClientContext* c, const uint8_t* cert,
                                uint32_t cert_len, whNvmId trustedRootNvmId);
```

```
int wh_Client_CertVerifyResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertVerify(whClientContext* c, const uint8_t* cert,
                         uint32_t cert_len, whNvmId trustedRootNvmId,
                         int32_t* out_rc);

int wh_Client_CertVerifyAndCacheLeafPubKeyRequest(
    whClientContext* c, const uint8_t* cert, uint32_t cert_len,
    whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId keyId);

int wh_Client_CertVerifyAndCacheLeafPubKeyResponse(whClientContext* c,
                                                   whKeyId*         out_keyId,
                                                   int32_t*         out_rc);

int wh_Client_CertVerifyAndCacheLeafPubKey(
    whClientContext* c, const uint8_t* cert, uint32_t cert_len,
    whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId* inout_keyId,
    int32_t* out_rc);


int wh_Client_CertVerifyMultiRootRequest(whClientContext* c,
                                         const uint8_t* cert, uint32_t cert_len,
                                          const whNvmId* trustedRootNvmIds,
                                          uint16_t       numRoots);

int wh_Client_CertVerifyMultiRootResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertVerifyMultiRoot(whClientContext* c, const uint8_t* cert,
                                  uint32_t       cert_len,
                                  const whNvmId* trustedRootNvmIds,
                                  uint16_t numRoots, int32_t* out_rc);

int wh_Client_CertVerifyMultiRootAndCacheLeafPubKeyRequest(
    whClientContext* c, const uint8_t* cert, uint32_t cert_len,
    const whNvmId* trustedRootNvmIds, uint16_t numRoots,
    whNvmFlags cachedKeyFlags, whKeyId keyId);

int wh_Client_CertVerifyMultiRootAndCacheLeafPubKeyResponse(whClientContext*
 ↪  c,
                                                            whKeyId* out_keyId,
                                                             int32_t* out_rc);

int wh_Client_CertVerifyMultiRootAndCacheLeafPubKey(
    whClientContext* c, const uint8_t* cert, uint32_t cert_len,
    const whNvmId* trustedRootNvmIds, uint16_t numRoots,
    whNvmFlags cachedKeyFlags, whKeyId* inout_keyId, int32_t* out_rc);

#ifdef WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE
int wh_Client_CertVerifyCacheClearRequest(whClientContext* c);

int wh_Client_CertVerifyCacheClearResponse(whClientContext* c, int32_t*
 ↪  out_rc);
```

```
int wh_Client_CertVerifyCacheClear(whClientContext* c, int32_t* out_rc);

int wh_Client_CertVerifyCacheSetEnabledRequest(whClientContext* c,
                                               uint8_t          enable);

int wh_Client_CertVerifyCacheSetEnabledResponse(whClientContext* c,
                                                int32_t*         out_rc);

int wh_Client_CertVerifyCacheSetEnabled(whClientContext* c, uint8_t enable,
                                        int32_t* out_rc);
#endif /* WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE */


#ifdef WOLFHSM_CFG_DMA

int wh_Client_CertAddTrustedDmaRequest(whClientContext* c, whNvmId id,
                                       whNvmAccess access, whNvmFlags flags,
                                       uint8_t* label, whNvmSize label_len,
                                       const void* cert, uint32_t cert_len);

int wh_Client_CertAddTrustedDmaResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertAddTrustedDma(whClientContext* c, whNvmId id,
                                whNvmAccess access, whNvmFlags flags,
                                uint8_t* label, whNvmSize label_len,
                                const void* cert, uint32_t cert_len,
                                int32_t* out_rc);

int wh_Client_CertReadTrustedDmaRequest(whClientContext* c, whNvmId id,
                                        void* cert, uint32_t cert_len);

int wh_Client_CertReadTrustedDmaResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertReadTrustedDma(whClientContext* c, whNvmId id, void* cert,
                                 uint32_t cert_len, int32_t* out_rc);

int wh_Client_CertVerifyDmaRequest(whClientContext* c, const void* cert,
                                   uint32_t cert_len, whNvmId trustedRootNvmId);

int wh_Client_CertVerifyDmaResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertVerifyDma(whClientContext* c, const void* cert,
                            uint32_t cert_len, whNvmId trustedRootNvmId,
                            int32_t* out_rc);


int wh_Client_CertVerifyDmaAndCacheLeafPubKeyRequest(
    whClientContext* c, const void* cert, uint32_t cert_len,
    whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId keyId);

int wh_Client_CertVerifyDmaAndCacheLeafPubKeyResponse(whClientContext* c,
                                                      whKeyId* out_keyId,
                                                      int32_t* out_rc);
```

```
int wh_Client_CertVerifyDmaAndCacheLeafPubKey(
    whClientContext* c, const void* cert, uint32_t cert_len,
    whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId* inout_keyId,
    int32_t* out_rc);


int wh_Client_CertVerifyMultiRootDmaRequest(whClientContext* c,
                                            const void* cert, uint32_t cert_len,
                                             const whNvmId* trustedRootNvmIds,
                                             uint16_t       numRoots);

int wh_Client_CertVerifyMultiRootDmaResponse(whClientContext* c,
                                              int32_t*         out_rc);

int wh_Client_CertVerifyMultiRootDma(whClientContext* c, const void* cert,
                                     uint32_t       cert_len,
                                     const whNvmId* trustedRootNvmIds,
                                     uint16_t numRoots, int32_t* out_rc);

int wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKeyRequest(
    whClientContext* c, const void* cert, uint32_t cert_len,
    const whNvmId* trustedRootNvmIds, uint16_t numRoots,
    whNvmFlags cachedKeyFlags, whKeyId keyId);

int wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKeyResponse(
    whClientContext* c, whKeyId* out_keyId, int32_t* out_rc);

int wh_Client_CertVerifyMultiRootDmaAndCacheLeafPubKey(
    whClientContext* c, const void* cert, uint32_t cert_len,
    const whNvmId* trustedRootNvmIds, uint16_t numRoots,
    whNvmFlags cachedKeyFlags, whKeyId* inout_keyId, int32_t* out_rc);


#endif /* WOLFHSM_CFG_DMA */

int wh_Client_CertVerifyAcertRequest(whClientContext* c, const void* cert,
                                     uint32_t cert_len,
                                     whNvmId  trustedRootNvmId);

int wh_Client_CertVerifyAcertResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertVerifyAcert(whClientContext* c, const void* cert,
                              uint32_t cert_len, whNvmId trustedRootNvmId,
                              int32_t* out_rc);

int wh_Client_CertVerifyAcertDmaRequest(whClientContext* c, const void* cert,
                                        uint32_t cert_len,
                                        whNvmId  trustedRootNvmId);

int wh_Client_CertVerifyAcertDmaResponse(whClientContext* c, int32_t* out_rc);

#if defined(WOLFHSM_CFG_DMA)
int wh_Client_DmaRegisterAllowList(struct whClientContext_t* client,
                                   const whDmaAddrAllowList* allowlist);
```

```
int wh_Client_DmaRegisterCb(struct whClientContext_t* client,
                            whClientDmaClientMemCb    cb);


int wh_Client_DmaProcessClientAddress(struct whClientContext_t* client,
                                      uintptr_t clientAddr, void** serverPtr,
                                      size_t len, whDmaOper oper,
                                      whDmaFlags flags);


int wh_Client_DmaAsyncPre(struct whClientContext_t* client,
                          whClientDmaAsyncBuf* buf, uintptr_t clientAddr,
                          uint64_t len, whDmaOper preOper,
                          uintptr_t* outXformedAddr);

int wh_Client_DmaAsyncPost(struct whClientContext_t* client,
                           whClientDmaAsyncBuf*       buf);


int wh_Client_CertVerifyAcertDma(whClientContext* c, const void* cert,
                                 uint32_t cert_len, whNvmId trustedRootNvmId,
                                 int32_t* out_rc);

#endif /* WOLFHSM_CFG_DMA */


#define WH_CLIENT_KEYID_MAKE_GLOBAL(_id) ((_id) | WH_KEYID_CLIENT_GLOBAL_FLAG)

#define WH_CLIENT_KEYID_MAKE_WRAPPED(_id) ((_id) |
 ↪  WH_KEYID_CLIENT_WRAPPED_FLAG)

#define WH_CLIENT_KEYID_MAKE_WRAPPED_GLOBAL(_id) \
    ((_id) | WH_KEYID_CLIENT_GLOBAL_FLAG | WH_KEYID_CLIENT_WRAPPED_FLAG)

#define WH_CLIENT_KEYID_MAKE_HW(_id) ((_id) | WH_KEYID_CLIENT_HW_FLAG)

#define WH_CLIENT_KEYID_MAKE_WRAPPED_META(_clientId, _id) \
    WH_MAKE_KEYID(WH_KEYTYPE_WRAPPED, (_clientId), (_id))

#endif /* !WOLFHSM_WH_CLIENT_H_ */
```

A.2 wolfhsm/wh_client_crypto.h

A.2.1 Functions

	Name
int	**wh_Client_RngGenerate**(whClientContext * ctx, uint8_t * out, uint32_t size)Generate random bytes.
int	**wh_Client_RngGenerateRequest**(whClientContext * ctx, uint32_t size)Async request half of a non-DMA RNG generate.

	Name
int	**wh_Client_RngGenerateResponse**(whClientContext * ctx, uint8_t * out, uint32_t * inout_size)Async response half of a non-DMA RNG generate.
int	**wh_Client_RngGenerateDma**(whClientContext * ctx, uint8_t * out, uint32_t size)Generate random bytes using DMA.
int	**wh_Client_RngGenerateDmaRequest**(whClientContext * ctx, uint8_t * out, uint32_t size)Async request half of a DMA RNG generate.
int	**wh_Client_RngGenerateDmaResponse**(whClientContext * ctx)Async response half of a DMA RNG generate.
int	**wh_Client_Curve25519SetKeyId**(curve25519_key * key, whKeyId keyId)Associates a Curve25519 key with a specific key ID.
int	**wh_Client_Curve25519GetKeyId**(curve25519_key * key, whKeyId * outId)Gets the wolfHSM keyId being used by the wolfCrypt struct.
int	**wh_Client_Curve25519ImportKey**(whClientContext * ctx, curve25519_key * key, whKeyId * inout_keyId, whNvmFlags flags, uint16_t label_len, uint8_t * label)Imports wolfCrypt Curve25519 key as a raw byte array into the wolfHSM server key cache.
int	**wh_Client_Curve25519ExportKey**(whClientContext * ctx, whKeyId keyId, curve25519_key * key, uint16_t label_len, uint8_t * label)Exports a serialized curve25519 key from the wolfHSM server keycache and decodes it into the wolfCrypt curve25519 key structure.
int	**wh_Client_Curve25519ExportPublicKey**(whClientContext * ctx, whKeyId keyId, curve25519_key * key, uint16_t label_len, uint8_t * label)Exports only the public part of a cached Curve25519 key.
int	**wh_Client_Curve25519MakeCacheKey**(whClientContext * ctx, uint16_t size, whKeyId * inout_key_id, whNvmFlags flags, const uint8_t * label, uint16_t label_len)Generate a Curve25519 key in the server key cache.
int	**wh_Client_Curve25519MakeExportKey**(whClientContext * ctx, uint16_t size, curve25519_key * key)Generate a Curve25519 key by the server and export to the client.
int	**wh_Client_Curve25519SharedSecret**(whClientContext * ctx, curve25519_key * priv_key, curve25519_key * pub_key, int endian, uint8_t * out, uint16_t * out_size)Compute an X25519 shared secret using a public and private key.

	Name
int	**wh_Client_Curve25519SharedSecretRequest**(whClientContext * ctx, whKeyId prv_key_id, whKeyId pub_key_id, int endian)Send a request to compute an X25519 shared secret that will be returned to the client.
int	**wh_Client_Curve25519SharedSecretResponse**(whClientContex * ctx, uint8_t * out, uint16_t * out_size)Retrieve the response to a wh_Client_Curve25519SharedSecretRequest.
int	**wh_Client_Curve25519SharedSecretCacheKey**(whClientContex * ctx, curve25519_key * priv_key, curve25519_key * pub_key, int endian, whKeyId * inout_key_id, whNvmFlags flags, const uint8_t * label, uint16_t label_len)Compute an X25519 shared secret and cache it on the server.
int	**wh_Client_Curve25519SharedSecretCacheKeyRequest**(whClien * ctx, whKeyId prv_key_id, whKeyId pub_key_id, int endian, whKeyId out_key_id, whNvmFlags flags, const uint8_t * label, uint16_t label_len)Async variant: send the request half of SharedSecretCacheKey. Both input keys must already be cached.
int	**wh_Client_Curve25519SharedSecretCacheKeyResponse**(whCli * ctx, whKeyId * out_key_id)Async variant: retrieve the response of SharedSecretCacheKeyRequest. On success, *out_key_id receives the assigned cache id.
int	**wh_Client_EccSetKeyId**(ecc_key * key, whKeyId keyId)Associates a Ecc key with a specific key ID.
int	**wh_Client_EccGetKeyId**(ecc_key * key, whKeyId * outId)Gets the wolfHSM keyId being used by the wolfCrypt struct.
int	**wh_Client_EccImportKey**(whClientContext * ctx, ecc_key * key, whKeyId * inout_keyId, whNvmFlags flags, uint16_t label_len, uint8_t * label)Imports a wolfCrypt ECC key as a DER-formatted blob into the wolfHSM server key cache.
int	**wh_Client_EccExportKey**(whClientContext * ctx, whKeyId keyId, ecc_key * key, uint16_t label_len, uint8_t * label)Exports a DER-formatted ECC key from the wolfHSM server keycache and decodes it into the wolfCrypt ECC key structure.
int	**wh_Client_EccExportPublicKey**(whClientContext * ctx, whKeyId keyId, ecc_key * key, uint16_t label_len, uint8_t * label)Exports only the public part of a cached ECC key.

	Name
int	**wh_Client_EccMakeExportKey**(whClientContext * ctx, int size, int curveId, ecc_key * key)Generate an ECC key pair on the server and export it to the client.
int	**wh_Client_EccMakeCacheKey**(whClientContext * ctx, int size, int curveId, whKeyId * inout_key_id, whNvmFlags flags, uint16_t label_len, uint8_t * label)Generate an ECC key pair in the server key cache.
int	**wh_Client_EccSharedSecret**(whClientContext * ctx, ecc_key * priv_key, ecc_key * pub_key, uint8_t * out, uint16_t * inout_size)Compute an ECDH shared secret using a public and private ECC key.
int	**wh_Client_EccSign**(whClientContext * ctx, ecc_key * key, const uint8_t * hash, uint16_t hash_len, uint8_t * sig, uint16_t * inout_sig_len)Generate an ECDSA signature of the provided hash on the server.
int	**wh_Client_EccVerify**(whClientContext * ctx, ecc_key * key, const uint8_t * sig, uint16_t sig_len, const uint8_t * hash, uint16_t hash_len, int * out_res)Verify an ECDSA signature of the provided hash on the server.
int	**wh_Client_EccSignRequest**(whClientContext * ctx, whKeyId keyId, const uint8_t * hash, uint16_t hash_len)Async request half of an ECC sign operation.
int	**wh_Client_EccSignResponse**(whClientContext * ctx, uint8_t * sig, uint16_t * inout_sig_len)Async response half of an ECC sign operation.
int	**wh_Client_EccVerifyRequest**(whClientContext * ctx, whKeyId keyId, const uint8_t * sig, uint16_t sig_len, const uint8_t * hash, uint16_t hash_len)Async request half of an ECC verify operation.
int	**wh_Client_EccVerifyResponse**(whClientContext * ctx, ecc_key * opt_key, int * out_res)Async response half of an ECC verify operation.
int	**wh_Client_EccSharedSecretRequest**(whClientContext * ctx, whKeyId prv_key_id, whKeyId pub_key_id)Async request half of an ECDH shared-secret operation.
int	**wh_Client_EccSharedSecretResponse**(whClientContext * ctx, uint8_t * out, uint16_t * inout_size)Async response half of an ECDH shared-secret operation.

	Name
int	**wh_Client_EccSharedSecretCacheKey**(whClientContext * ctx, ecc_key * priv_key, ecc_key * pub_key, whKeyId * inout_key_id, whNvmFlags flags, const uint8_t * label, uint16_t label_len)Compute an ECDH shared secret and cache it on the server.
int	**wh_Client_EccSharedSecretCacheKeyRequest**(whClientContext * ctx, whKeyId prv_key_id, whKeyId pub_key_id, whKeyId out_key_id, whNvmFlags flags, const uint8_t * label, uint16_t label_len)Async variant: send the request half of SharedSecretCacheKey. Both input keys must already be cached on the server.
int	**wh_Client_EccSharedSecretCacheKeyResponse**(whClientConte * ctx, whKeyId * out_key_id)Async variant: retrieve the response of SharedSecretCacheKeyRequest. On success, *out_key_id receives the assigned cache id.
int	**wh_Client_EccMakeCacheKeyRequest**(whClientContext * ctx, int size, int curveId, whKeyId key_id, whNvmFlags flags, uint16_t label_len, uint8_t * label)Async request half of an ECC server-side keygen that caches the new key in the server.
int	**wh_Client_EccMakeCacheKeyResponse**(whClientContext * ctx, whKeyId * out_key_id)Async response half of an ECC server-side keygen that caches the new key in the server.
int	**wh_Client_EccMakeExportKeyRequest**(whClientContext * ctx, int size, int curveId)Async request half of an ECC server-side keygen that exports the new key back to the client.
int	**wh_Client_EccMakeExportKeyResponse**(whClientContext * ctx, ecc_key * key)Async response half of an ECC server-side keygen that exports the new key back to the client.
int	**wh_Client_Ed25519SetKeyId**(ed25519_key * key, whKeyId keyId)Associates an Ed25519 key with a specific key ID.
int	**wh_Client_Ed25519GetKeyId**(ed25519_key * key, whKeyId * outId)Retrieves the key ID from an Ed25519 key device context.
int	**wh_Client_Ed25519ImportKey**(whClientContext * ctx, ed25519_key * key, whKeyId * inout_keyId, whNvmFlags flags, uint16_t label_len, uint8_t * label)Import an Ed25519 key into the server keystore/cache.
int	**wh_Client_Ed25519ExportKey**(whClientContext * ctx, whKeyId keyId, ed25519_key * key, uint16_t label_len, uint8_t * label)Export an Ed25519 key from the server to the client.

	Name
int	**wh_Client_Ed25519ExportPublicKey**(whClientContext * ctx, whKeyId keyId, ed25519_key * key, uint16_t label_len, uint8_t * label)Exports only the public part of a cached Ed25519 key.
int	**wh_Client_Ed25519MakeExportKey**(whClientContext * ctx, ed25519_key * key)Create a new Ed25519 key on the server and export it without caching.
int	**wh_Client_Ed25519MakeCacheKey**(whClientContext * ctx, whKeyId * inout_key_id, whNvmFlags flags, uint16_t label_len, uint8_t * label)Create a new Ed25519 key on the server and store it in cache/NVM.
int	**wh_Client_Ed25519Sign**(whClientContext * ctx, ed25519_key * key, const uint8_t * msg, uint32_t msgLen, uint8_t type, const uint8_t * context, uint32_t contextLen, uint8_t * sig, uint32_t * inout_sig_len)Sign a message using an Ed25519 key on the server.
int	**wh_Client_Ed25519Verify**(whClientContext * ctx, ed25519_key * key, const uint8_t * sig, uint32_t sigLen, const uint8_t * msg, uint32_t msgLen, uint8_t type, const uint8_t * context, uint32_t contextLen, int * out_res)Verify a message signature using an Ed25519 key on the server.
int	**wh_Client_Ed25519SignDma**(whClientContext * ctx, ed25519_key * key, const uint8_t * msg, uint32_t msgLen, uint8_t type, const uint8_t * context, uint32_t contextLen, uint8_t * sig, uint32_t * inout_sig_len)Sign a message using an Ed25519 key via DMA.
int	**wh_Client_Ed25519VerifyDma**(whClientContext * ctx, ed25519_key * key, const uint8_t * sig, uint32_t sigLen, const uint8_t * msg, uint32_t msgLen, uint8_t type, const uint8_t * context, uint32_t contextLen, int * out_res)Verify a signature using an Ed25519 key via DMA.
int	**wh_Client_RsaSetKeyId**(RsaKey * key, whNvmId keyId)Associates an RSA key with a specific key ID.
int	**wh_Client_RsaGetKeyId**(RsaKey * key, whNvmId * outId)Gets the wolfHSM keyId being used by the wolfCrypt struct.
int	**wh_Client_RsaImportKey**(whClientContext * ctx, const RsaKey * key, whKeyId * inout_keyId, whNvmFlags flags, uint32_t label_len, uint8_t * label)Imports wolfCrypt RSA key as a PCKS1 DER-formatted file into the wolfHSM server key cache.

	Name
int	**wh_Client_RsaExportKey**(whClientContext * ctx, whKeyId keyId, RsaKey * key, uint32_t label_len, uint8_t * label)Exports a PKCS1 DER-formated RSA key from the wolfHSM server keycache and decodes it into the wolfCrypt RSA key structure.
int	**wh_Client_RsaExportPublicKey**(whClientContext * ctx, whKeyId keyId, RsaKey * key, uint32_t label_len, uint8_t * label)Exports only the public part of a cached RSA key.
int	**wh_Client_RsaMakeExportKey**(whClientContext * ctx, uint32_t size, uint32_t e, RsaKey * rsa)
int	**wh_Client_RsaMakeCacheKey**(whClientContext * ctx, uint32_t size, uint32_t e, whKeyId * inout_key_id, whNvmFlags flags, uint32_t label_len, uint8_t * label)
int	**wh_Client_RsaFunction**(whClientContext * ctx, RsaKey * key, int rsa_type, const uint8_t * in, uint16_t in_len, uint8_t * out, uint16_t * inout_out_len)
int	**wh_Client_RsaGetSize**(whClientContext * ctx, const RsaKey * key, int * out_size)
int	**wh_Client_RsaFunctionRequest**(whClientContext * ctx, whKeyId keyId, int rsa_type, const uint8_t * in, uint16_t in_len, uint16_t out_capacity)Send an async RSA encrypt/decrypt/sign/verify request.
int	**wh_Client_RsaFunctionResponse**(whClientContext * ctx, uint8_t * out, uint16_t * inout_out_len)Receive the reply to an async RSA operation.
int	**wh_Client_RsaGetSizeRequest**(whClientContext * ctx, whKeyId keyId)Send an async RSA key-size query.
int	**wh_Client_RsaGetSizeResponse**(whClientContext * ctx, int * out_size)Receive the reply to an async RSA key-size query.
int	**wh_Client_RsaMakeCacheKeyRequest**(whClientContext * ctx, uint32_t size, uint32_t e, whKeyId key_id, whNvmFlags flags, uint32_t label_len, uint8_t * label)Ask the server to generate an RSA key and cache it.
int	**wh_Client_RsaMakeCacheKeyResponse**(whClientContext * ctx, whKeyId * out_key_id)Receive the reply to an async cache-keygen request.
int	**wh_Client_RsaMakeExportKeyRequest**(whClientContext * ctx, uint32_t size, uint32_t e)Ask the server to generate an RSA key and return it as DER.
int	**wh_Client_RsaMakeExportKeyResponse**(whClientContext * ctx, RsaKey * rsa)Receive the reply to an async export-keygen request.

	Name
int	**wh_Client_HkdfMakeCacheKey**(whClientContext * ctx, int hashType, whKeyId keyIdIn, const uint8_t * inKey, uint32_t inKeySz, const uint8_t * salt, uint32_t saltSz, const uint8_t * info, uint32_t infoSz, whKeyId * inout_key_id, whNvmFlags flags, const uint8_t * label, uint32_t label_len, uint32_t outSz)Generate HKDF output and store in the server key cache.
int	**wh_Client_HkdfMakeExportKey**(whClientContext * ctx, int hashType, whKeyId keyIdIn, const uint8_t * inKey, uint32_t inKeySz, const uint8_t * salt, uint32_t saltSz, const uint8_t * info, uint32_t infoSz, uint8_t * out, uint32_t outSz)Generate HKDF output and export to the client.
int	**wh_Client_CmacKdfMakeCacheKey**(whClientContext * ctx, whKeyId saltKeyId, const uint8_t * salt, uint32_t saltSz, whKeyId zKeyId, const uint8_t * z, uint32_t zSz, const uint8_t * fixedInfo, uint32_t fixedInfoSz, whKeyId * inout_key_id, whNvmFlags flags, const uint8_t * label, uint32_t label_len, uint32_t outSz)Generate CMAC two-step KDF output and store it in the server cache.
int	**wh_Client_CmacKdfMakeExportKey**(whClientContext * ctx, whKeyId saltKeyId, const uint8_t * salt, uint32_t saltSz, whKeyId zKeyId, const uint8_t * z, uint32_t zSz, const uint8_t * fixedInfo, uint32_t fixedInfoSz, uint8_t * out, uint32_t outSz)Generate CMAC two-step KDF output and export to the client.
int	**wh_Client_AesSetKeyId**(Aes * key, whNvmId keyId)Associates an AES key with a specific key ID.
int	**wh_Client_AesGetKeyId**(Aes * key, whNvmId * outId)Gets the wolfHSM keyId being used by the wolfCrypt struct.
int	**wh_Client_AesCtr**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-CTR operation.
int	**wh_Client_AesCtrDma**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-CTR operation using DMA.
int	**wh_Client_AesCtrRequest**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len)Send an AES-CTR encrypt/decrypt request to the server (non-blocking)
int	**wh_Client_AesCtrResponse**(whClientContext * ctx, Aes * aes, uint8_t * out, uint32_t * out_size)Receive the server's AES-CTR response (non-blocking)

	Name
int	**wh_Client_AesCtrDmaRequest**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Send an AES-CTR encrypt/decrypt DMA request to the server (non-blocking)
int	**wh_Client_AesCtrDmaResponse**(whClientContext * ctx, Aes * aes)Receive the server's AES-CTR DMA response (non-blocking)
int	**wh_Client_AesEcb**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-ECB operation.
int	**wh_Client_AesEcbDma**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-ECB operation using DMA.
int	**wh_Client_AesEcbRequest**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len)Send an AES-ECB encrypt/decrypt request to the server (non-blocking)
int	**wh_Client_AesEcbResponse**(whClientContext * ctx, Aes * aes, uint8_t * out, uint32_t * out_size)Receive the server's AES-ECB response (non-blocking)
int	**wh_Client_AesEcbDmaRequest**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Send an AES-ECB DMA request to the server (non-blocking)
int	**wh_Client_AesEcbDmaResponse**(whClientContext * ctx, Aes * aes)Receive the server's AES-ECB DMA response (non-blocking)
int	**wh_Client_AesCbc**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-CBC operation.
int	**wh_Client_AesCbcDma**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-CBC operation using DMA.
int	**wh_Client_AesCbcRequest**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len)Send an AES-CBC encrypt/decrypt request to the server (non-blocking)
int	**wh_Client_AesCbcResponse**(whClientContext * ctx, Aes * aes, uint8_t * out, uint32_t * out_size)Receive the server's AES-CBC response (non-blocking)
int	**wh_Client_AesCbcDmaRequest**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Send an AES-CBC DMA request to the server (non-blocking)
int	**wh_Client_AesCbcDmaResponse**(whClientContext * ctx, Aes * aes)Receive the server's AES-CBC DMA response (non-blocking)

	Name
int	**wh_Client_AesGcm**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, const uint8_t * iv, uint32_t iv_len, const uint8_t * authin, uint32_t authin_len, const uint8_t * dec_tag, uint8_t * enc_tag, uint32_t tag_len, uint8_t * out)Performs an AES-GCM operation.
int	**wh_Client_AesGcmDma**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, const uint8_t * iv, uint32_t iv_len, const uint8_t * authin, uint32_t authin_len, const uint8_t * dec_tag, uint8_t * enc_tag, uint32_t tag_len, uint8_t * out)Performs an AES-GCM operation using DMA.
int	**wh_Client_AesGcmRequest**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, const uint8_t * iv, uint32_t iv_len, const uint8_t * authin, uint32_t authin_len, const uint8_t * dec_tag, uint32_t tag_len)Send an AES-GCM encrypt/decrypt request to the server (non-blocking)
int	**wh_Client_AesGcmResponse**(whClientContext * ctx, Aes * aes, uint8_t * out, uint32_t out_capacity, uint32_t * out_size, uint8_t * enc_tag, uint32_t tag_len)Receive the server's AES-GCM response (non-blocking)
int	**wh_Client_AesGcmDmaRequest**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out, const uint8_t * iv, uint32_t iv_len, const uint8_t * authin, uint32_t authin_len, const uint8_t * dec_tag, uint32_t tag_len)Send an AES-GCM DMA request to the server (non-blocking)
int	**wh_Client_AesGcmDmaResponse**(whClientContext * ctx, Aes * aes, uint8_t * enc_tag, uint32_t tag_len)Receive the server's AES-GCM DMA response (non-blocking)
int	**wh_Client_Cmac**(whClientContext * ctx, Cmac * cmac, CmacType type, const uint8_t * key, uint32_t keyLen, const uint8_t * in, uint32_t inLen, uint8_t * outMac, uint32_t * outMacLen)Performs a CMAC operation on the input data.
int	**wh_Client_CmacGenerateRequest**(whClientContext * ctx, Cmac * cmac, CmacType type, const uint8_t * key, uint32_t keyLen, const uint8_t * in, uint32_t inLen, uint32_t outMacLen)Async request half of a non-DMA CMAC oneshot generate.
int	**wh_Client_CmacGenerateResponse**(whClientContext * ctx, Cmac * cmac, uint8_t * outMac, uint32_t * outMacLen)Async response half of a non-DMA CMAC oneshot generate.

	Name
int	**wh_Client_CmacUpdateRequest**(whClientContext * ctx, Cmac * cmac, CmacType type, const uint8_t * key, uint32_t keyLen, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a non-DMA CMAC streaming Update.
int	**wh_Client_CmacUpdateResponse**(whClientContext * ctx, Cmac * cmac)Async response half of a non-DMA CMAC streaming Update.
int	**wh_Client_CmacFinalRequest**(whClientContext * ctx, Cmac * cmac)Async request half of a non-DMA CMAC streaming Final.
int	**wh_Client_CmacFinalResponse**(whClientContext * ctx, Cmac * cmac, uint8_t * outMac, uint32_t * outMacLen)Async response half of a non-DMA CMAC streaming Final.
int	**wh_Client_CmacSetKeyId**(Cmac * key, whNvmId keyId)Associates a CMAC key with a specific key ID.
int	**wh_Client_CmacGetKeyId**(Cmac * key, whNvmId * outId)Gets the wolfHSM keyId being used by the wolfCrypt struct.
int	**wh_Client_CmacDma**(whClientContext * ctx, Cmac * cmac, CmacType type, const uint8_t * key, uint32_t keyLen, const uint8_t * in, uint32_t inLen, uint8_t * outMac, uint32_t * outMacLen)Performs CMAC operations using DMA for data transfer.
int	**wh_Client_CmacGenerateDmaRequest**(whClientContext * ctx, Cmac * cmac, CmacType type, const uint8_t * key, uint32_t keyLen, const uint8_t * in, uint32_t inLen, uint32_t outMacLen)Async request half of a DMA CMAC oneshot generate.
int	**wh_Client_CmacGenerateDmaResponse**(whClientContext * ctx, Cmac * cmac, uint8_t * outMac, uint32_t * outMacLen)Async response half of a DMA CMAC oneshot generate.
int	**wh_Client_CmacDmaUpdateRequest**(whClientContext * ctx, Cmac * cmac, CmacType type, const uint8_t * key, uint32_t keyLen, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a DMA CMAC streaming Update.
int	**wh_Client_CmacDmaUpdateResponse**(whClientContext * ctx, Cmac * cmac)Async response half of a DMA CMAC streaming Update.
int	**wh_Client_CmacDmaFinalRequest**(whClientContext * ctx, Cmac * cmac)Async request half of a DMA CMAC streaming Final.
int	**wh_Client_CmacDmaFinalResponse**(whClientContext * ctx, Cmac * cmac, uint8_t * outMac, uint32_t * outMacLen)Async response half of a DMA CMAC streaming Final.

	Name
int	**wh_Client_Sha256**(whClientContext * ctx, wc_Sha256 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-256 hash operation on the input data.
int	**wh_Client_Sha256UpdateRequest**(whClientContext * ctx, wc_Sha256 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a non-DMA SHA-256 Update.
int	**wh_Client_Sha256UpdateResponse**(whClientContext * ctx, wc_Sha256 * sha)Async response half of a non-DMA SHA-256 Update.
int	**wh_Client_Sha256FinalRequest**(whClientContext * ctx, wc_Sha256 * sha)Async request half of a non-DMA SHA-256 Final.
int	**wh_Client_Sha256FinalResponse**(whClientContext * ctx, wc_Sha256 * sha, uint8_t * out)Async response half of a non-DMA SHA-256 Final.
int	**wh_Client_Sha256Dma**(whClientContext * ctx, wc_Sha256 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-256 hash operation on the input data using DMA.
int	**wh_Client_Sha256DmaUpdateRequest**(whClientContext * ctx, wc_Sha256 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a DMA SHA-256 Update.
int	**wh_Client_Sha256DmaUpdateResponse**(whClientContext * ctx, wc_Sha256 * sha)Async response half of a DMA SHA-256 Update.
int	**wh_Client_Sha256DmaFinalRequest**(whClientContext * ctx, wc_Sha256 * sha)Async request half of a DMA SHA-256 Final.
int	**wh_Client_Sha256DmaFinalResponse**(whClientContext * ctx, wc_Sha256 * sha, uint8_t * out)Async response half of a DMA SHA-256 Final.
int	**wh_Client_Sha224**(whClientContext * ctx, wc_Sha224 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-224 hash operation on the input data.
int	**wh_Client_Sha224UpdateRequest**(whClientContext * ctx, wc_Sha224 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a non-DMA SHA-224 Update.
int	**wh_Client_Sha224UpdateResponse**(whClientContext * ctx, wc_Sha224 * sha)Async response half of a non-DMA SHA-224 Update.
int	**wh_Client_Sha224FinalRequest**(whClientContext * ctx, wc_Sha224 * sha)Async request half of a non-DMA SHA-224 Final.
int	**wh_Client_Sha224FinalResponse**(whClientContext * ctx, wc_Sha224 * sha, uint8_t * out)Async response half of a non-DMA SHA-224 Final.

	Name
int	**wh_Client_Sha224Dma**(whClientContext * ctx, wc_Sha224 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-224 hash operation on the input data using DMA.
int	**wh_Client_Sha224DmaUpdateRequest**(whClientContext * ctx, wc_Sha224 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)
int	**wh_Client_Sha224DmaUpdateResponse**(whClientContext * ctx, wc_Sha224 * sha)
int	**wh_Client_Sha224DmaFinalRequest**(whClientContext * ctx, wc_Sha224 * sha)
int	**wh_Client_Sha224DmaFinalResponse**(whClientContext * ctx, wc_Sha224 * sha, uint8_t * out)
int	**wh_Client_Sha384**(whClientContext * ctx, wc_Sha384 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-384 hash operation on the input data.
int	**wh_Client_Sha384UpdateRequest**(whClientContext * ctx, wc_Sha384 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a non-DMA SHA-384 Update.
int	**wh_Client_Sha384UpdateResponse**(whClientContext * ctx, wc_Sha384 * sha)Async response half of a non-DMA SHA-384 Update.
int	**wh_Client_Sha384FinalRequest**(whClientContext * ctx, wc_Sha384 * sha)Async request half of a non-DMA SHA-384 Final.
int	**wh_Client_Sha384FinalResponse**(whClientContext * ctx, wc_Sha384 * sha, uint8_t * out)Async response half of a non-DMA SHA-384 Final.
int	**wh_Client_Sha384Dma**(whClientContext * ctx, wc_Sha384 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-384 hash operation on the input data using DMA.
int	**wh_Client_Sha384DmaUpdateRequest**(whClientContext * ctx, wc_Sha384 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)
int	**wh_Client_Sha384DmaUpdateResponse**(whClientContext * ctx, wc_Sha384 * sha)
int	**wh_Client_Sha384DmaFinalRequest**(whClientContext * ctx, wc_Sha384 * sha)
int	**wh_Client_Sha384DmaFinalResponse**(whClientContext * ctx, wc_Sha384 * sha, uint8_t * out)
int	**wh_Client_Sha512**(whClientContext * ctx, wc_Sha512 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-512 hash operation on the input data.
int	**wh_Client_Sha512UpdateRequest**(whClientContext * ctx, wc_Sha512 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a non-DMA SHA-512 Update.

	Name
int	**wh_Client_Sha512UpdateResponse**(whClientContext * ctx, wc_Sha512 * sha)Async response half of a non-DMA SHA-512 Update.
int	**wh_Client_Sha512FinalRequest**(whClientContext * ctx, wc_Sha512 * sha)Async request half of a non-DMA SHA-512 Final.
int	**wh_Client_Sha512FinalResponse**(whClientContext * ctx, wc_Sha512 * sha, uint8_t * out)Async response half of a non-DMA SHA-512 Final.
int	**wh_Client_Sha512Dma**(whClientContext * ctx, wc_Sha512 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-512 hash operation on the input data using DMA.
int	**wh_Client_Sha512DmaUpdateRequest**(whClientContext * ctx, wc_Sha512 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)
int	**wh_Client_Sha512DmaUpdateResponse**(whClientContext * ctx, wc_Sha512 * sha)
int	**wh_Client_Sha512DmaFinalRequest**(whClientContext * ctx, wc_Sha512 * sha)
int	**wh_Client_Sha512DmaFinalResponse**(whClientContext * ctx, wc_Sha512 * sha, uint8_t * out)
int	**wh_Client_MlDsaSetKeyId**(wc_MlDsaKey * key, whKeyId keyId)Associates a ML-DSA key with a specific key ID.
int	**wh_Client_MlDsaGetKeyId**(wc_MlDsaKey * key, whKeyId * outId)Gets the wolfHSM keyId being used by the wolfCrypt struct.
int	**wh_Client_MlDsaImportKey**(whClientContext * ctx, wc_MlDsaKey * key, whKeyId * inout_keyId, whNvmFlags flags, uint16_t label_len, uint8_t * label)Import a ML-DSA key to the server key cache.
int	**wh_Client_MlDsaExportKey**(whClientContext * ctx, whKeyId keyId, wc_MlDsaKey * key, uint16_t label_len, uint8_t * label)Export a ML-DSA key from the server.
int	**wh_Client_MlDsaExportPublicKey**(whClientContext * ctx, whKeyId keyId, wc_MlDsaKey * key, uint16_t label_len, uint8_t * label)Exports only the public part of a cached ML-DSA key.
int	**wh_Client_MlDsaMakeExportKey**(whClientContext * ctx, int level, int size, wc_MlDsaKey * key)Generate a new ML-DSA key pair and export the public key.
int	**wh_Client_MlDsaMakeCacheKey**(whClientContext * ctx, int size, int level, whKeyId * inout_key_id, whNvmFlags flags, uint16_t label_len, uint8_t * label)Create and cache a new ML-DSA key on the server.

	Name
int	**wh_Client_MlDsaSign**(whClientContext * ctx, const byte * in, word32 in_len, byte * out, word32 * out_len, wc_MlDsaKey * key, const byte * context, byte contextLen, word32 preHashType)Sign a message using a ML-DSA private key.
int	**wh_Client_MlDsaVerify**(whClientContext * ctx, const byte * sig, word32 sig_len, const byte * msg, word32 msg_len, int * res, wc_MlDsaKey * key, const byte * context, byte contextLen, word32 preHashType)Verify a ML-DSA signature.
int	**wh_Client_MlDsaCheckPrivKey**(whClientContext * ctx, wc_MlDsaKey * key, const byte * pubKey, word32 pubKeySz)Check a ML-DSA private key.
int	**wh_Client_MlDsaImportKeyDma**(whClientContext * ctx, wc_MlDsaKey * key, whKeyId * inout_keyId, whNvmFlags flags, uint16_t label_len, uint8_t * label)Import a ML-DSA key using DMA.
int	**wh_Client_MlDsaExportKeyDma**(whClientContext * ctx, whKeyId keyId, wc_MlDsaKey * key, uint16_t label_len, uint8_t * label)Export a ML-DSA key using DMA.
int	**wh_Client_MlDsaExportPublicKeyDma**(whClientContext * ctx, whKeyId keyId, wc_MlDsaKey * key, uint16_t label_len, uint8_t * label)Export only the public part of a cached ML-DSA key using DMA.
int	**wh_Client_MlDsaMakeExportKeyDma**(whClientContext * ctx, int level, wc_MlDsaKey * key)Generate a new ML-DSA key pair and export it using DMA.
int	**wh_Client_MlDsaSignDma**(whClientContext * ctx, const byte * in, word32 in_len, byte * out, word32 * out_len, wc_MlDsaKey * key, const byte * context, byte contextLen, word32 preHashType)Sign a message using ML-DSA with DMA.
int	**wh_Client_MlDsaVerifyDma**(whClientContext * ctx, const byte * sig, word32 sig_len, const byte * msg, word32 msg_len, int * res, wc_MlDsaKey * key, const byte * context, byte contextLen, word32 preHashType)Verify a ML-DSA signature with DMA.
int	**wh_Client_MlDsaCheckPrivKeyDma**(whClientContext * ctx, wc_MlDsaKey * key, const byte * pubKey, word32 pubKeySz)Check a ML-DSA private key against public key with DMA.
int	**wh_Client_MlKemSetKeyId**(MlKemKey * key, whKeyId keyId)Associate a ML-KEM key with a specific key ID.

	Name
int	**wh_Client_MlKemGetKeyId**(MlKemKey * key, whKeyId * outId)Retrieve the key ID associated with a ML-KEM key.
int	**wh_Client_MlKemImportKey**(whClientContext * ctx, MlKemKey * key, whKeyId * inout_keyId, whNvmFlags flags, uint16_t label_len, uint8_t * label)Import a ML-KEM key to the server key cache.
int	**wh_Client_MlKemExportKey**(whClientContext * ctx, whKeyId keyId, MlKemKey * key, uint16_t label_len, uint8_t * label)Export a ML-KEM key from the server key cache.
int	**wh_Client_MlKemExportPublicKey**(whClientContext * ctx, whKeyId keyId, MlKemKey * key, uint16_t label_len, uint8_t * label)Exports only the public part of a cached ML-KEM key.
int	**wh_Client_MlKemMakeExportKey**(whClientContext * ctx, int level, MlKemKey * key)Generate a ML-KEM key pair and return it as an ephemeral key.
int	**wh_Client_MlKemMakeCacheKey**(whClientContext * ctx, int level, whKeyId * inout_key_id, whNvmFlags flags, uint16_t label_len, uint8_t * label)Generate a ML-KEM key pair and cache it on the server.
int	**wh_Client_MlKemEncapsulate**(whClientContext * ctx, MlKemKey * key, uint8_t * ct, uint32_t * inout_ct_len, uint8_t * ss, uint32_t * inout_ss_len)Perform ML-KEM encapsulation using a server-cached public key.
int	**wh_Client_MlKemDecapsulate**(whClientContext * ctx, MlKemKey * key, const uint8_t * ct, uint32_t ct_len, uint8_t * ss, uint32_t * inout_ss_len)Perform ML-KEM decapsulation using a server-cached private key.
int	**wh_Client_MlKemImportKeyDma**(whClientContext * ctx, MlKemKey * key, whKeyId * inout_keyId, whNvmFlags flags, uint16_t label_len, uint8_t * label)Import a ML-KEM key using DMA.
int	**wh_Client_MlKemExportKeyDma**(whClientContext * ctx, whKeyId keyId, MlKemKey * key, uint16_t label_len, uint8_t * label)Export a ML-KEM key from the server using DMA.
int	**wh_Client_MlKemExportPublicKeyDma**(whClientContext * ctx, whKeyId keyId, MlKemKey * key, uint16_t label_len, uint8_t * label)Exports only the public part of a cached ML-KEM key using DMA.
int	**wh_Client_MlKemMakeExportKeyDma**(whClientContext * ctx, int level, MlKemKey * key)Generate an ephemeral ML-KEM key pair using DMA.

	Name
int	**wh_Client_MlKemEncapsulateDma**(whClientContext * ctx, MlKemKey * key, uint8_t * ct, uint32_t * inout_ct_len, uint8_t * ss, uint32_t * inout_ss_len)Perform ML-KEM encapsulation using DMA.
int	**wh_Client_MlKemDecapsulateDma**(whClientContext * ctx, MlKemKey * key, const uint8_t * ct, uint32_t ct_len, uint8_t * ss, uint32_t * inout_ss_len)Perform ML-KEM decapsulation using DMA.
int	**wh_Client_LmsSetKeyId**(LmsKey * key, whKeyId keyId)Bind a wolfHSM keyId into an LmsKey's devCtx.
int	**wh_Client_LmsGetKeyId**(LmsKey * key, whKeyId * outId)Read the wolfHSM keyId stored in an LmsKey's devCtx.
int	**wh_Client_LmsMakeKeyDma**(whClientContext * ctx, LmsKey * key, whKeyId * inout_key_id, whNvmFlags flags, uint16_t label_len, uint8_t * label)Generate an LMS key on the server.
int	**wh_Client_LmsMakeExportKeyDma**(whClientContext * ctx, LmsKey * key)Convenience wrapper for keygen that returns the public key via DMA.
int	**wh_Client_LmsSignDma**(whClientContext * ctx, const byte * msg, word32 msgSz, byte * sig, word32 * sigSz, LmsKey * key)Sign a message with an HSM-resident LMS key.
int	**wh_Client_LmsVerifyDma**(whClientContext * ctx, const byte * sig, word32 sigSz, const byte * msg, word32 msgSz, int * res, LmsKey * key)Verify a signature using an HSM-resident LMS key.
int	**wh_Client_LmsSigsLeftDma**(whClientContext * ctx, LmsKey * key)Report whether an HSM-resident LMS key can still produce signatures.
int	**wh_Client_LmsImportPubKey**(whClientContext * ctx, LmsKey * key, whKeyId * inout_keyId, whNvmFlags flags, uint16_t label_len, uint8_t * label)Import a verify-only LMS public key into the keystore.
int	**wh_Client_XmssSetKeyId**(XmssKey * key, whKeyId keyId)Bind a wolfHSM keyId into an XmssKey's devCtx.
int	**wh_Client_XmssGetKeyId**(XmssKey * key, whKeyId * outId)Read the wolfHSM keyId stored in an XmssKey's devCtx.
int	**wh_Client_XmssMakeKeyDma**(whClientContext * ctx, XmssKey * key, whKeyId * inout_key_id, whNvmFlags flags, uint16_t label_len, uint8_t * label)Generate an XMSS / XMSS^MT key on the server.

	Name
int	**wh_Client_XmssMakeExportKeyDma**(whClientContext * ctx, XmssKey * key)Convenience wrapper for keygen that returns the public key via DMA.
int	**wh_Client_XmssSignDma**(whClientContext * ctx, const byte * msg, word32 msgSz, byte * sig, word32 * sigSz, XmssKey * key)Sign a message with an HSM-resident XMSS key.
int	**wh_Client_XmssVerifyDma**(whClientContext * ctx, const byte * sig, word32 sigSz, const byte * msg, word32 msgSz, int * res, XmssKey * key)Verify a signature using an HSM-resident XMSS key.
int	**wh_Client_XmssSigsLeftDma**(whClientContext * ctx, XmssKey * key)Report whether an HSM-resident XMSS key can still produce signatures.
int	**wh_Client_XmssImportPubKey**(whClientContext * ctx, XmssKey * key, whKeyId * inout_keyId, whNvmFlags flags, uint16_t label_len, uint8_t * label)Import a verify-only XMSS / XMSS^MT public key into the keystore.

A.2.2 Functions Documentation

A.2.2.1 function wh_Client_RngGenerate

```
int wh_Client_RngGenerate(
    whClientContext * ctx,
    uint8_t * out,
    uint32_t size
)
```

Generate random bytes.

Parameters:

- **ctx** Pointer to the client context
- **out** Pointer to where the bytes are to be placed. Must not be NULL.
- **size** Number of bytes to generate.

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server to generate random bytes by repeatedly requesting the maximum block size of data from the server at a time

A.2.2.2 function wh_Client_RngGenerateRequest

```
int wh_Client_RngGenerateRequest(
    whClientContext * ctx,
    uint32_t size
)
```

Async request half of a non-DMA RNG generate.

Parameters:

- **ctx** Client context.
- **size** Number of random bytes to request. Must be > 0 and must not exceed WH_MESSAGE_CRYPTO_RNG_MAX_IN

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args or a size exceeding the per-call inline capacity, or a negative error from the transport.

Serializes and sends a request for size bytes of random data. Does NOT wait for a reply. Single-shot per call: chunking large requests is the caller's responsibility. The blocking wrapper wh_Client_RngGenerate handles chunking automatically.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_RngGenerateResponse before issuing any other async Request on the same ctx.

A.2.2.3 function wh_Client_RngGenerateResponse

```
int wh_Client_RngGenerateResponse(
    whClientContext * ctx,
    uint8_t * out,
    uint32_t * inout_size
)
```

Async response half of a non-DMA RNG generate.

Parameters:

- **ctx** Client context.
- **out** Buffer to receive random bytes. May be NULL only if *inout_size is 0.
- **inout_size** On entry: capacity of out (typically equals the size passed to wh_Client_RngGenerateRequest). On success: number of bytes actually written to out.

Return: WH_ERROR_OK on success, WH_ERROR_NOTREADY if no reply yet, WH_ERROR_ABORTED if the server returned more bytes than the buffer can hold, WH_ERROR_BADARGS for invalid args.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, copies up to *inout_size random bytes into out and updates* inout_size to the actual number received.

A.2.2.4 function wh_Client_RngGenerateDma

```
int wh_Client_RngGenerateDma(
    whClientContext * ctx,
    uint8_t * out,
    uint32_t size
)
```

Generate random bytes using DMA.

Parameters:

- **ctx** Pointer to the client context
- **out** Pointer to where the bytes are to be placed
- **size** Number of bytes to generate

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server to generate random bytes directly into client memory using DMA, eliminating the need for chunking and copying through the communication buffer.

A.2.2.5 function wh_Client_RngGenerateDmaRequest

```
int wh_Client_RngGenerateDmaRequest(
    whClientContext * ctx,
    uint8_t * out,
    uint32_t size
)
```

Async request half of a DMA RNG generate.

Parameters:

- **ctx** Client context.
- **out** Client buffer that will receive the random bytes via DMA.
- **size** Number of random bytes to generate. Must be > 0.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args, or a negative error from the DMA layer or transport. On failure any acquired DMA mapping is released before returning.

Performs PRE address translation for the output buffer, sends the DMA request, and stashes the translated address for POST cleanup in the matching Response. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_RngGenerateDmaResponse before issuing any other async Request on the same ctx, and must keep out valid until the Response completes.

A.2.2.6 function wh_Client_RngGenerateDmaResponse

```
int wh_Client_RngGenerateDmaResponse(
    whClientContext * ctx
)
```

Async response half of a DMA RNG generate.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. The random bytes are written by the server directly to the client buffer passed to wh_Client_RngGenerateDmaRequest. POST DMA cleanup for the output buffer is performed on every non-NOTREADY return so the client buffer is safe to read regardless of error.

A.2.2.7 function wh_Client_Curve25519SetKeyId

```
int wh_Client_Curve25519SetKeyId(
    curve25519_key * key,
    whKeyId keyId
)
```

Associates a Curve25519 key with a specific key ID.

Parameters:

- **key** Pointer to the Curve25519 key structure.
- **keyId** Key ID to be associated with the Curve25519 key.

Return: int Returns 0 on success or a negative error code on failure.

This function sets the device context of a Curve25519 key to the specified key ID. On the server side, this key ID is used to reference the key stored in the HSM

A.2.2.8 function wh_Client_Curve25519GetKeyId

```
int wh_Client_Curve25519GetKeyId(
    curve25519_key * key,
```

```
    whKeyId * outId
)
```

Gets the wolfHSM keyId being used by the wolfCrypt struct.

Parameters:

- **key** Pointer to the Curve25519 key structure.
- **outId** Pointer to the key ID to return.

Return: int Returns 0 on success or a negative error code on failure.

This function gets the device context of a Curve25519 key that was previously set by either the crypto callback layer or wh_Client_SetKeyCurve25519.

A.2.2.9 function wh_Client_Curve25519ImportKey

```
int wh_Client_Curve25519ImportKey(
    whClientContext * ctx,
    curve25519_key * key,
    whKeyId * inout_keyId,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Imports wolfCrypt Curve25519 key as a raw byte array into the wolfHSM server key cache.

Parameters:

- **ctx** Pointer to the wolfHSM client structure.
- **key** Pointer to the curve25519 key structure.
- **inout_keyId** Pointer to the key ID. Set to WH_KEYID_ERASED to have the server allocate a unique id. May be NULL.
- **flags** Value of flags to indicate server usage
- **label_len** Length of the optional label in bytes, Valid values are 0 to WH_NVM_LABEL_LEN.
- **label** pointer to the optional label byte array. May be NULL.

Return: int Returns 0 on success or a negative error code on failure.

This function converts the curve25519_key struct to serialized format, installs into the server's key cache, and provides the server-allocated keyId for reference.

A.2.2.10 function wh_Client_Curve25519ExportKey

```
int wh_Client_Curve25519ExportKey(
    whClientContext * ctx,
    whKeyId keyId,
    curve25519_key * key,
    uint16_t label_len,
    uint8_t * label
)
```

Exports a serialized curve25519 key from the wolfHSM server keycache and decodes it into the wolfCrypt curve25519 key structure.

Parameters:

- **ctx** Pointer to the wolfHSM client structure.
- **out_keyId** Server key ID to export.
- **key** Pointer to the Curve25519 key structure.

- **label_len** Length of the optional label in bytes, Valid values are 0 to WH_NVM_LABEL_LEN.
- **label** pointer to the optional label byte array. May be NULL.

Return: int Returns 0 on success or a negative error code on failure.

This function exports the specified key from wolfHSM server key cache as a serialized byte array and decodes the key into the wolfCrypt curve25519_key structure, optionally copying out the associated label as well.

A.2.2.11 function wh_Client_Curve25519ExportPublicKey

```
int wh_Client_Curve25519ExportPublicKey(
    whClientContext * ctx,
    whKeyId keyId,
    curve25519_key * key,
    uint16_t label_len,
    uint8_t * label
)
```

Exports only the public part of a cached Curve25519 key.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **keyId** Server key ID whose public key should be exported. Must not be WH_KEYID_ERASED.
- **key** Pointer to a caller-initialized curve25519_key. On success, the public portion is populated.
- **label_len** Size of the optional label buffer in bytes. Values larger than WH_NVM_LABEL_LEN are truncated. Set to 0 if label is not needed.
- **label** Optional buffer to receive the key's label. May be NULL.

Return: int Returns 0 on success or a negative wolfHSM/wolfCrypt error code on failure (e.g. WH_ERROR_NOTFOUND, WH_ERROR_BADARGS).

Instructs the server to emit only the public portion of a cached Curve25519 key as SubjectPublicKeyInfo DER. The private scalar stays inside the HSM. The decoded key will have only the public part set.

The NONEXPORTABLE key flag does NOT block this call because public material is non-sensitive. The caller is responsible for initializing key (e.g. wc_curve25519_init_ex) prior to calling this function.

A.2.2.12 function wh_Client_Curve25519MakeCacheKey

```
int wh_Client_Curve25519MakeCacheKey(
    whClientContext * ctx,
    uint16_t size,
    whKeyId * inout_key_id,
    whNvmFlags flags,
    const uint8_t * label,
    uint16_t label_len
)
```

Generate a Curve25519 key in the server key cache.

Parameters:

- **ctx** Pointer to the client context
- **size** Size of the key to generate in bytes, normally set to CURVE25519_KEY_SIZE.
- **inout_key_id.** Set to WH_KEYID_ERASED to have the server select a unique id for this key.
- **flags** Optional flags to be associated with the key while in the key cache or after being committed. Set to WH_NVM_FLAGS_NONE if not used.

- **label** Optional label to be associated with the key while in the key cache or after being committed. Set to NULL if not used.
- **label_len** Size of the label up to WH_NVM_LABEL_SIZE. Set to 0 if not used.

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server to generate a new Curve25519 key and insert it into the server's key cache.

A.2.2.13 function wh_Client_Curve25519MakeExportKey

```
int wh_Client_Curve25519MakeExportKey(
    whClientContext * ctx,
    uint16_t size,
    curve25519_key * key
)
```

Generate a Curve25519 key by the server and export to the client.

Parameters:

- **ctx** Pointer to the client context
- **size** Size of the key to generate in bytes, normally set to CURVE25519_KEY_SIZE.
- **key** Pointer to a wolfCrypt key structure, which will be initialized to the new key pair when successful

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server to generate a new Curve25519 key pair and export it to the client, without using any key cache or additional resources

A.2.2.14 function wh_Client_Curve25519SharedSecret

```
int wh_Client_Curve25519SharedSecret(
    whClientContext * ctx,
    curve25519_key * priv_key,
    curve25519_key * pub_key,
    int endian,
    uint8_t * out,
    uint16_t * out_size
)
```

Compute an X25519 shared secret using a public and private key.

Parameters:

- **ctx** Pointer to the client context
- **priv_key** Pointer to a wolfCrypt key structure that holds the private key
- **pub_key** Pointer to a wolfCrypt key structure that holds the public key
- **endian** Endianness of the values. EC25519_BIG_ENDIAN (typical) or EC25519_LITTLE_ENDIAN

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server compute the shared secret using the provided wolfCrypt private and public keys. Note, the client will temporarily import any missing key material to the server as required.

A.2.2.15 function wh_Client_Curve25519SharedSecretRequest

```
int wh_Client_Curve25519SharedSecretRequest(
    whClientContext * ctx,
```

```
    whKeyId prv_key_id,
    whKeyId pub_key_id,
    int endian
)
```

Send a request to compute an X25519 shared secret that will be returned to the client.

The matching response is retrieved with wh_Client_Curve25519SharedSecretResponse. Both input keys must already be cached on the server.

A.2.2.16 function wh_Client_Curve25519SharedSecretResponse

```
int wh_Client_Curve25519SharedSecretResponse(
    whClientContext * ctx,
    uint8_t * out,
    uint16_t * out_size
)
```

Retrieve the response to a wh_Client_Curve25519SharedSecretRequest.

A.2.2.17 function wh_Client_Curve25519SharedSecretCacheKey

```
int wh_Client_Curve25519SharedSecretCacheKey(
    whClientContext * ctx,
    curve25519_key * priv_key,
    curve25519_key * pub_key,
    int endian,
    whKeyId * inout_key_id,
    whNvmFlags flags,
    const uint8_t * label,
    uint16_t label_len
)
```

Compute an X25519 shared secret and cache it on the server.

Parameters:

- **ctx** Pointer to the client context
- **priv_key** Private key (cached on the server or local)
- **pub_key** Public key (cached on the server or local)
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN
- **inout_key_id** Cache slot id (in) / assigned id (out)
- **flags** whNvmFlags applied to the cached secret
- **label** Optional label for the cached secret
- **label_len** Length of `label`, up to WH_NVM_LABEL_LEN

On return, *inout_key_id holds the keyId the secret is stored under (the server allocates one when* inout_key_id is WH_KEYID_ERASED on entry). `flags` must not include WH_NVM_FLAGS_EPHEMERAL.

A.2.2.18 function wh_Client_Curve25519SharedSecretCacheKeyRequest

```
int wh_Client_Curve25519SharedSecretCacheKeyRequest(
    whClientContext * ctx,
    whKeyId prv_key_id,
    whKeyId pub_key_id,
    int endian,
    whKeyId out_key_id,
    whNvmFlags flags,
```

```
    const uint8_t * label,
    uint16_t label_len
)
```

Async variant: send the request half of SharedSecretCacheKey. Both input keys must already be cached.

A.2.2.19 function wh_Client_Curve25519SharedSecretCacheKeyResponse

```
int wh_Client_Curve25519SharedSecretCacheKeyResponse(
    whClientContext * ctx,
    whKeyId * out_key_id
)
```

Async variant: retrieve the response of SharedSecretCacheKeyRequest. On success, *out_key_id receives the assigned cache id.

A.2.2.20 function wh_Client_EccSetKeyId

```
int wh_Client_EccSetKeyId(
    ecc_key * key,
    whKeyId keyId
)
```

Associates a Ecc key with a specific key ID.

Parameters:

- **key** Pointer to the Ecc key structure.
- **keyId** Key ID to be associated with the Ecc key.

Return: int Returns 0 on success or a negative error code on failure.

This function sets the device context of a Ecc key to the specified key ID. On the server side, this key ID is used to reference the key stored in the HSM

A.2.2.21 function wh_Client_EccGetKeyId

```
int wh_Client_EccGetKeyId(
    ecc_key * key,
    whKeyId * outId
)
```

Gets the wolfHSM keyId being used by the wolfCrypt struct.

Parameters:

- **key** Pointer to the Ecc key structure.
- **outId** Pointer to the key ID to return.

Return: int Returns 0 on success or a negative error code on failure.

This function gets the device context of a Ecc key that was previously set by either the crypto callback layer or wh_Client_EccSetKeyId.

A.2.2.22 function wh_Client_EccImportKey

```
int wh_Client_EccImportKey(
    whClientContext * ctx,
    ecc_key * key,
```

```
    whKeyId * inout_keyId,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Imports a wolfCrypt ECC key as a DER-formatted blob into the wolfHSM server key cache.

Parameters:

- **ctx** Pointer to the wolfHSM client structure.
- **key** Pointer to the ECC key structure.
- **inout_keyId** Pointer to the key ID. Set to WH_KEYID_ERASED to have the server allocate a unique id. May be NULL.
- **flags** Value of flags to indicate server usage
- **label_len** Length of the optional label in bytes, Valid values are 0 to WH_NVM_LABEL_LEN.
- **label** pointer to the optional label byte array. May be NULL.

Return: int Returns 0 on success or a negative error code on failure.

This function serializes the ecc_key struct to DER format, installs it into the server's key cache, and provides the server-allocated keyId for reference.

A.2.2.23 function wh_Client_EccExportKey

```
int wh_Client_EccExportKey(
    whClientContext * ctx,
    whKeyId keyId,
    ecc_key * key,
    uint16_t label_len,
    uint8_t * label
)
```

Exports a DER-formatted ECC key from the wolfHSM server keycache and decodes it into the wolfCrypt ECC key structure.

Parameters:

- **ctx** Pointer to the wolfHSM client structure.
- **keyId** Server key ID to export.
- **key** Pointer to the ECC key structure to populate.
- **label_len** Length of the optional label in bytes, Valid values are 0 to WH_NVM_LABEL_LEN.
- **label** pointer to the optional label byte array. May be NULL.

Return: int Returns 0 on success or a negative error code on failure.

This function exports the specified key from the wolfHSM server key cache as a DER blob and decodes it into the wolfCrypt ecc_key structure, optionally copying out the associated label as well.

A.2.2.24 function wh_Client_EccExportPublicKey

```
int wh_Client_EccExportPublicKey(
    whClientContext * ctx,
    whKeyId keyId,
    ecc_key * key,
    uint16_t label_len,
    uint8_t * label
)
```

Exports only the public part of a cached ECC key.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **keyId** Server key ID whose public key should be exported. Must not be WH_KEYID_ERASED.
- **key** Pointer to a caller-initialized ecc_key. On success, the public point is populated and key->type is set to ECC_PUBLICKEY.
- **label_len** Size of the optional label buffer in bytes. Values larger than WH_NVM_LABEL_LEN are truncated. Set to 0 if label is not needed.
- **label** Optional buffer to receive the key's label. May be NULL.

Return: int Returns 0 on success or a negative wolfHSM/wolfCrypt error code on failure (e.g. WH_ERROR_NOTFOUND, WH_ERROR_BADARGS).

Instructs the server to emit only the public portion (SubjectPublicKeyInfo DER with curve parameters) of a cached ECC key. The private scalar stays inside the HSM. The decoded key is written into the caller-initialized ecc_key struct and will report type == ECC_PUBLICKEY.

The NONEXPORTABLE key flag does NOT block this call because public material is non-sensitive. The caller is responsible for initializing key (e.g. wc_ecc_init_ex) prior to calling this function.

A.2.2.25 function wh_Client_EccMakeExportKey

```
int wh_Client_EccMakeExportKey(
    whClientContext * ctx,
    int size,
    int curveId,
    ecc_key * key
)
```

Generate an ECC key pair on the server and export it to the client.

Parameters:

- **ctx** Pointer to the client context.
- **size** Size of the key to generate in bytes (e.g. 32 for P-256).
- **curveId** wolfCrypt curve identifier (e.g. ECC_SECP256R1).
- **key** Pointer to a wolfCrypt ECC key structure, which will be populated with the new key pair when successful.

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server to generate a new ECC key pair and export it to the client, without using any key cache or additional resources.

A.2.2.26 function wh_Client_EccMakeCacheKey

```
int wh_Client_EccMakeCacheKey(
    whClientContext * ctx,
    int size,
    int curveId,
    whKeyId * inout_key_id,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Generate an ECC key pair in the server key cache.

Parameters:

- **ctx** Pointer to the client context.
- **size** Size of the key to generate in bytes (e.g. 32 for P-256).
- **curveId** wolfCrypt curve identifier (e.g. ECC_SECP256R1).
- **inout_key_id** Pointer to the key ID. Set to WH_KEYID_ERASED to have the server allocate a unique id. Must not be NULL.
- **flags** Optional flags to be associated with the key while in the key cache or after being committed. Set to WH_NVM_FLAGS_NONE if not used.
- **label_len** Size of the label up to WH_NVM_LABEL_LEN. Set to 0 if not used.
- **label** Optional label to be associated with the key while in the key cache or after being committed. Set to NULL if not used.

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server to generate a new ECC key pair and insert it into the server's key cache. The generated key material is not returned to the client.

A.2.2.27 function wh_Client_EccSharedSecret

```
int wh_Client_EccSharedSecret(
    whClientContext * ctx,
    ecc_key * priv_key,
    ecc_key * pub_key,
    uint8_t * out,
    uint16_t * inout_size
)
```

Compute an ECDH shared secret using a public and private ECC key.

Parameters:

- **ctx** Pointer to the client context.
- **priv_key** Pointer to a wolfCrypt key structure that either holds the private key material or references a server-cached private key via its devCtx (keyId).
- **pub_key** Pointer to a wolfCrypt key structure that either holds the public key material or references a server-cached public key via its devCtx (keyId).
- **out** Buffer to receive the computed shared secret. May be NULL to query the required size, in which case inout_size must be non-NULL and the required size will be written to *inout_size with WH_ERROR_BUFFER_SIZE returned.
- **inout_size** On input, the capacity of the out buffer in bytes when out is non-NULL. On output, the number of bytes written on success, or the required buffer size when WH_ERROR_BUFFER_SIZE is returned. Must not be NULL when out is non-NULL; may be NULL only when out is also NULL.

Return: int Returns 0 on success, WH_ERROR_BUFFER_SIZE if the caller's out buffer is too small to hold the shared secret (with required size written to *inout_size), or a negative error code on failure.

This function requests the server to compute the shared secret using the provided wolfCrypt private and public keys. Either key context may carry actual key material or refer to a server-cached key by keyId via its devCtx (associated by wh_Client_EccSetKeyId or returned from a server-side keygen). For any context that does not reference a cached keyId, the client will temporarily import its material to the server for the duration of the operation and evict it afterwards.

A.2.2.28 function wh_Client_EccSign

```
int wh_Client_EccSign(
    whClientContext * ctx,
    ecc_key * key,
```

```
    const uint8_t * hash,
    uint16_t hash_len,
    uint8_t * sig,
    uint16_t * inout_sig_len
)
```

Generate an ECDSA signature of the provided hash on the server.

Parameters:

- **ctx** Pointer to the client context.
- **key** Pointer to a wolfCrypt ECC key structure that either holds the private key material or references a server-cached private key via its devCtx (keyId).
- **hash** Hash data to sign. May be NULL only if hash_len is 0.
- **hash_len** Length of hash in bytes.
- **sig** Buffer to receive the generated signature. May be NULL to query the required size, in which case inout_sig_len must be non-NULL and the required size will be written to *inout_sig_len with WH_ERROR_BUFFER_SIZE returned.
- **inout_sig_len** On input, the capacity of the sig buffer in bytes when sig is non-NULL. On output, the number of bytes written on success, or the required buffer size when WH_ERROR_BUFFER_SIZE is returned. Must not be NULL when sig is non-NULL; may be NULL only when sig is also NULL.

Return: int Returns 0 on success, WH_ERROR_BUFFER_SIZE if the caller's sig buffer is too small to hold the signature (with required size written to *inout_sig_len), or a negative error code on failure.

This function requests the server to sign the provided hash using the specified ECC key. The key context may either carry actual key material or refer to a server-cached key by keyId via its devCtx (associated by wh_Client_EccSetKeyId or returned from a server-side keygen). If the key does not reference a cached keyId, the client will temporarily import its material to the server for the duration of the operation and evict it afterwards.

A.2.2.29 function wh_Client_EccVerify

```
int wh_Client_EccVerify(
    whClientContext * ctx,
    ecc_key * key,
    const uint8_t * sig,
    uint16_t sig_len,
    const uint8_t * hash,
    uint16_t hash_len,
    int * out_res
)
```

Verify an ECDSA signature of the provided hash on the server.

Parameters:

- **ctx** Pointer to the client context.
- **key** Pointer to a wolfCrypt ECC key structure that either holds the public key material or references a server-cached public key via its devCtx (keyId).
- **sig** Signature bytes.
- **sig_len** Length of sig in bytes.
- **hash** Hash bytes that were signed.
- **hash_len** Length of hash in bytes.
- **out_res** Pointer to receive the verification result. Set to 1 if the signature is valid, 0 otherwise. Must not be NULL.

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server to verify the provided signature against the provided hash using the specified ECC key. The key context may either carry actual key material or refer to a server-cached key by keyId via its devCtx (associated by wh_Client_EccSetKeyId or returned from a server-side keygen). If the key does not reference a cached keyId, the client will temporarily import its material to the server for the duration of the operation and evict it afterwards. If the supplied key is private-only, the server will derive the public key as needed.

A.2.2.30 function wh_Client_EccSignRequest

```
int wh_Client_EccSignRequest(
    whClientContext * ctx,
    whKeyId keyId,
    const uint8_t * hash,
    uint16_t hash_len
)
```

Async request half of an ECC sign operation.

Parameters:

- **ctx** Client context.
- **keyId** Key ID of a cached ECC private key. Must not be erased.
- **hash** Hash data to sign (may be NULL only if hash_len == 0).
- **hash_len** Length of hash in bytes.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args or erased keyId, or a negative error from the transport.

Serializes and sends a sign request for the hash using the server-cached private key identified by keyId. Does NOT wait for a reply. The key must already be cached on the server; auto-import is only available via the blocking wrapper wh_Client_EccSign.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_EccSignResponse before issuing any other async Request on the same ctx.

A.2.2.31 function wh_Client_EccSignResponse

```
int wh_Client_EccSignResponse(
    whClientContext * ctx,
    uint8_t * sig,
    uint16_t * inout_sig_len
)
```

Async response half of an ECC sign operation.

Parameters:

- **ctx** Client context.
- **sig** Buffer to receive the generated signature. May be NULL to query the required size, in which case inout_sig_len must be non-NULL and the required size will be written to *inout_sig_len with WH_ERROR_BUFFER_SIZE returned.
- **inout_sig_len** On input, the capacity of the sig buffer in bytes when sig is non-NULL. On output, the number of bytes written on success, or the required buffer size when WH_ERROR_BUFFER_SIZE is returned. Must not be NULL when sig is non-NULL; may be NULL only when sig is also NULL.

Return: WH_ERROR_OK on success, WH_ERROR_NOTREADY if no reply yet, WH_ERROR_BUFFER_SIZE if sig is too small (required size written to *inout_sig_len), WH_ERROR_BADARGS for invalid args, or a negative error code from the transport.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, copies the signature into sig and updates *inout_sig_len. If the server-reported signature is larger than the caller's* inout_sig_len capacity, returns WH_ERROR_BUFFER_SIZE with the required size written to *inout_sig_len.

A.2.2.32 function wh_Client_EccVerifyRequest

```
int wh_Client_EccVerifyRequest(
    whClientContext * ctx,
    whKeyId keyId,
    const uint8_t * sig,
    uint16_t sig_len,
    const uint8_t * hash,
    uint16_t hash_len
)
```

Async request half of an ECC verify operation.

Parameters:

- **ctx** Client context.
- **keyId** Key ID of a cached ECC public key. Must not be erased.
- **sig** Signature bytes.
- **sig_len** Length of sig.
- **hash** Hash bytes.
- **hash_len** Length of hash.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args or erased keyId, or a negative error from the transport.

Serializes and sends a verify request for (sig, hash) using the server-cached public key identified by keyId. Does NOT wait for a reply. The key must already be cached on the server; auto-import is only available via the blocking wrapper wh_Client_EccVerify.

Note: the async API does not support the EXPORTPUB convenience (deriving a public key from a private-only key) — that stays a blocking-wrapper convenience.

A.2.2.33 function wh_Client_EccVerifyResponse

```
int wh_Client_EccVerifyResponse(
    whClientContext * ctx,
    ecc_key * opt_key,
    int * out_res
)
```

Async response half of an ECC verify operation.

Parameters:

- **ctx** Client context.
- **opt_key** Optional ecc_key whose public half should be updated from the server-supplied DER bytes when the matching Request had EXPORTPUB set. Pass NULL when no key update is desired. The async Request half does not currently expose EXPORTPUB, so this parameter is primarily for the blocking wrapper wh_Client_EccVerify.
- **out_res** 1 if the signature is valid, 0 otherwise.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, writes the verify result (1 = valid, 0 = invalid) to *out_res.

A.2.2.34 function wh_Client_EccSharedSecretRequest

```
int wh_Client_EccSharedSecretRequest(
    whClientContext * ctx,
    whKeyId prv_key_id,
    whKeyId pub_key_id
)
```

Async request half of an ECDH shared-secret operation.

Parameters:

- **ctx** Client context.
- **prv_key_id** Key ID of the cached private key. Must not be erased.
- **pub_key_id** Key ID of the cached public key. Must not be erased.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args or erased keyIds, or a negative error from the transport.

Serializes and sends a shared-secret request using two server-cached keys (private and public). Does NOT wait for a reply. Both keys must already be cached on the server; auto-import is only available via the blocking wrapper wh_Client_EccSharedSecret.

A.2.2.35 function wh_Client_EccSharedSecretResponse

```
int wh_Client_EccSharedSecretResponse(
    whClientContext * ctx,
    uint8_t * out,
    uint16_t * inout_size
)
```

Async response half of an ECDH shared-secret operation.

Parameters:

- **ctx** Client context.
- **out** Buffer to receive the computed shared secret. May be NULL to query the required size, in which case inout_size must be non-NULL and the required size will be written to *inout_size with WH_ERROR_BUFFER_SIZE returned.
- **inout_size** On input, the capacity of the out buffer in bytes when out is non-NULL. On output, the number of bytes written on success, or the required buffer size when WH_ERROR_BUFFER_SIZE is returned. Must not be NULL when out is non-NULL; may be NULL only when out is also NULL.

Return: WH_ERROR_OK on success, WH_ERROR_NOTREADY if no reply yet, WH_ERROR_BUFFER_SIZE if out is too small (required size written to *inout_size), WH_ERROR_BADARGS for invalid args, or a negative error code from the transport.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, copies the shared secret into out and updates *inout_size*. inout_size is in/out: when out is non-NULL, callers must initialize it to the capacity of the out buffer. If the server-reported secret is larger than the caller's capacity, returns WH_ERROR_BUFFER_SIZE with the required size written to *inout_size — the partial buffer is NOT written, since truncated key material would be unsafe to use.

A.2.2.36 function wh_Client_EccSharedSecretCacheKey

```
int wh_Client_EccSharedSecretCacheKey(
    whClientContext * ctx,
    ecc_key * priv_key,
    ecc_key * pub_key,
    whKeyId * inout_key_id,
```

```
    whNvmFlags flags,
    const uint8_t * label,
    uint16_t label_len
)
```

Compute an ECDH shared secret and cache it on the server.

Parameters:

- **ctx** Client context
- **priv_key** Private ECC key (cached on the server or local)
- **pub_key** Public ECC key (cached on the server or local)
- **inout_key_id** Cache slot id (in) / assigned id (out)
- **flags** whNvmFlags applied to the cached secret
- **label** Optional label for the cached secret
- **label_len** Length of `label`, up to WH_NVM_LABEL_LEN

On return, *inout_key_id holds the keyId the secret is stored under (the server allocates one when* inout_key_id is WH_KEYID_ERASED on entry). `flags` must not include WH_NVM_FLAGS_EPHEMERAL.

A.2.2.37 function wh_Client_EccSharedSecretCacheKeyRequest

```
int wh_Client_EccSharedSecretCacheKeyRequest(
    whClientContext * ctx,
    whKeyId prv_key_id,
    whKeyId pub_key_id,
    whKeyId out_key_id,
    whNvmFlags flags,
    const uint8_t * label,
    uint16_t label_len
)
```

Async variant: send the request half of SharedSecretCacheKey. Both input keys must already be cached on the server.

A.2.2.38 function wh_Client_EccSharedSecretCacheKeyResponse

```
int wh_Client_EccSharedSecretCacheKeyResponse(
    whClientContext * ctx,
    whKeyId * out_key_id
)
```

Async variant: retrieve the response of SharedSecretCacheKeyRequest. On success, *out_key_id receives the assigned cache id.

A.2.2.39 function wh_Client_EccMakeCacheKeyRequest

```
int wh_Client_EccMakeCacheKeyRequest(
    whClientContext * ctx,
    int size,
    int curveId,
    whKeyId key_id,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Async request half of an ECC server-side keygen that caches the new key in the server.

Parameters:

- **ctx** Client context.
- **size** Size of the key to generate in bytes (e.g. 32 for P-256).
- **curveId** wolfCrypt curve identifier (e.g. ECC_SECP256R1).
- **key_id** Suggested key ID. Pass WH_KEYID_ERASED to have the server allocate one.
- **flags** Optional NVM flags. Must NOT include WH_NVM_FLAGS_EPHEMERAL — use the MakeExportKey async pair for ephemeral (export) keygen instead.
- **label_len** Size of the label up to WH_NVM_LABEL_LEN. Set to 0 if not used.
- **label** Optional label byte array. May be NULL when label_len is 0.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args (including EPHEMERAL flag), or a negative error from the transport.

Serializes and sends a keygen request that asks the server to generate a new ECC key pair on the specified curve and insert it into the server key cache. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_EccMakeCacheKeyResponse before issuing any other async Request on the same ctx.

A.2.2.40 function wh_Client_EccMakeCacheKeyResponse

```
int wh_Client_EccMakeCacheKeyResponse(
    whClientContext * ctx,
    whKeyId * out_key_id
)
```

Async response half of an ECC server-side keygen that caches the new key in the server.

Parameters:

- **ctx** Client context.
- **out_key_id** Pointer to receive the assigned key ID. Must not be NULL.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, writes the server-allocated key ID into *out_key_id.

A.2.2.41 function wh_Client_EccMakeExportKeyRequest

```
int wh_Client_EccMakeExportKeyRequest(
    whClientContext * ctx,
    int size,
    int curveId
)
```

Async request half of an ECC server-side keygen that exports the new key back to the client.

Parameters:

- **ctx** Client context.
- **size** Size of the key to generate in bytes.
- **curveId** wolfCrypt curve identifier.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args, or a negative error from the transport.

Serializes and sends an ephemeral keygen request that asks the server to generate a new ECC key pair on the specified curve and return its DER encoding to the client (the server does not retain the key). Does NOT wait for a reply.

A.2.2.42 function wh_Client_EccMakeExportKeyResponse

```
int wh_Client_EccMakeExportKeyResponse(
    whClientContext * ctx,
    ecc_key * key
)
```

Async response half of an ECC server-side keygen that exports the new key back to the client.

Parameters:

- **ctx** Client context.
- **key** Pointer to a wolfCrypt ECC key structure that will be populated with the new key pair. Must not be NULL.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, deserializes the DER blob returned by the server into the supplied wolfCrypt ecc_key.

A.2.2.43 function wh_Client_Ed25519SetKeyId

```
int wh_Client_Ed25519SetKeyId(
    ed25519_key * key,
    whKeyId keyId
)
```

Associates an Ed25519 key with a specific key ID.

Sets the device context of an Ed25519 key to the provided key ID.

A.2.2.44 function wh_Client_Ed25519GetKeyId

```
int wh_Client_Ed25519GetKeyId(
    ed25519_key * key,
    whKeyId * outId
)
```

Retrieves the key ID from an Ed25519 key device context.

A.2.2.45 function wh_Client_Ed25519ImportKey

```
int wh_Client_Ed25519ImportKey(
    whClientContext * ctx,
    ed25519_key * key,
    whKeyId * inout_keyId,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Import an Ed25519 key into the server keystore/cache.

A.2.2.46 function wh_Client_Ed25519ExportKey

```
int wh_Client_Ed25519ExportKey(
    whClientContext * ctx,
    whKeyId keyId,
    ed25519_key * key,
    uint16_t label_len,
    uint8_t * label
)
```

Export an Ed25519 key from the server to the client.

A.2.2.47 function wh_Client_Ed25519ExportPublicKey

```
int wh_Client_Ed25519ExportPublicKey(
    whClientContext * ctx,
    whKeyId keyId,
    ed25519_key * key,
    uint16_t label_len,
    uint8_t * label
)
```

Exports only the public part of a cached Ed25519 key.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **keyId** Server key ID whose public key should be exported. Must not be WH_KEYID_ERASED.
- **key** Pointer to a caller-initialized ed25519_key. On success, only the public half is populated (pubKeySet == 1, privKeySet == 0).
- **label_len** Size of the optional label buffer in bytes. Values larger than WH_NVM_LABEL_LEN are truncated. Set to 0 if label is not needed.
- **label** Optional buffer to receive the key's label. May be NULL.

Return: int Returns 0 on success or a negative wolfHSM/wolfCrypt error code on failure (e.g. WH_ERROR_NOTFOUND, WH_ERROR_BADARGS).

Instructs the server to emit only the public portion of a cached Ed25519 key as SubjectPublicKeyInfo DER. The private seed stays inside the HSM. The decoded key will have pubKeySet == 1 and privKeySet == 0.

The NONEXPORTABLE key flag does NOT block this call because public material is non-sensitive. The caller is responsible for initializing key (e.g. wc_ed25519_init_ex) prior to calling this function.

A.2.2.48 function wh_Client_Ed25519MakeExportKey

```
int wh_Client_Ed25519MakeExportKey(
    whClientContext * ctx,
    ed25519_key * key
)
```

Create a new Ed25519 key on the server and export it without caching.

A.2.2.49 function wh_Client_Ed25519MakeCacheKey

```
int wh_Client_Ed25519MakeCacheKey(
    whClientContext * ctx,
    whKeyId * inout_key_id,
    whNvmFlags flags,
```

```
    uint16_t label_len,
    uint8_t * label
)
```

Create a new Ed25519 key on the server and store it in cache/NVM.

A.2.2.50 function wh_Client_Ed25519Sign

```
int wh_Client_Ed25519Sign(
    whClientContext * ctx,
    ed25519_key * key,
    const uint8_t * msg,
    uint32_t msgLen,
    uint8_t type,
    const uint8_t * context,
    uint32_t contextLen,
    uint8_t * sig,
    uint32_t * inout_sig_len
)
```

Sign a message using an Ed25519 key on the server.

A.2.2.51 function wh_Client_Ed25519Verify

```
int wh_Client_Ed25519Verify(
    whClientContext * ctx,
    ed25519_key * key,
    const uint8_t * sig,
    uint32_t sigLen,
    const uint8_t * msg,
    uint32_t msgLen,
    uint8_t type,
    const uint8_t * context,
    uint32_t contextLen,
    int * out_res
)
```

Verify a message signature using an Ed25519 key on the server.

A.2.2.52 function wh_Client_Ed25519SignDma

```
int wh_Client_Ed25519SignDma(
    whClientContext * ctx,
    ed25519_key * key,
    const uint8_t * msg,
    uint32_t msgLen,
    uint8_t type,
    const uint8_t * context,
    uint32_t contextLen,
    uint8_t * sig,
    uint32_t * inout_sig_len
)
```

Sign a message using an Ed25519 key via DMA.

A.2.2.53 function wh_Client_Ed25519VerifyDma

```
int wh_Client_Ed25519VerifyDma(
    whClientContext * ctx,
    ed25519_key * key,
    const uint8_t * sig,
    uint32_t sigLen,
    const uint8_t * msg,
    uint32_t msgLen,
    uint8_t type,
    const uint8_t * context,
    uint32_t contextLen,
    int * out_res
)
```

Verify a signature using an Ed25519 key via DMA.

A.2.2.54 function wh_Client_RsaSetKeyId

```
int wh_Client_RsaSetKeyId(
    RsaKey * key,
    whNvmId keyId
)
```

Associates an RSA key with a specific key ID.

Parameters:

- **key** Pointer to the RSA key structure.
- **keyId** Key ID to be associated with the RSA key.

Return: int Returns 0 on success or a negative error code on failure.

This function sets the device context of an RSA key to the specified key ID. On the server side, this key ID is used to reference the key stored in the HSM.

A.2.2.55 function wh_Client_RsaGetKeyId

```
int wh_Client_RsaGetKeyId(
    RsaKey * key,
    whNvmId * outId
)
```

Gets the wolfHSM keyId being used by the wolfCrypt struct.

Parameters:

- **key** Pointer to the RSA key structure.
- **outId** Pointer to the key ID to return.

Return: int Returns 0 on success or a negative error code on failure.

This function gets the device context of a RSA key that was previously set by either the crypto callback layer or wh_Client_SetKeyRsa.

A.2.2.56 function wh_Client_RsaImportKey

```
int wh_Client_RsaImportKey(
    whClientContext * ctx,
    const RsaKey * key,
```

```
    whKeyId * inout_keyId,
    whNvmFlags flags,
    uint32_t label_len,
    uint8_t * label
)
```

Imports wolfCrypt RSA key as a PCKS1 DER-formatted file into the wolfHSM server key cache.

Parameters:

- **ctx** Pointer to the wolfHSM client structure.
- **key** Pointer to the RSA key structure.
- **flags** Value of flags to indicate server usage
- **label_len** Length of the optional label in bytes, Valid values are 0 to WH_NVM_LABEL_LEN.
- **label** pointer to the optional label byte array. May be NULL.
- **out_keyId** Pointer to the key ID to return.

Return: int Returns 0 on success or a negative error code on failure.

This function converts the RsaKey struct to DER format, installs into the server's key cache, and provides the server-allocated keyId for reference.

A.2.2.57 function wh_Client_RsaExportKey

```
int wh_Client_RsaExportKey(
    whClientContext * ctx,
    whKeyId keyId,
    RsaKey * key,
    uint32_t label_len,
    uint8_t * label
)
```

Exports a PKCS1 DER-formated RSA key from the wolfHSM server keycache and decodes it into the wolfCrypt RSA key structure.

Parameters:

- **ctx** Pointer to the wolfHSM client structure.
- **out_keyId** Server key ID to export.
- **key** Pointer to the RSA key structure.
- **label_len** Length of the optional label in bytes, Valid values are 0 to WH_NVM_LABEL_LEN.
- **label** pointer to the optional label byte array. May be NULL.

Return: int Returns 0 on success or a negative error code on failure.

This function exports the specified key from wolfHSM server key cache as a PCKS1 DER file and decodes the key into the wolfCrypt RsaKey structure, optionally copying out the associated label as well.

A.2.2.58 function wh_Client_RsaExportPublicKey

```
int wh_Client_RsaExportPublicKey(
    whClientContext * ctx,
    whKeyId keyId,
    RsaKey * key,
    uint32_t label_len,
    uint8_t * label
)
```

Exports only the public part of a cached RSA key.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **keyId** Server key ID whose public key should be exported. Must not be WH_KEYID_ERASED.
- **key** Pointer to a caller-initialized RsaKey. On success, the modulus and public exponent are populated and key->type is set to RSA_PUBLIC.
- **label_len** Size of the optional label buffer in bytes. Values larger than WH_NVM_LABEL_LEN are truncated. Set to 0 if label is not needed.
- **label** Optional buffer to receive the key's label. May be NULL.

Return: int Returns 0 on success or a negative wolfHSM/wolfCrypt error code on failure (e.g. WH_ERROR_NOTFOUND, WH_ERROR_BADARGS).

Unlike wh_Client_RsaExportKey(), which returns the full cached key (including private material), this function instructs the server to emit only the public portion as a PKCS#1 DER blob. The private key stays inside the HSM. The decoded key is written into the caller-initialized RsaKey struct and will report type == RSA_PUBLIC.

The NONEXPORTABLE key flag does NOT block this call because public material is non-sensitive.

The caller is responsible for initializing key (e.g. wc_InitRsaKey_ex) prior to calling this function.

A.2.2.59 function wh_Client_RsaMakeExportKey

```
int wh_Client_RsaMakeExportKey(
    whClientContext * ctx,
    uint32_t size,
    uint32_t e,
    RsaKey * rsa
)
```

A.2.2.60 function wh_Client_RsaMakeCacheKey

```
int wh_Client_RsaMakeCacheKey(
    whClientContext * ctx,
    uint32_t size,
    uint32_t e,
    whKeyId * inout_key_id,
    whNvmFlags flags,
    uint32_t label_len,
    uint8_t * label
)
```

A.2.2.61 function wh_Client_RsaFunction

```
int wh_Client_RsaFunction(
    whClientContext * ctx,
    RsaKey * key,
    int rsa_type,
    const uint8_t * in,
    uint16_t in_len,
    uint8_t * out,
    uint16_t * inout_out_len
)
```

A.2.2.62 function wh_Client_RsaGetSize

```
int wh_Client_RsaGetSize(
    whClientContext * ctx,
    const RsaKey * key,
    int * out_size
)
```

A.2.2.63 function wh_Client_RsaFunctionRequest

```
int wh_Client_RsaFunctionRequest(
    whClientContext * ctx,
    whKeyId keyId,
    int rsa_type,
    const uint8_t * in,
    uint16_t in_len,
    uint16_t out_capacity
)
```

Send an async RSA encrypt/decrypt/sign/verify request.

Parameters:

- **ctx** Client context.
- **keyId** Cached RSA key ID. Must not be erased.
- **rsa_type** RSA_PUBLIC_ENCRYPT, RSA_PRIVATE_ENCRYPT, RSA_PUBLIC_DECRYPT, or RSA_PRIVATE_DECRYPT.
- **in** Input bytes (may be NULL only if in_len == 0).
- **in_len** Length of in.
- **out_capacity** Maximum number of output bytes the server is allowed to produce for this operation (forwarded as the wc_RsaFunction outLen cap). Typically the RSA modulus size in bytes. This is independent of the client-side response buffer passed to wh_Client_RsaFunctionResponse.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args or erased keyId, or a negative error from the transport.

The key must already be cached on the server; auto-import is only available via the blocking wrapper wh_Client_RsaFunction. Server-side eviction in the same round trip is also only available via the blocking wrapper — async callers that want the key evicted must pair this with wh_Client_KeyEvict. Only one async request may be in flight per ctx — caller must pair this with a matching Response call before issuing another async request.

A.2.2.64 function wh_Client_RsaFunctionResponse

```
int wh_Client_RsaFunctionResponse(
    whClientContext * ctx,
    uint8_t * out,
    uint16_t * inout_out_len
)
```

Receive the reply to an async RSA operation.

Parameters:

- **ctx** Client context.
- **out** Buffer for the output. May be NULL to discard the bytes (the server has already done the work — this is not a pre-flight size query). If non-NULL, inout_out_len must be non-NULL.
- **inout_out_len** In: capacity of out (when out is non-NULL). Out: bytes written, or required size on WH_ERROR_BUFFER_SIZE. May be NULL when out is NULL to discard the count.

Return: WH_ERROR_OK on success, WH_ERROR_NOTREADY if no reply yet, WH_ERROR_BUFFER_SIZE if out is too small, WH_ERROR_BADARGS for invalid args, or a negative error from the transport.

Single-shot receive: returns WH_ERROR_NOTREADY if no reply yet. On success, copies the output into out and writes the byte count to *inout_out_len. If the output is larger than* inout_out_len, returns WH_ERROR_BUFFER_SIZE with the required size written and out left untouched.

A.2.2.65 function wh_Client_RsaGetSizeRequest

```
int wh_Client_RsaGetSizeRequest(
    whClientContext * ctx,
    whKeyId keyId
)
```

Send an async RSA key-size query.

Parameters:

- **ctx** Client context.
- **keyId** Cached RSA key ID. Must not be erased.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args or erased keyId, or a negative error from the transport.

The key must already be cached on the server; auto-import is only available via the blocking wrapper wh_Client_RsaGetSize.

A.2.2.66 function wh_Client_RsaGetSizeResponse

```
int wh_Client_RsaGetSizeResponse(
    whClientContext * ctx,
    int * out_size
)
```

Receive the reply to an async RSA key-size query.

Parameters:

- **ctx** Client context.
- **out_size** Receives the key size in bytes.

Return: WH_ERROR_OK on success, WH_ERROR_NOTREADY if no reply yet, WH_ERROR_BADARGS for invalid args, or a negative error from the transport.

Single-shot receive: returns WH_ERROR_NOTREADY if no reply yet. On success, writes the key size in bytes to *out_size.

A.2.2.67 function wh_Client_RsaMakeCacheKeyRequest

```
int wh_Client_RsaMakeCacheKeyRequest(
    whClientContext * ctx,
    uint32_t size,
    uint32_t e,
    whKeyId key_id,
    whNvmFlags flags,
    uint32_t label_len,
    uint8_t * label
)
```

Ask the server to generate an RSA key and cache it.

Parameters:

- **ctx** Client context.
- **size** RSA modulus size in bits (e.g. 2048).
- **e** RSA public exponent (e.g. WC_RSA_EXPONENT).
- **key_id** Suggested keyId for the new key, or WH_KEYID_ERASED to let the server choose.
- **flags** NVM flags (must NOT include WH_NVM_FLAGS_EPHEMERAL).
- **label_len** Length of the optional label, 0..WH_NVM_LABEL_LEN.
- **label** Optional label bytes. May be NULL when label_len == 0.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args or EPHEMERAL flag set, or a negative error from the transport.

Rejects WH_NVM_FLAGS_EPHEMERAL — ephemeral keygen belongs to the export pair (wh_Client_RsaMakeExportKey{ Only one async request may be in flight per ctx — caller must pair this with a matching Response call before issuing another.

A.2.2.68 function wh_Client_RsaMakeCacheKeyResponse

```
int wh_Client_RsaMakeCacheKeyResponse(
    whClientContext * ctx,
    whKeyId * out_key_id
)
```

Receive the reply to an async cache-keygen request.

Parameters:

- **ctx** Client context.
- **out_key_id** Receives the assigned key ID.

Return: WH_ERROR_OK on success, WH_ERROR_NOTREADY if no reply yet, WH_ERROR_BADARGS for invalid args, or a negative error from the transport.

Single-shot receive: returns WH_ERROR_NOTREADY if no reply yet. On success, writes the server-assigned key ID to *out_key_id.

A.2.2.69 function wh_Client_RsaMakeExportKeyRequest

```
int wh_Client_RsaMakeExportKeyRequest(
    whClientContext * ctx,
    uint32_t size,
    uint32_t e
)
```

Ask the server to generate an RSA key and return it as DER.

Parameters:

- **ctx** Client context.
- **size** RSA modulus size in bits (e.g. 2048).
- **e** RSA public exponent (e.g. WC_RSA_EXPONENT).

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args, or a negative error from the transport.

The server does NOT cache the key — it emits it back as a DER blob.

A.2.2.70 function wh_Client_RsaMakeExportKeyResponse

```
int wh_Client_RsaMakeExportKeyResponse(
    whClientContext * ctx,
    RsaKey * rsa
)
```

Receive the reply to an async export-keygen request.

Parameters:

- **ctx** Client context.
- **rsa** Caller-initialized RsaKey. Populated with the server-generated key material on success.

Return: WH_ERROR_OK on success, WH_ERROR_NOTREADY if no reply yet, WH_ERROR_BADARGS for invalid args, or a negative error from the transport.

Single-shot receive: returns WH_ERROR_NOTREADY if no reply yet. On success, deserializes the returned PKCS#1 DER blob into rsa.

A.2.2.71 function wh_Client_HkdfMakeCacheKey

```
int wh_Client_HkdfMakeCacheKey(
    whClientContext * ctx,
    int hashType,
    whKeyId keyIdIn,
    const uint8_t * inKey,
    uint32_t inKeySz,
    const uint8_t * salt,
    uint32_t saltSz,
    const uint8_t * info,
    uint32_t infoSz,
    whKeyId * inout_key_id,
    whNvmFlags flags,
    const uint8_t * label,
    uint32_t label_len,
    uint32_t outSz
)
```

Generate HKDF output and store in the server key cache.

Parameters:

- **ctx** Pointer to the client context
- **hashType** Hash type (WC_SHA256, WC_SHA384, WC_SHA512, etc.)
- **keyIdIn** Key ID of input key material from cache. Set to WH_KEYID_ERASED to use inKey/inKeySz instead.
- **inKey** Input keying material (can be NULL if keyIdIn is set)
- **inKeySz** Size of input keying material (must be 0 if using keyIdIn)
- **salt** Optional salt (can be NULL)
- **saltSz** Size of salt (0 if NULL)
- **info** Optional info (can be NULL)
- **infoSz** Size of info (0 if NULL)
- **inout_key_id.** Set to WH_KEYID_ERASED to have the server select a unique id for this key.
- **flags** NVM flags to be associated with the key metadata
- **label** Label to be associated with the key metadata
- **label_len** Size of the label up to WH_NVM_LABEL_SIZE. Set to 0 if not used.
- **outSz** Size of key material to generate and cache

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server to generate HKDF output and store it in the server's key cache. The generated key material is not returned to the client.

A.2.2.72 function wh_Client_HkdfMakeExportKey

```
int wh_Client_HkdfMakeExportKey(
    whClientContext * ctx,
    int hashType,
    whKeyId keyIdIn,
    const uint8_t * inKey,
    uint32_t inKeySz,
    const uint8_t * salt,
    uint32_t saltSz,
    const uint8_t * info,
    uint32_t infoSz,
    uint8_t * out,
    uint32_t outSz
)
```

Generate HKDF output and export to the client.

Parameters:

- **ctx** Pointer to the client context
- **hashType** Hash type (WC_SHA256, WC_SHA384, WC_SHA512, etc.)
- **keyIdIn** Key ID of input key material from cache. Set to WH_KEYID_ERASED to use inKey/inKeySz instead.
- **inKey** Input keying material (can be NULL if keyIdIn is set)
- **inKeySz** Size of input keying material (must be 0 if using keyIdIn)
- **salt** Optional salt (can be NULL)
- **saltSz** Size of salt (0 if NULL)
- **info** Optional info (can be NULL)
- **infoSz** Size of info (0 if NULL)
- **out** Output buffer for key material
- **outSz** Size of output buffer

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server to generate HKDF output and export it to the client, without using any key cache or additional resources

A.2.2.73 function wh_Client_CmacKdfMakeCacheKey

```
int wh_Client_CmacKdfMakeCacheKey(
    whClientContext * ctx,
    whKeyId saltKeyId,
    const uint8_t * salt,
    uint32_t saltSz,
    whKeyId zKeyId,
    const uint8_t * z,
    uint32_t zSz,
    const uint8_t * fixedInfo,
    uint32_t fixedInfoSz,
    whKeyId * inout_key_id,
    whNvmFlags flags,
```

```
    const uint8_t * label,
    uint32_t label_len,
    uint32_t outSz
)
```

Generate CMAC two-step KDF output and store it in the server cache.

Parameters:

- **ctx** Pointer to the client context.
- **saltKeyId** Key ID of the salt material. Set to WH_KEYID_ERASED to use the salt buffer instead.
- **salt** Pointer to the salt buffer. May be NULL when saltKeyId is provided.
- **saltSz** Size of the salt buffer in bytes.
- **zKeyId** Key ID of the Z shared secret. Set to WH_KEYID_ERASED to use the z buffer instead.
- **z** Pointer to the shared secret buffer. May be NULL when zKeyId is provided.
- **zSz** Size of the shared secret buffer in bytes.
- **fixedInfo** Optional fixed info buffer (may be NULL).
- **fixedInfoSz** Size of the fixed info buffer in bytes.
- **inout_key_id** Pointer to the key ID to use or update. Set to WH_KEYID_ERASED to have the server allocate one.
- **flags** NVM flags to associate with the generated key.
- **label** Optional label metadata to store alongside the key.
- **label_len** Length of the optional label in bytes.
- **outSz** Desired size of the derived key material.

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server to run the NIST SP 800-56C two-step CMAC KDF. The derived key material is cached on the server and not returned to the client.

A.2.2.74 function wh_Client_CmacKdfMakeExportKey

```
int wh_Client_CmacKdfMakeExportKey(
    whClientContext * ctx,
    whKeyId saltKeyId,
    const uint8_t * salt,
    uint32_t saltSz,
    whKeyId zKeyId,
    const uint8_t * z,
    uint32_t zSz,
    const uint8_t * fixedInfo,
    uint32_t fixedInfoSz,
    uint8_t * out,
    uint32_t outSz
)
```

Generate CMAC two-step KDF output and export to the client.

Parameters:

- **ctx** Pointer to the client context.
- **saltKeyId** Key ID of the salt material. Set to WH_KEYID_ERASED to use the salt buffer instead.
- **salt** Pointer to the salt buffer. May be NULL when saltKeyId is provided.
- **saltSz** Size of the salt buffer in bytes.
- **zKeyId** Key ID of the Z shared secret. Set to WH_KEYID_ERASED to use the z buffer instead.
- **z** Pointer to the shared secret buffer. May be NULL when zKeyId is provided.
- **zSz** Size of the shared secret buffer in bytes.
- **fixedInfo** Optional fixed info buffer (may be NULL).

- **fixedInfoSz** Size of the fixed info buffer in bytes.
- **out** Output buffer for the derived key material.
- **outSz** Size of the output buffer in bytes.

Return: int Returns 0 on success or a negative error code on failure.

This function requests the server to run the NIST SP 800-56C two-step CMAC KDF and return the derived key material directly to the client.

A.2.2.75 function wh_Client_AesSetKeyId

```
int wh_Client_AesSetKeyId(
    Aes * key,
    whNvmId keyId
)
```

Associates an AES key with a specific key ID.

Parameters:

- **key** Pointer to the AES key structure.
- **keyId** Key ID to be associated with the AES key.

Return: int Returns 0 on success or a negative error code on failure.

This function sets the device context of an AES key to the specified key ID. On the server side, this key ID is used to reference the key stored in the HSM

A.2.2.76 function wh_Client_AesGetKeyId

```
int wh_Client_AesGetKeyId(
    Aes * key,
    whNvmId * outId
)
```

Gets the wolfHSM keyId being used by the wolfCrypt struct.

Parameters:

- **key** Pointer to the AES key structure.
- **outId** Pointer to the key ID to return.

Return: int Returns 0 on success or a negative error code on failure.

This function gets the device context of a AES key that was previously set by either the crypto callback layer or wh_Client_SetKeyAes.

A.2.2.77 function wh_Client_AesCtr

```
int wh_Client_AesCtr(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    uint8_t * out
)
```

Performs an AES-CTR operation.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

Return: int Returns 0 on success or a negative error code on failure.

This function performs an AES-CTR encrypt or decrypt operation on the input data and stores the result in the output buffer.

A.2.2.78 function wh_Client_AesCtrDma

```
int wh_Client_AesCtrDma(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    uint8_t * out
)
```

Performs an AES-CTR operation using DMA.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

Return: int Returns 0 on success or a negative error code on failure.

This function performs an AES-CTR encrypt or decrypt operation on the input data and stores the result in the output buffer using direct memory access when communicating with the wolfHSM server.

A.2.2.79 function wh_Client_AesCtrRequest

```
int wh_Client_AesCtrRequest(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len
)
```

Send an AES-CTR encrypt/decrypt request to the server (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure with key and counter state
- **enc** 1 for encrypt, 0 for decrypt (ignored by CTR, present for API symmetry with the other modes)
- **in** Pointer to the input data (may be NULL only if len == 0)
- **len** Length of the input data in bytes

Return: int Returns 0 on success or a negative error code on failure.

Sends a single AES-CTR request to the server. The key material is read from the Aes struct (set via wc_AesSetKey or wh_Client_AesSetKeyId). The counter state (IV register and partial-block remainder) is carried on the Aes struct across the Request/Response boundary; callers must not mutate the Aes struct between the two halves. Use wh_Client_AesCtrResponse to retrieve the result.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_AesCtrResponse before issuing any other async Request on the same ctx.

A.2.2.80 function wh_Client_AesCtrResponse

```
int wh_Client_AesCtrResponse(
    whClientContext * ctx,
    Aes * aes,
    uint8_t * out,
    uint32_t * out_size
)
```

Receive the server's AES-CTR response (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure (counter state updated)
- **out** Pointer to where the output data is placed. Must not be NULL.
- **out_size** Set to the number of bytes produced. May be NULL.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if the response is not yet available, or a negative error code on failure.

Retrieves the result of a prior wh_Client_AesCtrRequest call. The counter state (IV register and partial-block remainder) in the Aes struct is updated from the server response so subsequent CTR calls continue from the correct counter. Returns WH_ERROR_NOTREADY if the response is not yet available.

A.2.2.81 function wh_Client_AesCtrDmaRequest

```
int wh_Client_AesCtrDmaRequest(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    uint8_t * out
)
```

Send an AES-CTR encrypt/decrypt DMA request to the server (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure with key and counter state
- **enc** 1 for encrypt, 0 for decrypt
- **in** Pointer to the input data (may be NULL only if len == 0)
- **len** Length of the input data in bytes
- **out** Pointer to the output buffer

Return: int Returns 0 on success, WH_ERROR_REQUEST_PENDING if the transport is still busy with a prior request, or a negative error code on failure. On failure any acquired DMA mapping is released before returning.

Performs PRE address translation for the input and output buffers, stashes the translated addresses in ctx->dma.asyncCtx.aes for POST cleanup, and sends the DMA request to the server. Does NOT wait for a reply. Caller must keep in and out valid until the matching wh_Client_AesCtrDmaResponse completes.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_AesCtrDmaResponse before issuing any other async Request on the same ctx.

A.2.2.82 function wh_Client_AesCtrDmaResponse

```
int wh_Client_AesCtrDmaResponse(
    whClientContext * ctx,
    Aes * aes
)
```

Receive the server's AES-CTR DMA response (non-blocking)

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. The output data is written by the server directly to the client buffer passed to wh_Client_AesCtrDmaRequest. POST DMA cleanup for both input and output buffers is performed on every non-NOTREADY return so the client buffer is safe to read regardless of error. The counter state on the Aes struct is updated on success.

A.2.2.83 function wh_Client_AesEcb

```
int wh_Client_AesEcb(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    uint8_t * out
)
```

Performs an AES-ECB operation.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

Return: int Returns 0 on success or a negative error code on failure.

This function performs an AES-ECB encrypt or decrypt operation on the input data and stores the result in the output buffer.

A.2.2.84 function wh_Client_AesEcbDma

```
int wh_Client_AesEcbDma(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
```

```
    uint8_t * out
)
```

Performs an AES-ECB operation using DMA.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

Return: int Returns 0 on success or a negative error code on failure.

This function performs an AES-ECB encrypt or decrypt operation on the input data and stores the result in the output buffer using direct memory access when communicating with the wolfHSM server.

A.2.2.85 function wh_Client_AesEcbRequest

```
int wh_Client_AesEcbRequest(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len
)
```

Send an AES-ECB encrypt/decrypt request to the server (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure with key state
- **enc** 1 for encrypt, 0 for decrypt
- **in** Pointer to the input data (must be block-aligned)
- **len** Length of the input data in bytes (must be a multiple of AES_BLOCK_SIZE)

Return: int Returns 0 on success or a negative error code on failure.

Sends a single AES-ECB request to the server. The key material is read from the Aes struct (set via wc_AesSetKey or wh_Client_AesSetKeyId). Use wh_Client_AesEcbResponse to retrieve the result.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_AesEcbResponse before issuing any other async Request on the same ctx. As a special case, len == 0 is a no-op: this function returns WH_ERROR_OK without sending, and the caller MUST NOT call wh_Client_AesEcbResponse afterwards (no response will arrive).

A.2.2.86 function wh_Client_AesEcbResponse

```
int wh_Client_AesEcbResponse(
    whClientContext * ctx,
    Aes * aes,
    uint8_t * out,
    uint32_t * out_size
)
```

Receive the server's AES-ECB response (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure
- **out** Pointer to where the output data is placed. Must not be NULL.
- **out_size** Set to the number of bytes produced. May be NULL.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if the response is not yet available, or a negative error code on failure.

Retrieves the result of a prior wh_Client_AesEcbRequest call. Returns WH_ERROR_NOTREADY if the response is not yet available.

A.2.2.87 function wh_Client_AesEcbDmaRequest

```
int wh_Client_AesEcbDmaRequest(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    uint8_t * out
)
```

Send an AES-ECB DMA request to the server (non-blocking)

Return: int Returns 0 on success, WH_ERROR_REQUEST_PENDING if the transport is still busy with a prior request, or a negative error code on failure. On failure any acquired DMA mapping is released before returning.

Performs PRE address translation for the input and output buffers, stashes the translated addresses in ctx->dma.asyncCtx.aes for POST cleanup, and sends the DMA request to the server. Does NOT wait for a reply. Caller must keep in and out valid until the matching wh_Client_AesEcbDmaResponse completes.

A.2.2.88 function wh_Client_AesEcbDmaResponse

```
int wh_Client_AesEcbDmaResponse(
    whClientContext * ctx,
    Aes * aes
)
```

Receive the server's AES-ECB DMA response (non-blocking)

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. The output data is written by the server directly to the client buffer passed to wh_Client_AesEcbDmaRequest. POST DMA cleanup for both input and output buffers is performed on every non-NOTREADY return so the client buffer is safe to read regardless of error.

A.2.2.89 function wh_Client_AesCbc

```
int wh_Client_AesCbc(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    uint8_t * out
)
```

Performs an AES-CBC operation.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

Return: int Returns 0 on success or a negative error code on failure.

This function performs an AES-CBC encrypt or decrypt operation on the input data and stores the result in the output buffer.

A.2.2.90 function wh_Client_AesCbcDma

```
int wh_Client_AesCbcDma(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    uint8_t * out
)
```

Performs an AES-CBC operation using DMA.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

Return: int Returns 0 on success or a negative error code on failure.

This function performs an AES-CBC encrypt or decrypt operation on the input data and stores the result in the output buffer using direct memory access when communicating with the wolfHSM server.

A.2.2.91 function wh_Client_AesCbcRequest

```
int wh_Client_AesCbcRequest(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len
)
```

Send an AES-CBC encrypt/decrypt request to the server (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure with key and IV state
- **enc** 1 for encrypt, 0 for decrypt

- **in** Pointer to the input data (must be block-aligned)
- **len** Length of the input data in bytes (must be a multiple of AES_BLOCK_SIZE)

Return: int Returns 0 on success or a negative error code on failure.

Sends a single AES-CBC request to the server. The key material is read from the Aes struct (set via wc_AesSetKey or wh_Client_AesSetKeyId). The IV state on the Aes struct is updated only after the matching wh_Client_AesCbcResponse succeeds; a failed Request leaves aes->reg unchanged so callers can retry.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_AesCbcResponse before issuing any other async Request on the same ctx. As a special case, len == 0 is a no-op: this function returns WH_ERROR_OK without sending, and the caller MUST NOT call wh_Client_AesCbcResponse afterwards (no response will arrive).

A.2.2.92 function wh_Client_AesCbcResponse

```
int wh_Client_AesCbcResponse(
    whClientContext * ctx,
    Aes * aes,
    uint8_t * out,
    uint32_t * out_size
)
```

Receive the server's AES-CBC response (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure (IV updated on encrypt)
- **out** Pointer to where the output data is placed. Must not be NULL.
- **out_size** Set to the number of bytes produced. May be NULL.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if the response is not yet available, or a negative error code on failure.

Retrieves the result of a prior wh_Client_AesCbcRequest call. For encryption, the IV in the Aes struct is updated with the last ciphertext block for CBC chaining. Returns WH_ERROR_NOTREADY if the response is not yet available.

A.2.2.93 function wh_Client_AesCbcDmaRequest

```
int wh_Client_AesCbcDmaRequest(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    uint8_t * out
)
```

Send an AES-CBC DMA request to the server (non-blocking)

Return: int Returns 0 on success, WH_ERROR_REQUEST_PENDING if the transport is still busy with a prior request, or a negative error code on failure. On failure any acquired DMA mapping is released before returning.

Performs PRE address translation for the input and output buffers, stashes the translated addresses in ctx->dma.asyncCtx.aes for POST cleanup, and sends the DMA request to the server. Does NOT wait

for a reply. Caller must keep in and out valid until the matching wh_Client_AesCbcDmaResponse completes.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_AesCbcDmaResponse before issuing any other async Request on the same ctx.

A.2.2.94 function wh_Client_AesCbcDmaResponse

```
int wh_Client_AesCbcDmaResponse(
    whClientContext * ctx,
    Aes * aes
)
```

Receive the server's AES-CBC DMA response (non-blocking)

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. The output data is written by the server directly to the client buffer passed to wh_Client_AesCbcDmaRequest; the updated IV is returned inline and copied back onto the Aes struct for CBC chaining. POST DMA cleanup for both input and output buffers is performed on every non-NOTREADY return so the client buffer is safe to read regardless of error.

A.2.2.95 function wh_Client_AesGcm

```
int wh_Client_AesGcm(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    const uint8_t * iv,
    uint32_t iv_len,
    const uint8_t * authin,
    uint32_t authin_len,
    const uint8_t * dec_tag,
    uint8_t * enc_tag,
    uint32_t tag_len,
    uint8_t * out
)
```

Performs an AES-GCM operation.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **iv** Pointer to the IV data.
- **iv_len** Length of the IV data in bytes.
- **authin** Pointer to the authentication data.
- **authin_len** Length of the authentication data in bytes.
- **dec_tag** Pointer to the decryption tag data.
- **enc_tag** Pointer to the encryption tag data.
- **tag_len** Length of the tag data in bytes.
- **out** Pointer to the output data.

Return: int Returns 0 on success or a negative error code on failure.

This function performs an AES-GCM encrypt or decrypt operation on the input data and stores the result in the output buffer.

A.2.2.96 function wh_Client_AesGcmDma

```
int wh_Client_AesGcmDma(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    const uint8_t * iv,
    uint32_t iv_len,
    const uint8_t * authin,
    uint32_t authin_len,
    const uint8_t * dec_tag,
    uint8_t * enc_tag,
    uint32_t tag_len,
    uint8_t * out
)
```

Performs an AES-GCM operation using DMA.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **iv** Pointer to the IV data.
- **iv_len** Length of the IV data in bytes.
- **authin** Pointer to the authentication data.
- **authin_len** Length of the authentication data in bytes.
- **dec_tag** Pointer to the decryption tag data.
- **enc_tag** Pointer to the encryption tag data.
- **tag_len** Length of the tag data in bytes.
- **out** Pointer to the output data.

Return: int Returns 0 on success or a negative error code on failure.

This function performs an AES-GCM encrypt or decrypt operation on the input data and stores the result in the output buffer using direct memory access when communicating with the wolfHSM server.

A.2.2.97 function wh_Client_AesGcmRequest

```
int wh_Client_AesGcmRequest(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    const uint8_t * iv,
    uint32_t iv_len,
    const uint8_t * authin,
```

```
    uint32_t authin_len,
    const uint8_t * dec_tag,
    uint32_t tag_len
)
```

Send an AES-GCM encrypt/decrypt request to the server (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure with key state
- **enc** 1 for encrypt, 0 for decrypt
- **in** Pointer to the input data (may be NULL only if len == 0)
- **len** Length of the input data in bytes
- **iv** Pointer to the IV (may be NULL only if iv_len == 0)
- **iv_len** Length of the IV in bytes
- **authin** Pointer to the AAD (may be NULL only if authin_len == 0)
- **authin_len** Length of the AAD in bytes
- **dec_tag** For decrypt: pointer to the expected auth tag (NULL only if enc == 1). Ignored for encrypt.
- **tag_len** Length of the auth tag in bytes

Return: int Returns 0 on success or a negative error code on failure.

Sends a single AES-GCM request to the server. The key material is read from the Aes struct (set via wc_AesSetKey or wh_Client_AesSetKeyId). The ciphertext, AAD, and decrypt tag (if any) are inlined in the request. Use wh_Client_AesGcmResponse to retrieve the output ciphertext/plaintext and (for encrypt) the auth tag.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_AesGcmResponse before issuing any other async Request on the same ctx.

A.2.2.98 function wh_Client_AesGcmResponse

```
int wh_Client_AesGcmResponse(
    whClientContext * ctx,
    Aes * aes,
    uint8_t * out,
    uint32_t out_capacity,
    uint32_t * out_size,
    uint8_t * enc_tag,
    uint32_t tag_len
)
```

Receive the server's AES-GCM response (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure
- **out** Pointer to where the output data is placed. May be NULL for GMAC (tag-only) operations, in which case out_capacity must be 0.
- **out_capacity** Capacity of the out buffer in bytes. If the server reports a larger payload, the call returns WH_ERROR_ABORTED instead of writing past out.
- **out_size** Set to the number of bytes produced. May be NULL.
- **enc_tag** For encrypt: buffer to receive the auth tag. Ignored for decrypt (may be NULL).
- **tag_len** Length of the enc_tag buffer in bytes.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if the response is not yet available, WH_ERROR_ABORTED if the server's reported payload size exceeds out_capacity, or a negative error

code on failure.

Retrieves the result of a prior wh_Client_AesGcmRequest call. For encrypt, the auth tag is copied into enc_tag. Returns WH_ERROR_NOTREADY if the response is not yet available. For decrypt, a failing tag comparison is surfaced as a negative error from the server (AES_GCM_AUTH_E).

A.2.2.99 function wh_Client_AesGcmDmaRequest

```
int wh_Client_AesGcmDmaRequest(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    uint8_t * out,
    const uint8_t * iv,
    uint32_t iv_len,
    const uint8_t * authin,
    uint32_t authin_len,
    const uint8_t * dec_tag,
    uint32_t tag_len
)
```

Send an AES-GCM DMA request to the server (non-blocking)

Return: int Returns 0 on success, WH_ERROR_REQUEST_PENDING if the transport is still busy with a prior request, or a negative error code on failure. On failure any acquired DMA mapping is released before returning.

Performs PRE address translation for the input, output, and AAD buffers, stashes the translated addresses in ctx->dma.asyncCtx.aes for POST cleanup, and sends the DMA request to the server. Does NOT wait for a reply. The IV, auth tag (for decrypt), and key are passed inline. Caller must keep in, out, and authin valid until the matching wh_Client_AesGcmDmaResponse completes.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_AesGcmDmaResponse before issuing any other async Request on the same ctx.

A.2.2.100 function wh_Client_AesGcmDmaResponse

```
int wh_Client_AesGcmDmaResponse(
    whClientContext * ctx,
    Aes * aes,
    uint8_t * enc_tag,
    uint32_t tag_len
)
```

Receive the server's AES-GCM DMA response (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure
- **enc_tag** For encrypt: buffer to receive the auth tag. Ignored for decrypt (may be NULL).
- **tag_len** Length of the enc_tag buffer in bytes.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. The output data is written by the server directly to the client buffer passed to wh_Client_AesGcmDmaRequest; for encrypt the auth tag is returned inline and copied into enc_tag. POST DMA cleanup for input, output, and AAD buffers is performed on every non-NOTREADY return.

A.2.2.101 function wh_Client_Cmac

```
int wh_Client_Cmac(
    whClientContext * ctx,
    Cmac * cmac,
    CmacType type,
    const uint8_t * key,
    uint32_t keyLen,
    const uint8_t * in,
    uint32_t inLen,
    uint8_t * outMac,
    uint32_t * outMacLen
)
```

Performs a CMAC operation on the input data.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **cmac** Pointer to the CMAC structure.
- **type** The type of CMAC operation.
- **key** Pointer to the key buffer, or NULL if using a key stored in HSM.
- **keyLen** Length of the key in bytes.
- **in** Pointer to the input data buffer, or NULL for finalization.
- **inLen** Length of the input data in bytes.
- **outMac** Pointer to the output buffer for the CMAC tag.
- **outMacLen** Pointer to the size of the output buffer, updated with actual size.

Return: int Returns WH_ERROR_OK (0) on success, or a negative error code on failure.

This function performs a CMAC operation with the specified parameters. It can be used for initialization, update, or finalization of CMAC operations, depending on the input arguments.

A.2.2.102 function wh_Client_CmacGenerateRequest

```
int wh_Client_CmacGenerateRequest(
    whClientContext * ctx,
    Cmac * cmac,
    CmacType type,
    const uint8_t * key,
    uint32_t keyLen,
    const uint8_t * in,
    uint32_t inLen,
    uint32_t outMacLen
)
```

Async request half of a non-DMA CMAC oneshot generate.

Parameters:

- **ctx** Client context.
- **cmac** CMAC context (type and non-HSM key bytes are cached on success).
- **type** CMAC type (e.g., WC_CMAC_AES).
- **key** Inline key bytes, or NULL if using a cached/HSM key.
- **keyLen** Key length in bytes (0 if using a cached/HSM key).
- **in** Input data. Must not be NULL.
- **inLen** Input length. Must be > 0 and must not exceed WH_MESSAGE_CRYPTO_CMAC_MAX_INLINE_GENERATE_SZ.
- **outMacLen** Requested MAC length in bytes. Must be > 0.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS for invalid args or oversize input, or a negative error from the transport. On any error the cmac struct is left unchanged.

Serializes and sends a single request that performs init + update + final on the server in one round trip. The server returns the MAC in the matching Response. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_CmacGenerateResponse before issuing any other async Request on the same ctx. Any existing streaming state in the cmac struct is silently reset — this is a oneshot, equivalent to wc_AesCmacGenerate_ex.

A.2.2.103 function wh_Client_CmacGenerateResponse

```
int wh_Client_CmacGenerateResponse(
    whClientContext * ctx,
    Cmac * cmac,
    uint8_t * outMac,
    uint32_t * outMacLen
)
```

Async response half of a non-DMA CMAC oneshot generate.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, restores state from the response, copies the MAC into outMac (truncated to *outMacLen) and updates* outMacLen to the actual number of bytes written.

A.2.2.104 function wh_Client_CmacUpdateRequest

```
int wh_Client_CmacUpdateRequest(
    whClientContext * ctx,
    Cmac * cmac,
    CmacType type,
    const uint8_t * key,
    uint32_t keyLen,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a non-DMA CMAC streaming Update.

Parameters:

- **ctx** Client context.
- **cmac** CMAC context (full state round-tripped on success, type and cached key bytes updated on SendRequest success).
- **type** CMAC type (written to cmac->type on success).
- **key** Optional inline key bytes (NULL for cached/HSM key).
- **keyLen** Key length in bytes (must not exceed AES_256_KEY_SIZE; 0 for cached/HSM key).
- **in** Input data (may be NULL only if inLen == 0).
- **inLen** Input length. Must fit in the comm buffer alongside the request header and key bytes.
- **requestSent** Set to true if a server request was sent and a matching Response call is required; false only when inLen == 0 and keyLen == 0 (no-op).

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS on invalid arguments or when inLen exceeds the per-call capacity.

Serializes and sends an Update request carrying inLen bytes of inline input plus the full CMAC state (digest + buffer + bookkeeping) via resumeState. The server runs wc_CmacUpdate against the round-

tripped state, so all partial-block accounting happens server-side and the post-Update state is returned in the matching Response. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext (enforced by the comm layer). If *requestSent is true, the caller MUST call wh_Client_CmacUpdateResponse before issuing any other async Request on the same ctx.

Key handling: if key/keyLen are provided, the bytes are cached client-side so subsequent Update/Final calls can replay them. If using an HSM-cached key, set it via wh_Client_CmacSetKeyId before the first Update and pass NULL / 0 for key/keyLen.

A.2.2.105 function wh_Client_CmacUpdateResponse

```
int wh_Client_CmacUpdateResponse(
    whClientContext * ctx,
    Cmac * cmac
)
```

Async response half of a non-DMA CMAC streaming Update.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, restores the full CMAC state (buffer, bufferSz, digest, totalSz) from the response — the server may leave a partial or whole block in its buffer after wc_CmacUpdate (CMAC's last block has special handling), so that bookkeeping is round-tripped back to the client. MUST only be called if the matching Request returned requestSent == true.

A.2.2.106 function wh_Client_CmacFinalRequest

```
int wh_Client_CmacFinalRequest(
    whClientContext * ctx,
    Cmac * cmac
)
```

Async request half of a non-DMA CMAC streaming Final.

Sends a Final request with no inline input — the round-tripped resumeState carries the current cmac->buffer (0..AES_BLOCK_SIZE-1 bytes) as the trailing partial block for the server to finalize. Key material travels with the request when available.

A.2.2.107 function wh_Client_CmacFinalResponse

```
int wh_Client_CmacFinalResponse(
    whClientContext * ctx,
    Cmac * cmac,
    uint8_t * outMac,
    uint32_t * outMacLen
)
```

Async response half of a non-DMA CMAC streaming Final.

Single-shot RecvResponse. Restores final state from the response, then copies the MAC into outMac (truncated to *outMacLen) and updates* outMacLen.

A.2.2.108 function wh_Client_CmacSetKeyId

```
int wh_Client_CmacSetKeyId(
    Cmac * key,
    whNvmId keyId
)
```

Associates a CMAC key with a specific key ID.

Parameters:

- **key** Pointer to the CMAC key structure.
- **keyId** Key ID to be associated with the CMAC key.

Return: int Returns 0 on success or a negative error code on failure.

This function sets the device context of a CMAC key to the specified key ID. On the server side, this key ID is used to reference the key stored in the HSM

A.2.2.109 function wh_Client_CmacGetKeyId

```
int wh_Client_CmacGetKeyId(
    Cmac * key,
    whNvmId * outId
)
```

Gets the wolfHSM keyId being used by the wolfCrypt struct.

Parameters:

- **key** Pointer to the CMAC key structure.
- **outId** Pointer to the key ID to return.

Return: int Returns 0 on success or a negative error code on failure.

This function gets the device context of a CMAC key that was previously set by either the crypto callback layer or wh_Client_SetKeyCmac.

A.2.2.110 function wh_Client_CmacDma

```
int wh_Client_CmacDma(
    whClientContext * ctx,
    Cmac * cmac,
    CmacType type,
    const uint8_t * key,
    uint32_t keyLen,
    const uint8_t * in,
    uint32_t inLen,
    uint8_t * outMac,
    uint32_t * outMacLen
)
```

Performs CMAC operations using DMA for data transfer.

Parameters:

- **ctx** Pointer to the client context structure.
- **cmac** Pointer to the CMAC structure to be used.
- **type** The type of CMAC operation (e.g., WC_CMAC_AES).
- **key** Pointer to the key data. NULL if using a stored key.
- **keyLen** Length of the key in bytes.
- **in** Pointer to the input data. NULL if not performing an update.
- **inLen** Length of the input data in bytes.
- **outMac** Pointer to store the CMAC result. NULL if not finalizing.
- **outMacLen** Pointer to the size of the outMac buffer. Updated with actual size on return.

Return: int Returns 0 on success or a negative error code on failure.

This function performs CMAC operations (initialize, update, finalize) using DMA for efficient data transfer between client and server. The operation performed depends on which parameters are non-NULL.

A.2.2.111 function wh_Client_CmacGenerateDmaRequest

```
int wh_Client_CmacGenerateDmaRequest(
    whClientContext * ctx,
    Cmac * cmac,
    CmacType type,
    const uint8_t * key,
    uint32_t keyLen,
    const uint8_t * in,
    uint32_t inLen,
    uint32_t outMacLen
)
```

Async request half of a DMA CMAC oneshot generate.

Performs PRE address translation for the input buffer, sends the DMA request, and stashes the translated address for POST cleanup in the matching Response. Does NOT wait for a reply. The server processes the oneshot in a single round trip via wc_AesCmacGenerate_ex.

Contract: at most one outstanding async request may be in flight per whClientContext. The caller MUST call wh_Client_CmacGenerateDmaResponse before issuing any other async Request on the same ctx, and must keep in valid until the Response completes. Any existing streaming state in the cmac struct is silently reset — this is a oneshot, equivalent to wc_AesCmacGenerate_ex.

A.2.2.112 function wh_Client_CmacGenerateDmaResponse

```
int wh_Client_CmacGenerateDmaResponse(
    whClientContext * ctx,
    Cmac * cmac,
    uint8_t * outMac,
    uint32_t * outMacLen
)
```

Async response half of a DMA CMAC oneshot generate.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On any non-NOTREADY exit, performs POST DMA cleanup on the input buffer. On success, copies the MAC into outMac (truncated to *outMacLen), updates* outMacLen, and restores the post-finalization CMAC state (buffer, bufferSz, digest, totalSz) from the response. The AES round key and CMAC subkey material in cmac are NOT reset — callers recycling the cmac struct must reinitialize it via wc_InitCmac_ex.

A.2.2.113 function wh_Client_CmacDmaUpdateRequest

```
int wh_Client_CmacDmaUpdateRequest(
    whClientContext * ctx,
    Cmac * cmac,
    CmacType type,
    const uint8_t * key,
    uint32_t keyLen,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a DMA CMAC streaming Update.

Performs PRE address translation for the input buffer, round-trips the full CMAC state to the server via resumeState, and sends every byte of the input via DMA. No client-side partial-block buffering and no inline trailing data — the server runs wc_CmacUpdate against the round-tripped state. Stashes the translated input address for POST cleanup in the matching Response. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext. If *requestSent is true, the caller MUST keep in valid and call wh_Client_CmacDmaUpdateResponse before issuing any other async Request.* requestSent is false only when inLen == 0 and keyLen == 0 (no-op).

A.2.2.114 function wh_Client_CmacDmaUpdateResponse

```
int wh_Client_CmacDmaUpdateResponse(
    whClientContext * ctx,
    Cmac * cmac
)
```

Async response half of a DMA CMAC streaming Update.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On any non-NOTREADY exit, performs POST DMA cleanup for the input buffer. On success, restores the full CMAC state (buffer, bufferSz, digest, totalSz) from the response — including any partial/whole block left in the server's wc_CmacUpdate buffer.

A.2.2.115 function wh_Client_CmacDmaFinalRequest

```
int wh_Client_CmacDmaFinalRequest(
    whClientContext * ctx,
    Cmac * cmac
)
```

Async request half of a DMA CMAC streaming Final.

Sends a Final request with no DMA addresses and no inline input — the round-tripped resumeState carries the partial-block tail (0..AES_BLOCK_SIZE-1 bytes) for the server to finalize. Key material travels with the request when available.

A.2.2.116 function wh_Client_CmacDmaFinalResponse

```
int wh_Client_CmacDmaFinalResponse(
    whClientContext * ctx,
    Cmac * cmac,
    uint8_t * outMac,
    uint32_t * outMacLen
)
```

Async response half of a DMA CMAC streaming Final.

Single-shot RecvResponse. Copies the MAC into outMac (truncated to *outMacLen), updates* outMacLen, and restores the post-finalization CMAC state (buffer, bufferSz, digest, totalSz) from the response. The AES round key and CMAC subkey material in cmac are NOT reset — callers recycling the cmac struct must reinitialize it via wc_InitCmac_ex. No DMA cleanup is needed (Final doesn't use DMA addresses).

A.2.2.117 function wh_Client_Sha256

```
int wh_Client_Sha256(
    whClientContext * ctx,
```

```
    wc_Sha256 * sha,
    const uint8_t * in,
    uint32_t inLen,
    uint8_t * out
)
```

Performs a SHA-256 hash operation on the input data.

Parameters:

- **ctx** Pointer to the client context structure.
- **sha** Pointer to the SHA-256 context structure.
- **in** Pointer to the input data.
- **inLen** Length of the input data in bytes.
- **out** Pointer to the output buffer.

Return: int Returns 0 on success or a negative error code on failure.

This function performs a SHA-256 hash operation on the input data and stores the result in the output buffer.

A.2.2.118 function wh_Client_Sha256UpdateRequest

```
int wh_Client_Sha256UpdateRequest(
    whClientContext * ctx,
    wc_Sha256 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a non-DMA SHA-256 Update.

Parameters:

- **ctx** Client context.
- **sha** SHA-256 context (buffer/buffLen updated on success).
- **in** Input data (may be NULL only if inLen == 0).
- **inLen** Input length. Must not exceed the per-call capacity (max inline + remaining buffer slack); use the blocking wrapper for arbitrary lengths.
- **requestSent** Set to true if a server request was sent and a matching Response call is required; false if the input was fully absorbed into sha->buffer and no round-trip was issued.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if inLen exceeds the per-call capacity (sha is left unchanged in that case).

Serializes and sends an Update request carrying as many full blocks as fit in the comm buffer (up to WH_MESSAGE_CRYPTO_SHA256_MAX_INLINE_UPDATE_SZ bytes), absorbing any leading bytes already buffered in sha->buffer. Any tail (<64 bytes) remaining after this call is stored in sha->buffer for the next call. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext (enforced by the comm layer's pending-request tracking). If *requestSent is true, the caller MUST call wh_Client_Sha256UpdateResponse before issuing any other async Request on the same ctx, including a Request using a different wc_Sha256 instance or a different algorithm.

A.2.2.119 function wh_Client_Sha256UpdateResponse

```
int wh_Client_Sha256UpdateResponse(
    whClientContext * ctx,
```

```
    wc_Sha256 * sha
)
```

Async response half of a non-DMA SHA-256 Update.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, updates sha->digest/hiLen/loLen from the reply. MUST only be called if the matching Request returned requestSent == true.

A.2.2.120 function wh_Client_Sha256FinalRequest

```
int wh_Client_Sha256FinalRequest(
    whClientContext * ctx,
    wc_Sha256 * sha
)
```

Async request half of a non-DMA SHA-256 Final.

Sends the current sha->buffer (0..63 bytes) as the last block.

A.2.2.121 function wh_Client_Sha256FinalResponse

```
int wh_Client_Sha256FinalResponse(
    whClientContext * ctx,
    wc_Sha256 * sha,
    uint8_t * out
)
```

Async response half of a non-DMA SHA-256 Final.

Single-shot RecvResponse. Copies final digest into out, then resets sha state via wc_InitSha256_ex (preserving devId).

A.2.2.122 function wh_Client_Sha256Dma

```
int wh_Client_Sha256Dma(
    whClientContext * ctx,
    wc_Sha256 * sha,
    const uint8_t * in,
    uint32_t inLen,
    uint8_t * out
)
```

Performs a SHA-256 hash operation on the input data using DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **sha** Pointer to the SHA-256 context structure.
- **in** Pointer to the input data.
- **inLen** Length of the input data in bytes.
- **out** Pointer to the output buffer.

Return: int Returns 0 on success or a negative error code on failure.

This function performs a SHA-256 hash operation on the input data and stores the result in the output buffer using DMA.

A.2.2.123 function wh_Client_Sha256DmaUpdateRequest

```
int wh_Client_Sha256DmaUpdateRequest(
    whClientContext * ctx,
    wc_Sha256 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a DMA SHA-256 Update.

Buffers partial blocks on the client. Sends whole blocks via DMA to the server, with any assembled first block (from the partial buffer) as inline trailing data. Sets *requestSent to indicate whether a message was sent (false when all input was absorbed into the partial-block buffer).

A.2.2.124 function wh_Client_Sha256DmaUpdateResponse

```
int wh_Client_Sha256DmaUpdateResponse(
    whClientContext * ctx,
    wc_Sha256 * sha
)
```

Async response half of a DMA SHA-256 Update.

Receives the server response and restores the updated SHA state from the inline response. Runs POST DMA cleanup for the input buffer.

A.2.2.125 function wh_Client_Sha256DmaFinalRequest

```
int wh_Client_Sha256DmaFinalRequest(
    whClientContext * ctx,
    wc_Sha256 * sha
)
```

Async request half of a DMA SHA-256 Final.

Sends the partial-block tail as inline data with the resume state. No DMA addresses are used (the final hash is returned inline in the response).

A.2.2.126 function wh_Client_Sha256DmaFinalResponse

```
int wh_Client_Sha256DmaFinalResponse(
    whClientContext * ctx,
    wc_Sha256 * sha,
    uint8_t * out
)
```

Async response half of a DMA SHA-256 Final.

Receives the final hash from the inline response and copies it to out.

A.2.2.127 function wh_Client_Sha224

```
int wh_Client_Sha224(
    whClientContext * ctx,
    wc_Sha224 * sha,
    const uint8_t * in,
    uint32_t inLen,
```

```
    uint8_t * out
)
```

Performs a SHA-224 hash operation on the input data.

Parameters:

- **ctx** Pointer to the client context structure.
- **sha** Pointer to the SHA-224 context structure.
- **in** Pointer to the input data.
- **inLen** Length of the input data in bytes.
- **out** Pointer to the output buffer.

Return: int Returns 0 on success or a negative error code on failure.

This function performs a SHA-224 hash operation on the input data and stores the result in the output buffer.

A.2.2.128 function wh_Client_Sha224UpdateRequest

```
int wh_Client_Sha224UpdateRequest(
    whClientContext * ctx,
    wc_Sha224 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a non-DMA SHA-224 Update.

Parameters:

- **ctx** Client context.
- **sha** SHA-224 context (buffer/buffLen updated on success).
- **in** Input data (may be NULL only if inLen == 0).
- **inLen** Input length. Must not exceed the per-call capacity (max inline + remaining buffer slack); use the blocking wrapper for arbitrary lengths.
- **requestSent** Set to true if a server request was sent and a matching Response call is required; false if the input was fully absorbed into sha->buffer and no round-trip was issued.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if inLen exceeds the per-call capacity (sha is left unchanged in that case).

Serializes and sends an Update request carrying as many full blocks as fit in the comm buffer (up to WH_MESSAGE_CRYPTO_SHA224_MAX_INLINE_UPDATE_SZ bytes), absorbing any leading bytes already buffered in sha->buffer. Any tail (<64 bytes) remaining after this call is stored in sha->buffer for the next call. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext (enforced by the comm layer's pending-request tracking). If *requestSent is true, the caller MUST call wh_Client_Sha224UpdateResponse before issuing any other async Request on the same ctx, including a Request using a different wc_Sha224 instance or a different algorithm.

A.2.2.129 function wh_Client_Sha224UpdateResponse

```
int wh_Client_Sha224UpdateResponse(
    whClientContext * ctx,
    wc_Sha224 * sha
)
```

Async response half of a non-DMA SHA-224 Update.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, updates sha->digest/hiLen/loLen from the reply. MUST only be called if the matching Request returned requestSent == true.

A.2.2.130 function wh_Client_Sha224FinalRequest

```
int wh_Client_Sha224FinalRequest(
    whClientContext * ctx,
    wc_Sha224 * sha
)
```

Async request half of a non-DMA SHA-224 Final.

Sends the current sha->buffer (0..63 bytes) as the last block.

A.2.2.131 function wh_Client_Sha224FinalResponse

```
int wh_Client_Sha224FinalResponse(
    whClientContext * ctx,
    wc_Sha224 * sha,
    uint8_t * out
)
```

Async response half of a non-DMA SHA-224 Final.

Single-shot RecvResponse. Copies final digest into out, then resets sha state via wc_InitSha224_ex (preserving devId).

A.2.2.132 function wh_Client_Sha224Dma

```
int wh_Client_Sha224Dma(
    whClientContext * ctx,
    wc_Sha224 * sha,
    const uint8_t * in,
    uint32_t inLen,
    uint8_t * out
)
```

Performs a SHA-224 hash operation on the input data using DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **sha** Pointer to the SHA-224 context structure.
- **in** Pointer to the input data.
- **inLen** Length of the input data in bytes.
- **out** Pointer to the output buffer.

Return: int Returns 0 on success or a negative error code on failure.

This function performs a SHA-224 hash operation on the input data and stores the result in the output buffer using DMA.

A.2.2.133 function wh_Client_Sha224DmaUpdateRequest

```
int wh_Client_Sha224DmaUpdateRequest(
    whClientContext * ctx,
```

```
    wc_Sha224 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

A.2.2.134 function wh_Client_Sha224DmaUpdateResponse

```
int wh_Client_Sha224DmaUpdateResponse(
    whClientContext * ctx,
    wc_Sha224 * sha
)
```

A.2.2.135 function wh_Client_Sha224DmaFinalRequest

```
int wh_Client_Sha224DmaFinalRequest(
    whClientContext * ctx,
    wc_Sha224 * sha
)
```

A.2.2.136 function wh_Client_Sha224DmaFinalResponse

```
int wh_Client_Sha224DmaFinalResponse(
    whClientContext * ctx,
    wc_Sha224 * sha,
    uint8_t * out
)
```

A.2.2.137 function wh_Client_Sha384

```
int wh_Client_Sha384(
    whClientContext * ctx,
    wc_Sha384 * sha,
    const uint8_t * in,
    uint32_t inLen,
    uint8_t * out
)
```

Performs a SHA-384 hash operation on the input data.

Parameters:

- **ctx** Pointer to the client context structure.
- **sha** Pointer to the SHA-384 context structure.
- **in** Pointer to the input data.
- **inLen** Length of the input data in bytes.
- **out** Pointer to the output buffer.

Return: int Returns 0 on success or a negative error code on failure.

This function performs a SHA-384 hash operation on the input data and stores the result in the output buffer.

A.2.2.138 function wh_Client_Sha384UpdateRequest

```
int wh_Client_Sha384UpdateRequest(
    whClientContext * ctx,
    wc_Sha384 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a non-DMA SHA-384 Update.

Parameters:

- **ctx** Client context.
- **sha** SHA-384 context (buffer/buffLen updated on success).
- **in** Input data (may be NULL only if inLen == 0).
- **inLen** Input length. Must not exceed the per-call capacity (max inline + remaining buffer slack); use the blocking wrapper for arbitrary lengths.
- **requestSent** Set to true if a server request was sent and a matching Response call is required; false if the input was fully absorbed into sha->buffer and no round-trip was issued.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if inLen exceeds the per-call capacity (sha is left unchanged in that case).

Serializes and sends an Update request carrying as many full blocks as fit in the comm buffer (up to WH_MESSAGE_CRYPTO_SHA384_MAX_INLINE_UPDATE_SZ bytes), absorbing any leading bytes already buffered in sha->buffer. Any tail (<128 bytes) remaining after this call is stored in sha->buffer for the next call. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext (enforced by the comm layer's pending-request tracking). If *requestSent is true, the caller MUST call wh_Client_Sha384UpdateResponse before issuing any other async Request on the same ctx, including a Request using a different wc_Sha384 instance or a different algorithm.

A.2.2.139 function wh_Client_Sha384UpdateResponse

```
int wh_Client_Sha384UpdateResponse(
    whClientContext * ctx,
    wc_Sha384 * sha
)
```

Async response half of a non-DMA SHA-384 Update.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, updates sha->digest/hiLen/loLen from the reply. MUST only be called if the matching Request returned requestSent == true.

A.2.2.140 function wh_Client_Sha384FinalRequest

```
int wh_Client_Sha384FinalRequest(
    whClientContext * ctx,
    wc_Sha384 * sha
)
```

Async request half of a non-DMA SHA-384 Final.

Sends the current sha->buffer (0..127 bytes) as the last block.

A.2.2.141 function wh_Client_Sha384FinalResponse

```
int wh_Client_Sha384FinalResponse(
    whClientContext * ctx,
    wc_Sha384 * sha,
    uint8_t * out
)
```

Async response half of a non-DMA SHA-384 Final.

Single-shot RecvResponse. Copies final digest into out, then resets sha state via wc_InitSha384_ex (preserving devId).

A.2.2.142 function wh_Client_Sha384Dma

```
int wh_Client_Sha384Dma(
    whClientContext * ctx,
    wc_Sha384 * sha,
    const uint8_t * in,
    uint32_t inLen,
    uint8_t * out
)
```

Performs a SHA-384 hash operation on the input data using DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **sha** Pointer to the SHA-384 context structure.
- **in** Pointer to the input data.
- **inLen** Length of the input data in bytes.
- **out** Pointer to the output buffer.

Return: int Returns 0 on success or a negative error code on failure.

This function performs a SHA-384 hash operation on the input data and stores the result in the output buffer using DMA.

A.2.2.143 function wh_Client_Sha384DmaUpdateRequest

```
int wh_Client_Sha384DmaUpdateRequest(
    whClientContext * ctx,
    wc_Sha384 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

A.2.2.144 function wh_Client_Sha384DmaUpdateResponse

```
int wh_Client_Sha384DmaUpdateResponse(
    whClientContext * ctx,
    wc_Sha384 * sha
)
```

A.2.2.145 function wh_Client_Sha384DmaFinalRequest

```
int wh_Client_Sha384DmaFinalRequest(
    whClientContext * ctx,
    wc_Sha384 * sha
)
```

A.2.2.146 function wh_Client_Sha384DmaFinalResponse

```
int wh_Client_Sha384DmaFinalResponse(
    whClientContext * ctx,
    wc_Sha384 * sha,
    uint8_t * out
)
```

A.2.2.147 function wh_Client_Sha512

```
int wh_Client_Sha512(
    whClientContext * ctx,
    wc_Sha512 * sha,
    const uint8_t * in,
    uint32_t inLen,
    uint8_t * out
)
```

Performs a SHA-512 hash operation on the input data.

Parameters:

- **ctx** Pointer to the client context structure.
- **sha** Pointer to the SHA-512 context structure.
- **in** Pointer to the input data.
- **inLen** Length of the input data in bytes.
- **out** Pointer to the output buffer.

Return: int Returns 0 on success or a negative error code on failure.

This function performs a SHA-512 hash operation on the input data and stores the result in the output buffer.

A.2.2.148 function wh_Client_Sha512UpdateRequest

```
int wh_Client_Sha512UpdateRequest(
    whClientContext * ctx,
    wc_Sha512 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a non-DMA SHA-512 Update.

Parameters:

- **ctx** Client context.
- **sha** SHA-512 context (buffer/buffLen updated on success).
- **in** Input data (may be NULL only if inLen == 0).
- **inLen** Input length. Must not exceed the per-call capacity (max inline + remaining buffer slack); use the blocking wrapper for arbitrary lengths.

- **requestSent** Set to true if a server request was sent and a matching Response call is required; false if the input was fully absorbed into sha->buffer and no round-trip was issued.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if inLen exceeds the per-call capacity (sha is left unchanged in that case).

Serializes and sends an Update request carrying as many full blocks as fit in the comm buffer (up to WH_MESSAGE_CRYPTO_SHA512_MAX_INLINE_UPDATE_SZ bytes), absorbing any leading bytes already buffered in sha->buffer. Any tail (<128 bytes) remaining after this call is stored in sha->buffer for the next call. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext (enforced by the comm layer's pending-request tracking). If *requestSent is true, the caller MUST call wh_Client_Sha512UpdateResponse before issuing any other async Request on the same ctx, including a Request using a different wc_Sha512 instance or a different algorithm.

A.2.2.149 function wh_Client_Sha512UpdateResponse

```
int wh_Client_Sha512UpdateResponse(
    whClientContext * ctx,
    wc_Sha512 * sha
)
```

Async response half of a non-DMA SHA-512 Update.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, updates sha->digest/hiLen/loLen from the reply. MUST only be called if the matching Request returned requestSent == true.

A.2.2.150 function wh_Client_Sha512FinalRequest

```
int wh_Client_Sha512FinalRequest(
    whClientContext * ctx,
    wc_Sha512 * sha
)
```

Async request half of a non-DMA SHA-512 Final.

Sends the current sha->buffer (0..127 bytes) as the last block.

A.2.2.151 function wh_Client_Sha512FinalResponse

```
int wh_Client_Sha512FinalResponse(
    whClientContext * ctx,
    wc_Sha512 * sha,
    uint8_t * out
)
```

Async response half of a non-DMA SHA-512 Final.

Single-shot RecvResponse. Copies final digest into out, then resets sha state via wc_InitSha512_ex (preserving devId and hashType).

A.2.2.152 function wh_Client_Sha512Dma

```
int wh_Client_Sha512Dma(
    whClientContext * ctx,
    wc_Sha512 * sha,
    const uint8_t * in,
```

```
    uint32_t inLen,
    uint8_t * out
)
```

Performs a SHA-512 hash operation on the input data using DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **sha** Pointer to the SHA-512 context structure.
- **in** Pointer to the input data.
- **inLen** Length of the input data in bytes.
- **out** Pointer to the output buffer.

Return: int Returns 0 on success or a negative error code on failure.

This function performs a SHA-512 hash operation on the input data and stores the result in the output buffer using DMA.

A.2.2.153 function wh_Client_Sha512DmaUpdateRequest

```
int wh_Client_Sha512DmaUpdateRequest(
    whClientContext * ctx,
    wc_Sha512 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

A.2.2.154 function wh_Client_Sha512DmaUpdateResponse

```
int wh_Client_Sha512DmaUpdateResponse(
    whClientContext * ctx,
    wc_Sha512 * sha
)
```

A.2.2.155 function wh_Client_Sha512DmaFinalRequest

```
int wh_Client_Sha512DmaFinalRequest(
    whClientContext * ctx,
    wc_Sha512 * sha
)
```

A.2.2.156 function wh_Client_Sha512DmaFinalResponse

```
int wh_Client_Sha512DmaFinalResponse(
    whClientContext * ctx,
    wc_Sha512 * sha,
    uint8_t * out
)
```

A.2.2.157 function wh_Client_MlDsaSetKeyId

```
int wh_Client_MlDsaSetKeyId(
    wc_MlDsaKey * key,
    whKeyId keyId
)
```

Associates a ML-DSA key with a specific key ID.

Parameters:

- **key** Pointer to the ML-DSA key structure.
- **keyId** Key ID to be associated with the ML-DSA key.

Return: int Returns 0 on success or a negative error code on failure.

This function sets the device context of a ML-DSA key to the specified key ID. On the server side, this key ID is used to reference the key stored in the HSM

A.2.2.158 function wh_Client_MlDsaGetKeyId

```
int wh_Client_MlDsaGetKeyId(
    wc_MlDsaKey * key,
    whKeyId * outId
)
```

Gets the wolfHSM keyId being used by the wolfCrypt struct.

Parameters:

- **key** Pointer to the ML-DSA key structure.
- **outId** Pointer to the key ID to return.

Return: int Returns 0 on success or a negative error code on failure.

This function gets the device context of a ML-DSA key that was previously set by either the crypto callback layer or wh_Client_MlDsaSetKeyId.

A.2.2.159 function wh_Client_MlDsaImportKey

```
int wh_Client_MlDsaImportKey(
    whClientContext * ctx,
    wc_MlDsaKey * key,
    whKeyId * inout_keyId,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Import a ML-DSA key to the server key cache.

Parameters:

- **ctx** Pointer to the client context
- **key** Pointer to the key to import
- **inout_keyId** Pointer to key ID to use/receive
- **flags** Flags to control key persistence
- **label_len** Length of optional label
- **label** Optional label to associate with key

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.160 function wh_Client_MlDsaExportKey

```
int wh_Client_MlDsaExportKey(
    whClientContext * ctx,
    whKeyId keyId,
    wc_MlDsaKey * key,
```

```
    uint16_t label_len,
    uint8_t * label
)
```

Export a ML-DSA key from the server.

Parameters:

- **ctx** Pointer to the client context
- **keyId** ID of key to export
- **key** Pointer to receive exported key
- **label_len** Length of optional label buffer
- **label** Optional buffer to receive key label

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.161 function wh_Client_MlDsaExportPublicKey

```
int wh_Client_MlDsaExportPublicKey(
    whClientContext * ctx,
    whKeyId keyId,
    wc_MlDsaKey * key,
    uint16_t label_len,
    uint8_t * label
)
```

Exports only the public part of a cached ML-DSA key.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **keyId** Server key ID whose public key should be exported. Must not be WH_KEYID_ERASED.
- **key** Pointer to a caller-initialized wc_MlDsaKey. On success, only the public half is populated (pubKeySet == 1, prvKeySet == 0).
- **label_len** Size of the optional label buffer in bytes. Values larger than WH_NVM_LABEL_LEN are truncated. Set to 0 if label is not needed.
- **label** Optional buffer to receive the key's label. May be NULL.

Return: int Returns 0 on success or a negative wolfHSM/wolfCrypt error code on failure (e.g. WH_ERROR_NOTFOUND, WH_ERROR_BADARGS).

Instructs the server to emit only the public portion of a cached ML-DSA key as SubjectPublicKeyInfo DER. The private key stays inside the HSM. The decoded key will have pubKeySet == 1 and prvKeySet == 0.

The NONEXPORTABLE key flag does NOT block this call because public material is non-sensitive. The caller is responsible for initializing key (e.g. wc_MlDsaKey_Init) and, if required, setting the ML-DSA parameter level via wc_MlDsaKey_SetParams prior to calling this function.

A.2.2.162 function wh_Client_MlDsaMakeExportKey

```
int wh_Client_MlDsaMakeExportKey(
    whClientContext * ctx,
    int level,
    int size,
    wc_MlDsaKey * key
)
```

Generate a new ML-DSA key pair and export the public key.

Parameters:

- **ctx** Pointer to the client context structure.
- **type** The ML-DSA algorithm type.
- **size** Size of the key in bits.
- **key** Pointer to the ML-DSA key structure to store the key.

Return: int Returns 0 on success, or a negative error code on failure.

This function generates a new ML-DSA key pair in the HSM and exports the public key to the client. The private key remains securely stored in the HSM.

A.2.2.163 function wh_Client_MlDsaMakeCacheKey

```
int wh_Client_MlDsaMakeCacheKey(
    whClientContext * ctx,
    int size,
    int level,
    whKeyId * inout_key_id,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Create and cache a new ML-DSA key on the server.

Parameters:

- **ctx** Pointer to the client context
- **size** Size of key to generate
- **level** ML-DSA security level of the key to generate
- **inout_key_id** Pointer to key ID to use/receive
- **flags** Flags to control key persistence
- **label_len** Length of optional label
- **label** Optional label to associate with key

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.164 function wh_Client_MlDsaSign

```
int wh_Client_MlDsaSign(
    whClientContext * ctx,
    const byte * in,
    word32 in_len,
    byte * out,
    word32 * out_len,
    wc_MlDsaKey * key,
    const byte * context,
    byte contextLen,
    word32 preHashType
)
```

Sign a message using a ML-DSA private key.

Parameters:

- **ctx** Pointer to the client context structure.
- **in** Pointer to the message to sign.
- **in_len** Length of the message in bytes.

- **out** Buffer to store the signature.
- **out_len** Pointer to size of output buffer, updated with actual size.
- **key** Pointer to the ML-DSA key structure.
- **context** Optional FIPS 204 context string for domain separation, or NULL for no context.
- **contextLen** Length of the context string (max 255).
- **preHashType** Hash type for HashML-DSA (e.g. WC_HASH_TYPE_SHA256), or WC_HASH_TYPE_NONE for pure ML-DSA.

Return: int Returns 0 on success, or a negative error code on failure.

This function signs a message using a ML-DSA private key stored in the HSM.

A.2.2.165 function wh_Client_MlDsaVerify

```
int wh_Client_MlDsaVerify(
    whClientContext * ctx,
    const byte * sig,
    word32 sig_len,
    const byte * msg,
    word32 msg_len,
    int * res,
    wc_MlDsaKey * key,
    const byte * context,
    byte contextLen,
    word32 preHashType
)
```

Verify a ML-DSA signature.

Parameters:

- **ctx** Pointer to the client context structure.
- **sig** Pointer to the signature to verify.
- **sig_len** Length of the signature in bytes.
- **msg** Pointer to the original message.
- **msg_len** Length of the message in bytes.
- **res** Pointer to store verification result (1=success, 0=failure).
- **key** Pointer to the ML-DSA key structure.
- **context** Optional FIPS 204 context string for domain separation, or NULL for no context.
- **contextLen** Length of the context string (max 255).
- **preHashType** Hash type for HashML-DSA (e.g. WC_HASH_TYPE_SHA256), or WC_HASH_TYPE_NONE for pure ML-DSA.

Return: int Returns 0 on success, or a negative error code on failure.

This function verifies a ML-DSA signature using the HSM.

A.2.2.166 function wh_Client_MlDsaCheckPrivKey

```
int wh_Client_MlDsaCheckPrivKey(
    whClientContext * ctx,
    wc_MlDsaKey * key,
    const byte * pubKey,
    word32 pubKeySz
)
```

Check a ML-DSA private key.

Parameters:

- **ctx** Pointer to the client context structure.
- **key** Pointer to the ML-DSA key structure.
- **pubKey** Pointer to the public key data.
- **pubKeySz** Size of the public key in bytes.

Return: int Returns 0 on success, or a negative error code on failure.

This function validates a ML-DSA private key against its public key using the HSM.

A.2.2.167 function wh_Client_MlDsaImportKeyDma

```
int wh_Client_MlDsaImportKeyDma(
    whClientContext * ctx,
    wc_MlDsaKey * key,
    whKeyId * inout_keyId,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Import a ML-DSA key using DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **key** Pointer to the ML-DSA key structure representing the key to import.
- **inout_keyId** Pointer to store/provide the key ID.
- **flags** NVM flags for key storage.
- **label_len** Length of the key label in bytes.
- **label** Pointer to the key label.

Return: int Returns 0 on success, or a negative error code on failure.

This function imports a ML-DSA key into the HSM using DMA.

A.2.2.168 function wh_Client_MlDsaExportKeyDma

```
int wh_Client_MlDsaExportKeyDma(
    whClientContext * ctx,
    whKeyId keyId,
    wc_MlDsaKey * key,
    uint16_t label_len,
    uint8_t * label
)
```

Export a ML-DSA key using DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **keyId** ID of the key to export.
- **key** Pointer to the ML-DSA key structure to hold the exported key.
- **label_len** Length of the key label in bytes.
- **label** Pointer to the key label.

Return: int Returns 0 on success, or a negative error code on failure.

This function exports a ML-DSA key from the HSM using DMA.

A.2.2.169 function wh_Client_MlDsaExportPublicKeyDma

```
int wh_Client_MlDsaExportPublicKeyDma(
    whClientContext * ctx,
    whKeyId keyId,
    wc_MlDsaKey * key,
    uint16_t label_len,
    uint8_t * label
)
```

Export only the public part of a cached ML-DSA key using DMA.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **keyId** Server key ID whose public key should be exported. Must not be WH_KEYID_ERASED.
- **key** Pointer to a caller-initialized wc_MlDsaKey. On success, only the public half is populated (pubKeySet == 1, prvKeySet == 0).
- **label_len** Size of the optional label buffer in bytes.
- **label** Optional buffer to receive the key's label. May be NULL.

Return: int Returns 0 on success, or a negative error code on failure.

DMA counterpart to wh_Client_MlDsaExportPublicKey. The server emits the public-only DER and DMAs it directly into a client-side staging buffer; the wrapper then deserializes it into the caller-provided wc_MlDsaKey.

The NONEXPORTABLE key flag does NOT block this call because public material is non-sensitive. The caller is responsible for initializing key (e.g. wc_MlDsaKey_Init + wc_MlDsaKey_SetParams) prior to calling.

A.2.2.170 function wh_Client_MlDsaMakeExportKeyDma

```
int wh_Client_MlDsaMakeExportKeyDma(
    whClientContext * ctx,
    int level,
    wc_MlDsaKey * key
)
```

Generate a new ML-DSA key pair and export it using DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **level** The ML-DSA security level.
- **key** Pointer to the ML-DSA key structure to store the key.

Return: int Returns 0 on success, or a negative error code on failure.

This function generates a new ML-DSA key pair in the HSM and exports it using DMA.

A.2.2.171 function wh_Client_MlDsaSignDma

```
int wh_Client_MlDsaSignDma(
    whClientContext * ctx,
    const byte * in,
    word32 in_len,
    byte * out,
    word32 * out_len,
    wc_MlDsaKey * key,
```

```
    const byte * context,
    byte contextLen,
    word32 preHashType
)
```

Sign a message using ML-DSA with DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **in** Pointer to the message to sign.
- **in_len** Length of the message in bytes.
- **out** Pointer to store the signature.
- **out_len** On input, size of out buffer. On output, length of signature.
- **key** Pointer to the ML-DSA key structure.
- **context** Optional FIPS 204 context string for domain separation, or NULL for no context.
- **contextLen** Length of the context string (max 255).
- **preHashType** Hash type for HashML-DSA (e.g. WC_HASH_TYPE_SHA256), or WC_HASH_TYPE_NONE for pure ML-DSA.

Return: int Returns 0 on success, or a negative error code on failure.

This function signs a message using ML-DSA with DMA.

A.2.2.172 function wh_Client_MlDsaVerifyDma

```
int wh_Client_MlDsaVerifyDma(
    whClientContext * ctx,
    const byte * sig,
    word32 sig_len,
    const byte * msg,
    word32 msg_len,
    int * res,
    wc_MlDsaKey * key,
    const byte * context,
    byte contextLen,
    word32 preHashType
)
```

Verify a ML-DSA signature with DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **sig** Pointer to the signature to verify.
- **sig_len** Length of the signature in bytes.
- **msg** Pointer to the message that was signed.
- **msg_len** Length of the message in bytes.
- **res** Result of verification (1 = success, 0 = failure).
- **key** Pointer to the ML-DSA key structure.
- **context** Optional FIPS 204 context string for domain separation, or NULL for no context.
- **contextLen** Length of the context string (max 255).
- **preHashType** Hash type for HashML-DSA (e.g. WC_HASH_TYPE_SHA256), or WC_HASH_TYPE_NONE for pure ML-DSA.

Return: int Returns 0 on success, or a negative error code on failure.

This function verifies a ML-DSA signature with DMA.

A.2.2.173 function wh_Client_MlDsaCheckPrivKeyDma

```
int wh_Client_MlDsaCheckPrivKeyDma(
    whClientContext * ctx,
    wc_MlDsaKey * key,
    const byte * pubKey,
    word32 pubKeySz
)
```

Check a ML-DSA private key against public key with DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **key** Pointer to the ML-DSA private key structure.
- **pubKey** Pointer to the public key to check against.
- **pubKeySz** Size of the public key in bytes.

Return: int Returns 0 on success, or a negative error code on failure.

This function checks if a ML-DSA private key matches a public key with DMA.

A.2.2.174 function wh_Client_MlKemSetKeyId

```
int wh_Client_MlKemSetKeyId(
    MlKemKey * key,
    whKeyId keyId
)
```

Associate a ML-KEM key with a specific key ID.

Parameters:

- **key** Pointer to the ML-KEM key structure.
- **keyId** Key ID to be associated with the ML-KEM key.

Return: int Returns 0 on success or a negative error code on failure.

Sets the device context of a ML-KEM key to the specified key ID. On the server side, this key ID is used to reference the key stored in the HSM.

A.2.2.175 function wh_Client_MlKemGetKeyId

```
int wh_Client_MlKemGetKeyId(
    MlKemKey * key,
    whKeyId * outId
)
```

Retrieve the key ID associated with a ML-KEM key.

Parameters:

- **key** Pointer to the ML-KEM key structure.
- **outId** Pointer to store the retrieved key ID.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.176 function wh_Client_MlKemImportKey

```
int wh_Client_MlKemImportKey(
    whClientContext * ctx,
    MlKemKey * key,
```

```
    whKeyId * inout_keyId,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Import a ML-KEM key to the server key cache.

Parameters:

- **ctx** Pointer to the client context.
- **key** Pointer to the ML-KEM key to import.
- **inout_keyId** Pointer to key ID to use/receive.
- **flags** Flags to control key persistence.
- **label_len** Length of optional label in bytes.
- **label** Optional label to associate with the key.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.177 function wh_Client_MlKemExportKey

```
int wh_Client_MlKemExportKey(
    whClientContext * ctx,
    whKeyId keyId,
    MlKemKey * key,
    uint16_t label_len,
    uint8_t * label
)
```

Export a ML-KEM key from the server key cache.

Parameters:

- **ctx** Pointer to the client context.
- **keyId** Key ID of the key to export.
- **key** Pointer to the ML-KEM key structure to populate.
- **label_len** Length of optional label in bytes.
- **label** Optional label buffer to receive the key label.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.178 function wh_Client_MlKemExportPublicKey

```
int wh_Client_MlKemExportPublicKey(
    whClientContext * ctx,
    whKeyId keyId,
    MlKemKey * key,
    uint16_t label_len,
    uint8_t * label
)
```

Exports only the public part of a cached ML-KEM key.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **keyId** Server key ID whose public key should be exported. Must not be WH_KEYID_ERASED.
- **key** Pointer to a caller-initialized MlKemKey. On success, only the public half is populated.
- **label_len** Size of the optional label buffer in bytes. Values larger than WH_NVM_LABEL_LEN are truncated. Set to 0 if label is not needed.

- **label** Optional buffer to receive the key's label. May be NULL.

Return: int Returns 0 on success or a negative wolfHSM/wolfCrypt error code on failure (e.g. WH_ERROR_NOTFOUND, WH_ERROR_BADARGS, WH_ERROR_NOSPACE).

Instructs the server to emit only the public portion of a cached ML-KEM key as raw FIPS 203 wire-format bytes (not DER). The private key stays inside the HSM. The decoded key will have MLKEM_FLAG_PUB_SET set and MLKEM_FLAG_PRIV_SET clear.

The NONEXPORTABLE key flag does NOT block this call because public material is non-sensitive. The caller is responsible for initializing key (e.g. wc_MlKemKey_Init).

A.2.2.179 function wh_Client_MlKemMakeExportKey

```
int wh_Client_MlKemMakeExportKey(
    whClientContext * ctx,
    int level,
    MlKemKey * key
)
```

Generate a ML-KEM key pair and return it as an ephemeral key.

Parameters:

- **ctx** Pointer to the client context.
- **level** ML-KEM security level (WC_ML_KEM_512/768/1024).
- **key** Pointer to the ML-KEM key to populate with the generated key.

Return: int Returns 0 on success or a negative error code on failure.

The key pair is generated on the server, serialized, and returned to the client without being cached.

A.2.2.180 function wh_Client_MlKemMakeCacheKey

```
int wh_Client_MlKemMakeCacheKey(
    whClientContext * ctx,
    int level,
    whKeyId * inout_key_id,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Generate a ML-KEM key pair and cache it on the server.

Parameters:

- **ctx** Pointer to the client context.
- **level** ML-KEM security level (WC_ML_KEM_512/768/1024).
- **inout_key_id** Pointer to key ID to use/receive.
- **flags** Flags to control key persistence and usage.
- **label_len** Length of optional label in bytes.
- **label** Optional label to associate with the key.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.181 function wh_Client_MlKemEncapsulate

```
int wh_Client_MlKemEncapsulate(
    whClientContext * ctx,
```

```
    MlKemKey * key,
    uint8_t * ct,
    uint32_t * inout_ct_len,
    uint8_t * ss,
    uint32_t * inout_ss_len
)
```

Perform ML-KEM encapsulation using a server-cached public key.

Parameters:

- **ctx** Pointer to the client context.
- **key** Pointer to the ML-KEM key (must have key ID set).
- **ct** Buffer to receive the ciphertext.
- **inout_ct_len** On input, size of ct buffer; on output, actual ciphertext length.
- **ss** Buffer to receive the shared secret.
- **inout_ss_len** On input, size of ss buffer; on output, actual shared secret length.

Return: int Returns 0 on success or a negative error code on failure.

Generates a shared secret and ciphertext using the public key identified by the key ID stored in the provided MlKemKey. If the key is not yet cached, it will be auto-imported and evicted after use.

A.2.2.182 function wh_Client_MlKemDecapsulate

```
int wh_Client_MlKemDecapsulate(
    whClientContext * ctx,
    MlKemKey * key,
    const uint8_t * ct,
    uint32_t ct_len,
    uint8_t * ss,
    uint32_t * inout_ss_len
)
```

Perform ML-KEM decapsulation using a server-cached private key.

Parameters:

- **ctx** Pointer to the client context.
- **key** Pointer to the ML-KEM key (must have key ID set).
- **ct** Pointer to the ciphertext.
- **ct_len** Length of the ciphertext in bytes.
- **ss** Buffer to receive the shared secret.
- **inout_ss_len** On input, size of ss buffer; on output, actual shared secret length.

Return: int Returns 0 on success or a negative error code on failure.

Recovers the shared secret from the ciphertext using the private key identified by the key ID stored in the provided MlKemKey. If the key is not yet cached, it will be auto-imported and evicted after use.

A.2.2.183 function wh_Client_MlKemImportKeyDma

```
int wh_Client_MlKemImportKeyDma(
    whClientContext * ctx,
    MlKemKey * key,
    whKeyId * inout_keyId,
    whNvmFlags flags,
    uint16_t label_len,
```

```
    uint8_t * label
)
```

Import a ML-KEM key using DMA.

Parameters:

- **ctx** Pointer to the client context.
- **key** Pointer to the ML-KEM key to import.
- **inout_keyId** Pointer to store/provide the key ID.
- **flags** NVM flags for key storage.
- **label_len** Length of the key label in bytes.
- **label** Pointer to the key label.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.184 function wh_Client_MlKemExportKeyDma

```
int wh_Client_MlKemExportKeyDma(
    whClientContext * ctx,
    whKeyId keyId,
    MlKemKey * key,
    uint16_t label_len,
    uint8_t * label
)
```

Export a ML-KEM key from the server using DMA.

Parameters:

- **ctx** Pointer to the client context.
- **keyId** Key ID of the key to export.
- **key** Pointer to the ML-KEM key structure to populate.
- **label_len** Length of the key label in bytes.
- **label** Pointer to the key label buffer.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.185 function wh_Client_MlKemExportPublicKeyDma

```
int wh_Client_MlKemExportPublicKeyDma(
    whClientContext * ctx,
    whKeyId keyId,
    MlKemKey * key,
    uint16_t label_len,
    uint8_t * label
)
```

Exports only the public part of a cached ML-KEM key using DMA.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **keyId** Server key ID whose public key should be exported. Must not be WH_KEYID_ERASED.
- **key** Pointer to a caller-initialized MlKemKey. On success, only the public half is populated.
- **label_len** Size of the optional label buffer in bytes.
- **label** Optional buffer to receive the key's label. May be NULL.

Return: int Returns 0 on success or a negative wolfHSM/wolfCrypt error code on failure.

DMA counterpart to wh_Client_MlKemExportPublicKey. The server writes the raw FIPS 203 wire-format public bytes directly into the client-provided buffer.

The NONEXPORTABLE key flag does NOT block this call because public material is non-sensitive.

A.2.2.186 function wh_Client_MlKemMakeExportKeyDma

```
int wh_Client_MlKemMakeExportKeyDma(
    whClientContext * ctx,
    int level,
    MlKemKey * key
)
```

Generate an ephemeral ML-KEM key pair using DMA.

Parameters:

- **ctx** Pointer to the client context.
- **level** ML-KEM security level (WC_ML_KEM_512/768/1024).
- **key** Pointer to the ML-KEM key to populate.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.187 function wh_Client_MlKemEncapsulateDma

```
int wh_Client_MlKemEncapsulateDma(
    whClientContext * ctx,
    MlKemKey * key,
    uint8_t * ct,
    uint32_t * inout_ct_len,
    uint8_t * ss,
    uint32_t * inout_ss_len
)
```

Perform ML-KEM encapsulation using DMA.

Parameters:

- **ctx** Pointer to the client context.
- **key** Pointer to the ML-KEM key (must have key ID set).
- **ct** Buffer to receive the ciphertext.
- **inout_ct_len** On input, size of ct buffer; on output, actual ciphertext length.
- **ss** Buffer to receive the shared secret.
- **inout_ss_len** On input, size of ss buffer; on output, actual shared secret length.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.188 function wh_Client_MlKemDecapsulateDma

```
int wh_Client_MlKemDecapsulateDma(
    whClientContext * ctx,
    MlKemKey * key,
    const uint8_t * ct,
    uint32_t ct_len,
    uint8_t * ss,
    uint32_t * inout_ss_len
)
```

Perform ML-KEM decapsulation using DMA.

Parameters:

- **ctx** Pointer to the client context.
- **key** Pointer to the ML-KEM key (must have key ID set).
- **ct** Pointer to the ciphertext.
- **ct_len** Length of the ciphertext in bytes.
- **ss** Buffer to receive the shared secret.
- **inout_ss_len** On input, size of ss buffer; on output, actual shared secret length.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.189 function wh_Client_LmsSetKeyId

```
int wh_Client_LmsSetKeyId(
    LmsKey * key,
    whKeyId keyId
)
```

Bind a wolfHSM keyId into an LmsKey's devCtx.

Parameters:

- **key** LmsKey to update.
- **keyId** Server-side keyId to store in key->devCtx.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.190 function wh_Client_LmsGetKeyId

```
int wh_Client_LmsGetKeyId(
    LmsKey * key,
    whKeyId * outId
)
```

Read the wolfHSM keyId stored in an LmsKey's devCtx.

Parameters:

- **key** LmsKey to query.
- **outId** Receives the keyId held in key->devCtx.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.191 function wh_Client_LmsMakeKeyDma

```
int wh_Client_LmsMakeKeyDma(
    whClientContext * ctx,
    LmsKey * key,
    whKeyId * inout_key_id,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Generate an LMS key on the server.

Parameters:

- **ctx** Pointer to the client context.

- **key** LmsKey with its parameter set bound; on success its devCtx carries the keyId.
- **inout_key_id** On entry an optional requested keyId; on success the assigned keyId. May be NULL.
- **flags** NVM flags; WH_NVM_FLAGS_EPHEMERAL is rejected.
- **label_len** Length of label in bytes (0 if none).
- **label** Optional label, or NULL.

Return: int Returns 0 on success or a negative error code on failure.

The key's parameter set (levels/height/winternitz) must be bound on the in-memory key before this call (e.g. via wc_LmsKey_SetParameters). On success the key's devCtx carries the server-side keyId and the public key is returned via DMA. The key is always committed to the keystore before the public key is returned: WH_NVM_FLAGS_EPHEMERAL is rejected with WH_ERROR_BADARGS for stateful keys, since releasing the public key of a non-durable private key would orphan it on power loss.

A.2.2.192 function wh_Client_LmsMakeExportKeyDma

```
int wh_Client_LmsMakeExportKeyDma(
    whClientContext * ctx,
    LmsKey * key
)
```

Convenience wrapper for keygen that returns the public key via DMA.

Parameters:

- **ctx** Pointer to the client context.
- **key** LmsKey with its parameter set bound.

Return: int Returns 0 on success or a negative error code on failure.

Equivalent to wh_Client_LmsMakeKeyDma with a server-assigned keyId. As with that call the key is committed to the keystore before its public key is returned.

A.2.2.193 function wh_Client_LmsSignDma

```
int wh_Client_LmsSignDma(
    whClientContext * ctx,
    const byte * msg,
    word32 msgSz,
    byte * sig,
    word32 * sigSz,
    LmsKey * key
)
```

Sign a message with an HSM-resident LMS key.

Parameters:

- **ctx** Pointer to the client context.
- **msg** Message to sign.
- **msgSz** Length of msg in bytes.
- **sig** Buffer to receive the signature.
- **sigSz** On entry the capacity of sig; on success the signature length.
- **key** LmsKey whose devCtx carries the keyId.

Return: int Returns 0 on success or a negative error code on failure.

The keyId is taken from key->devCtx. The new private state is committed atomically to NVM by the server before the signature is returned.

A.2.2.194 function wh_Client_LmsVerifyDma

```
int wh_Client_LmsVerifyDma(
    whClientContext * ctx,
    const byte * sig,
    word32 sigSz,
    const byte * msg,
    word32 msgSz,
    int * res,
    LmsKey * key
)
```

Verify a signature using an HSM-resident LMS key.

Parameters:

- **ctx** Pointer to the client context.
- **sig** Signature to verify.
- **sigSz** Length of sig in bytes.
- **msg** Message that was signed.
- **msgSz** Length of msg in bytes.
- **res** Set to 1 on a valid signature, 0 on mismatch.
- **key** LmsKey whose devCtx carries the keyId.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.195 function wh_Client_LmsSigsLeftDma

```
int wh_Client_LmsSigsLeftDma(
    whClientContext * ctx,
    LmsKey * key
)
```

Report whether an HSM-resident LMS key can still produce signatures.

Parameters:

- **ctx** Pointer to the client context.
- **key** LmsKey whose devCtx carries the keyId.

Return: int Returns 1 if signatures remain, 0 if the key is exhausted, or a negative error code on failure.

Mirrors wc_LmsKey_SigsLeft(): the result is a boolean, not a count.

A.2.2.196 function wh_Client_LmsImportPubKey

```
int wh_Client_LmsImportPubKey(
    whClientContext * ctx,
    LmsKey * key,
    whKeyId * inout_keyId,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Import a verify-only LMS public key into the keystore.

Parameters:

- **ctx** Pointer to the client context.

- **key** LmsKey with its parameter set bound and public key loaded; on success its devCtx carries the keyId.
- **inout_keyId** On entry a specific keyId to provision, or WH_KEYID_ERASED to be assigned one; on success the keyId. May be NULL.
- **flags** NVM flags; WH_NVM_FLAGS_NONMODIFIABLE pins the key, and it is committed to NVM unless WH_NVM_FLAGS_EPHEMERAL is set.
- **label_len** Length of label in bytes (0 if none).
- **label** Optional label, or NULL.

Return: int Returns 0 on success or a negative error code on failure.

The in-memory key must have its parameter set bound and the public key loaded (e.g. via wc_LmsKey_SetParameters + wc_LmsKey_ImportPubRaw). On success the key's devCtx carries the server-side keyId, usable with wh_Client_LmsVerifyDma. No private state is stored, so the key cannot sign.

A.2.2.197 function wh_Client_XmssSetKeyId

```
int wh_Client_XmssSetKeyId(
    XmssKey * key,
    whKeyId keyId
)
```

Bind a wolfHSM keyId into an XmssKey's devCtx.

Parameters:

- **key** XmssKey to update.
- **keyId** Server-side keyId to store in key->devCtx.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.198 function wh_Client_XmssGetKeyId

```
int wh_Client_XmssGetKeyId(
    XmssKey * key,
    whKeyId * outId
)
```

Read the wolfHSM keyId stored in an XmssKey's devCtx.

Parameters:

- **key** XmssKey to query.
- **outId** Receives the keyId held in key->devCtx.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.199 function wh_Client_XmssMakeKeyDma

```
int wh_Client_XmssMakeKeyDma(
    whClientContext * ctx,
    XmssKey * key,
    whKeyId * inout_key_id,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Generate an XMSS / XMSS^MT key on the server.

Parameters:

- **ctx** Pointer to the client context.
- **key** XmssKey with its parameter string bound; on success its devCtx carries the keyId.
- **inout_key_id** On entry an optional requested keyId; on success the assigned keyId. May be NULL.
- **flags** NVM flags; WH_NVM_FLAGS_EPHEMERAL is rejected.
- **label_len** Length of label in bytes (0 if none).
- **label** Optional label, or NULL.

Return: int Returns 0 on success or a negative error code on failure.

The parameter string must be bound on the in-memory key (via wc_XmssKey_SetParamStr) before this call. On success the key's devCtx carries the server-side keyId and the public key is returned via DMA. The key is always committed to the keystore before the public key is returned: WH_NVM_FLAGS_EPHEMERAL is rejected with WH_ERROR_BADARGS for stateful keys, since releasing the public key of a non-durable private key would orphan it on power loss.

A.2.2.200 function wh_Client_XmssMakeExportKeyDma

```
int wh_Client_XmssMakeExportKeyDma(
    whClientContext * ctx,
    XmssKey * key
)
```

Convenience wrapper for keygen that returns the public key via DMA.

Parameters:

- **ctx** Pointer to the client context.
- **key** XmssKey with its parameter string bound.

Return: int Returns 0 on success or a negative error code on failure.

Equivalent to wh_Client_XmssMakeKeyDma with a server-assigned keyId. As with that call the key is committed to the keystore before its public key is returned.

A.2.2.201 function wh_Client_XmssSignDma

```
int wh_Client_XmssSignDma(
    whClientContext * ctx,
    const byte * msg,
    word32 msgSz,
    byte * sig,
    word32 * sigSz,
    XmssKey * key
)
```

Sign a message with an HSM-resident XMSS key.

Parameters:

- **ctx** Pointer to the client context.
- **msg** Message to sign.
- **msgSz** Length of msg in bytes.
- **sig** Buffer to receive the signature.
- **sigSz** On entry the capacity of sig; on success the signature length.
- **key** XmssKey whose devCtx carries the keyId.

Return: int Returns 0 on success or a negative error code on failure.

The keyId is taken from key->devCtx. The new private state is committed atomically to NVM by the server before the signature is returned.

A.2.2.202 function wh_Client_XmssVerifyDma

```
int wh_Client_XmssVerifyDma(
    whClientContext * ctx,
    const byte * sig,
    word32 sigSz,
    const byte * msg,
    word32 msgSz,
    int * res,
    XmssKey * key
)
```

Verify a signature using an HSM-resident XMSS key.

Parameters:

- **ctx** Pointer to the client context.
- **sig** Signature to verify.
- **sigSz** Length of sig in bytes.
- **msg** Message that was signed.
- **msgSz** Length of msg in bytes.
- **res** Set to 1 on a valid signature, 0 on mismatch.
- **key** XmssKey whose devCtx carries the keyId.

Return: int Returns 0 on success or a negative error code on failure.

A.2.2.203 function wh_Client_XmssSigsLeftDma

```
int wh_Client_XmssSigsLeftDma(
    whClientContext * ctx,
    XmssKey * key
)
```

Report whether an HSM-resident XMSS key can still produce signatures.

Parameters:

- **ctx** Pointer to the client context.
- **key** XmssKey whose devCtx carries the keyId.

Return: int Returns 1 if signatures remain, 0 if the key is exhausted, or a negative error code on failure.

Mirrors wc_XmssKey_SigsLeft(): the result is a boolean, not a count.

A.2.2.204 function wh_Client_XmssImportPubKey

```
int wh_Client_XmssImportPubKey(
    whClientContext * ctx,
    XmssKey * key,
    whKeyId * inout_keyId,
    whNvmFlags flags,
    uint16_t label_len,
    uint8_t * label
)
```

Import a verify-only XMSS / XMSS^MT public key into the keystore.

Parameters:

- **ctx** Pointer to the client context.
- **key** XmssKey with its parameter string bound and public key loaded; on success its devCtx carries the keyId.
- **inout_keyId** On entry a specific keyId to provision, or WH_KEYID_ERASED to be assigned one; on success the keyId. May be NULL.
- **flags** NVM flags (see wh_Client_LmsImportPubKey).
- **label_len** Length of label in bytes (0 if none).
- **label** Optional label, or NULL.

Return: int Returns 0 on success or a negative error code on failure.

The in-memory key must have its parameter string bound and the public key loaded (e.g. via wc_XmssKey_SetParamStr + wc_XmssKey_ImportPubRaw). Semantics match wh_Client_LmsImportPubKey: no private state is stored (verify only) so the key cannot sign, it may be pinned with WH_NVM_FLAGS_NONMODIFIABLE, and it is committed to NVM unless flags include WH_NVM_FLAGS_EPHEMERAL.

A.2.3 Source code

```
/*
 * Copyright (C) 2024 wolfSSL Inc.
 *
 * This file is part of wolfHSM.
 *
 * wolfHSM is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
 * wolfHSM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with wolfHSM.  If not, see <http://www.gnu.org/licenses/>.
 */
/*
 * wolfhsm/wh_client_crypto.h
 */

#ifndef WOLFHSM_WH_CLIENT_CRYPTO_H_
#define WOLFHSM_WH_CLIENT_CRYPTO_H_

/* Pick up compile-time configuration */
#include "wolfhsm/wh_settings.h"

#ifndef WOLFHSM_CFG_NO_CRYPTO

/* System libraries */
#include <stdint.h>
#include <stdbool.h>
```

```c
/* Common WolfHSM types and defines shared with the server */
#include "wolfhsm/wh_common.h"

/* Component includes */
#include "wolfhsm/wh_comm.h"
#include "wolfhsm/wh_client.h"

#include "wolfssl/wolfcrypt/settings.h"
#include "wolfssl/wolfcrypt/types.h"
#include "wolfssl/wolfcrypt/error-crypt.h"
#include "wolfssl/wolfcrypt/wc_port.h"
#include "wolfssl/wolfcrypt/cryptocb.h"
#include "wolfssl/wolfcrypt/aes.h"
#include "wolfssl/wolfcrypt/cmac.h"
#include "wolfssl/wolfcrypt/curve25519.h"
#include "wolfssl/wolfcrypt/rsa.h"
#include "wolfssl/wolfcrypt/ecc.h"
#include "wolfssl/wolfcrypt/ed25519.h"
#include "wolfssl/wolfcrypt/wc_mldsa.h"
#include "wolfssl/wolfcrypt/wc_mlkem.h"
#include "wolfssl/wolfcrypt/hmac.h"

int wh_Client_RngGenerate(whClientContext* ctx, uint8_t* out, uint32_t size);

int wh_Client_RngGenerateRequest(whClientContext* ctx, uint32_t size);

int wh_Client_RngGenerateResponse(whClientContext* ctx, uint8_t* out,
                                  uint32_t* inout_size);

#ifdef WOLFHSM_CFG_DMA
int wh_Client_RngGenerateDma(whClientContext* ctx, uint8_t* out, uint32_t
 ↪  size);

int wh_Client_RngGenerateDmaRequest(whClientContext* ctx, uint8_t* out,
                                    uint32_t size);

int wh_Client_RngGenerateDmaResponse(whClientContext* ctx);
#endif /* WOLFHSM_CFG_DMA */

#ifdef HAVE_CURVE25519
int wh_Client_Curve25519SetKeyId(curve25519_key* key, whKeyId keyId);

int wh_Client_Curve25519GetKeyId(curve25519_key* key, whKeyId* outId);

int wh_Client_Curve25519ImportKey(whClientContext* ctx, curve25519_key* key,
        whKeyId *inout_keyId, whNvmFlags flags,
        uint16_t label_len, uint8_t* label);

int wh_Client_Curve25519ExportKey(whClientContext* ctx, whKeyId keyId,
        curve25519_key* key, uint16_t label_len, uint8_t* label);

int wh_Client_Curve25519ExportPublicKey(whClientContext* ctx, whKeyId keyId,
        curve25519_key* key, uint16_t label_len, uint8_t* label);
```

```
int wh_Client_Curve25519MakeCacheKey(whClientContext* ctx,
        uint16_t size,
        whKeyId *inout_key_id, whNvmFlags flags,
        const uint8_t* label, uint16_t label_len);

int wh_Client_Curve25519MakeExportKey(whClientContext* ctx,
        uint16_t size, curve25519_key* key);

int wh_Client_Curve25519SharedSecret(whClientContext* ctx,
        curve25519_key* priv_key, curve25519_key* pub_key,
        int endian, uint8_t* out, uint16_t *out_size);

int wh_Client_Curve25519SharedSecretRequest(whClientContext* ctx,
                                            whKeyId          prv_key_id,
                                            whKeyId pub_key_id, int endian);

int wh_Client_Curve25519SharedSecretResponse(whClientContext* ctx, uint8_t*
 ↪  out,
                                             uint16_t* out_size);

int wh_Client_Curve25519SharedSecretCacheKey(
    whClientContext* ctx, curve25519_key* priv_key, curve25519_key* pub_key,
    int endian, whKeyId* inout_key_id, whNvmFlags flags, const uint8_t* label,
    uint16_t label_len);

int wh_Client_Curve25519SharedSecretCacheKeyRequest(
    whClientContext* ctx, whKeyId prv_key_id, whKeyId pub_key_id, int endian,
    whKeyId out_key_id, whNvmFlags flags, const uint8_t* label,
    uint16_t label_len);

int wh_Client_Curve25519SharedSecretCacheKeyResponse(whClientContext* ctx,
                                                     whKeyId* out_key_id);

#endif /* HAVE_CURVE25519 */

#ifdef HAVE_ECC
int wh_Client_EccSetKeyId(ecc_key* key, whKeyId keyId);

int wh_Client_EccGetKeyId(ecc_key* key, whKeyId* outId);

int wh_Client_EccImportKey(whClientContext* ctx, ecc_key* key,
        whKeyId *inout_keyId, whNvmFlags flags,
        uint16_t label_len, uint8_t* label);

int wh_Client_EccExportKey(whClientContext* ctx, whKeyId keyId,
        ecc_key* key,
        uint16_t label_len, uint8_t* label);

 int wh_Client_EccExportPublicKey(whClientContext* ctx, whKeyId keyId,
        ecc_key* key, uint16_t label_len, uint8_t* label);

int wh_Client_EccMakeExportKey(whClientContext* ctx,
        int size, int curveId, ecc_key* key);
```

```
int wh_Client_EccMakeCacheKey(whClientContext* ctx,
        int size, int curveId,
        whKeyId *inout_key_id, whNvmFlags flags,
        uint16_t label_len, uint8_t* label);

int wh_Client_EccSharedSecret(whClientContext* ctx, ecc_key* priv_key,
                              ecc_key* pub_key, uint8_t* out,
                              uint16_t* inout_size);

int wh_Client_EccSign(whClientContext* ctx,
        ecc_key* key,
        const uint8_t* hash, uint16_t hash_len,
        uint8_t* sig, uint16_t *inout_sig_len);

int wh_Client_EccVerify(whClientContext* ctx, ecc_key* key,
        const uint8_t* sig, uint16_t sig_len,
        const uint8_t* hash, uint16_t hash_len,
        int *out_res);

int wh_Client_EccSignRequest(whClientContext* ctx, whKeyId keyId,
                             const uint8_t* hash, uint16_t hash_len);

int wh_Client_EccSignResponse(whClientContext* ctx, uint8_t* sig,
                              uint16_t* inout_sig_len);

int wh_Client_EccVerifyRequest(whClientContext* ctx, whKeyId keyId,
                               const uint8_t* sig, uint16_t sig_len,
                               const uint8_t* hash, uint16_t hash_len);

int wh_Client_EccVerifyResponse(whClientContext* ctx, ecc_key* opt_key,
                                int* out_res);

int wh_Client_EccSharedSecretRequest(whClientContext* ctx, whKeyId prv_key_id,
                                     whKeyId pub_key_id);

int wh_Client_EccSharedSecretResponse(whClientContext* ctx, uint8_t* out,
                                      uint16_t* inout_size);

int wh_Client_EccSharedSecretCacheKey(whClientContext* ctx, ecc_key* priv_key,
                                      ecc_key* pub_key, whKeyId* inout_key_id,
                                      whNvmFlags flags, const uint8_t* label,
                                      uint16_t label_len);

int wh_Client_EccSharedSecretCacheKeyRequest(
    whClientContext* ctx, whKeyId prv_key_id, whKeyId pub_key_id,
    whKeyId out_key_id, whNvmFlags flags, const uint8_t* label,
    uint16_t label_len);

int wh_Client_EccSharedSecretCacheKeyResponse(whClientContext* ctx,
                                              whKeyId*         out_key_id);

int wh_Client_EccMakeCacheKeyRequest(whClientContext* ctx, int size,
                                     int curveId, whKeyId key_id,
                                     whNvmFlags flags, uint16_t label_len,
```

```
                                    uint8_t* label);

int wh_Client_EccMakeCacheKeyResponse(whClientContext* ctx,
                                      whKeyId*         out_key_id);

int wh_Client_EccMakeExportKeyRequest(whClientContext* ctx, int size,
                                      int curveId);

int wh_Client_EccMakeExportKeyResponse(whClientContext* ctx, ecc_key* key);

#endif /* HAVE_ECC */

#ifdef HAVE_ED25519
int wh_Client_Ed25519SetKeyId(ed25519_key* key, whKeyId keyId);

int wh_Client_Ed25519GetKeyId(ed25519_key* key, whKeyId* outId);

int wh_Client_Ed25519ImportKey(whClientContext* ctx, ed25519_key* key,
                               whKeyId* inout_keyId, whNvmFlags flags,
                               uint16_t label_len, uint8_t* label);

int wh_Client_Ed25519ExportKey(whClientContext* ctx, whKeyId keyId,
                               ed25519_key* key, uint16_t label_len,
                               uint8_t* label);

int wh_Client_Ed25519ExportPublicKey(whClientContext* ctx, whKeyId keyId,
                                     ed25519_key* key, uint16_t label_len,
                                     uint8_t* label);

int wh_Client_Ed25519MakeExportKey(whClientContext* ctx, ed25519_key* key);

int wh_Client_Ed25519MakeCacheKey(whClientContext* ctx, whKeyId* inout_key_id,
                                  whNvmFlags flags, uint16_t label_len,
                                  uint8_t* label);

int wh_Client_Ed25519Sign(whClientContext* ctx, ed25519_key* key,
                          const uint8_t* msg, uint32_t msgLen, uint8_t type,
                          const uint8_t* context, uint32_t contextLen,
                          uint8_t* sig, uint32_t* inout_sig_len);

int wh_Client_Ed25519Verify(whClientContext* ctx, ed25519_key* key,
                            const uint8_t* sig, uint32_t sigLen,
                            const uint8_t* msg, uint32_t msgLen, uint8_t type,
                            const uint8_t* context, uint32_t contextLen,
                            int* out_res);

#ifdef WOLFHSM_CFG_DMA
int wh_Client_Ed25519SignDma(whClientContext* ctx, ed25519_key* key,
                             const uint8_t* msg, uint32_t msgLen, uint8_t type,
                             const uint8_t* context, uint32_t contextLen,
                             uint8_t* sig, uint32_t* inout_sig_len);

int wh_Client_Ed25519VerifyDma(whClientContext* ctx, ed25519_key* key,
                               const uint8_t* sig, uint32_t sigLen,
```

```
                                const uint8_t* msg, uint32_t msgLen,
                                uint8_t type, const uint8_t* context,
                                uint32_t contextLen, int* out_res);
#endif /* WOLFHSM_CFG_DMA */
#endif /* HAVE_ED25519 */

#ifndef NO_RSA
int wh_Client_RsaSetKeyId(RsaKey* key, whNvmId keyId);

int wh_Client_RsaGetKeyId(RsaKey* key, whNvmId* outId);

int wh_Client_RsaImportKey(whClientContext* ctx, const RsaKey* key,
        whKeyId *inout_keyId, whNvmFlags flags,
        uint32_t label_len, uint8_t* label);

int wh_Client_RsaExportKey(whClientContext* ctx, whKeyId keyId,
        RsaKey* key, uint32_t label_len, uint8_t* label);

int wh_Client_RsaExportPublicKey(whClientContext* ctx, whKeyId keyId,
        RsaKey* key, uint32_t label_len, uint8_t* label);

/* Generate an RSA key on the server and export it inta an RSA struct */
int wh_Client_RsaMakeExportKey(whClientContext* ctx,
        uint32_t size, uint32_t e, RsaKey* rsa);

/* Generate an RSA key on the server and put it in the server keycache */
int wh_Client_RsaMakeCacheKey(whClientContext* ctx,
        uint32_t size, uint32_t e,
        whKeyId* inout_key_id, whNvmFlags flags,
        uint32_t label_len, uint8_t* label);

/* TODO: Request server to perform the RSA function */
int wh_Client_RsaFunction(whClientContext* ctx,
        RsaKey* key, int rsa_type,
        const uint8_t* in, uint16_t in_len,
        uint8_t* out, uint16_t *inout_out_len);

/* TODO: Request server to get the RSA size */
int wh_Client_RsaGetSize(whClientContext* ctx,
        const RsaKey* key, int* out_size);

int wh_Client_RsaFunctionRequest(whClientContext* ctx, whKeyId keyId,
                                 int rsa_type, const uint8_t* in,
                                 uint16_t in_len, uint16_t out_capacity);

int wh_Client_RsaFunctionResponse(whClientContext* ctx, uint8_t* out,
                                  uint16_t* inout_out_len);

int wh_Client_RsaGetSizeRequest(whClientContext* ctx, whKeyId keyId);

int wh_Client_RsaGetSizeResponse(whClientContext* ctx, int* out_size);

int wh_Client_RsaMakeCacheKeyRequest(whClientContext* ctx, uint32_t size,
                                     uint32_t e, whKeyId key_id,
```

```
                                    whNvmFlags flags, uint32_t label_len,
                                    uint8_t* label);

int wh_Client_RsaMakeCacheKeyResponse(whClientContext* ctx,
                                      whKeyId*         out_key_id);

int wh_Client_RsaMakeExportKeyRequest(whClientContext* ctx, uint32_t size,
                                      uint32_t e);

int wh_Client_RsaMakeExportKeyResponse(whClientContext* ctx, RsaKey* rsa);


#endif /* !NO_RSA */

#ifdef HAVE_HKDF
int wh_Client_HkdfMakeCacheKey(whClientContext* ctx, int hashType,
                               whKeyId keyIdIn, const uint8_t* inKey,
                               uint32_t inKeySz, const uint8_t* salt,
                               uint32_t saltSz, const uint8_t* info,
                               uint32_t infoSz, whKeyId* inout_key_id,
                               whNvmFlags flags, const uint8_t* label,
                               uint32_t label_len, uint32_t outSz);

int wh_Client_HkdfMakeExportKey(whClientContext* ctx, int hashType,
                                whKeyId keyIdIn, const uint8_t* inKey,
                                uint32_t inKeySz, const uint8_t* salt,
                                uint32_t saltSz, const uint8_t* info,
                               uint32_t infoSz, uint8_t* out, uint32_t outSz);

#endif /* HAVE_HKDF */

#ifdef HAVE_CMAC_KDF
int wh_Client_CmacKdfMakeCacheKey(whClientContext* ctx, whKeyId saltKeyId,
                                  const uint8_t* salt, uint32_t saltSz,
                                  whKeyId zKeyId, const uint8_t* z,
                                  uint32_t zSz, const uint8_t* fixedInfo,
                                  uint32_t fixedInfoSz, whKeyId* inout_key_id,
                                  whNvmFlags flags, const uint8_t* label,
                                  uint32_t label_len, uint32_t outSz);

int wh_Client_CmacKdfMakeExportKey(whClientContext* ctx, whKeyId saltKeyId,
                                   const uint8_t* salt, uint32_t saltSz,
                                   whKeyId zKeyId, const uint8_t* z,
                                   uint32_t zSz, const uint8_t* fixedInfo,
                                   uint32_t fixedInfoSz, uint8_t* out,
                                   uint32_t outSz);
#endif /* HAVE_CMAC_KDF */

#ifndef NO_AES
int wh_Client_AesSetKeyId(Aes* key, whNvmId keyId);

int wh_Client_AesGetKeyId(Aes* key, whNvmId* outId);

#ifdef WOLFSSL_AES_COUNTER
```

```
int wh_Client_AesCtr(whClientContext* ctx, Aes* aes, int enc, const uint8_t*
 ↪  in,
                     uint32_t len, uint8_t* out);

int wh_Client_AesCtrDma(whClientContext* ctx, Aes* aes, int enc,
                        const uint8_t* in, uint32_t len, uint8_t* out);

int wh_Client_AesCtrRequest(whClientContext* ctx, Aes* aes, int enc,
                            const uint8_t* in, uint32_t len);

int wh_Client_AesCtrResponse(whClientContext* ctx, Aes* aes, uint8_t* out,
                             uint32_t* out_size);

int wh_Client_AesCtrDmaRequest(whClientContext* ctx, Aes* aes, int enc,
                               const uint8_t* in, uint32_t len, uint8_t* out);

int wh_Client_AesCtrDmaResponse(whClientContext* ctx, Aes* aes);
#endif /* WOLFSSL_AES_COUNTER */

#ifdef HAVE_AES_ECB
int wh_Client_AesEcb(whClientContext* ctx, Aes* aes, int enc, const uint8_t*
 ↪  in,
                     uint32_t len, uint8_t* out);

int wh_Client_AesEcbDma(whClientContext* ctx, Aes* aes, int enc,
                        const uint8_t* in, uint32_t len, uint8_t* out);

int wh_Client_AesEcbRequest(whClientContext* ctx, Aes* aes, int enc,
                            const uint8_t* in, uint32_t len);

int wh_Client_AesEcbResponse(whClientContext* ctx, Aes* aes, uint8_t* out,
                             uint32_t* out_size);

int wh_Client_AesEcbDmaRequest(whClientContext* ctx, Aes* aes, int enc,
                               const uint8_t* in, uint32_t len, uint8_t* out);

int wh_Client_AesEcbDmaResponse(whClientContext* ctx, Aes* aes);
#endif /* HAVE_AES_ECB */

#ifdef HAVE_AES_CBC
int wh_Client_AesCbc(whClientContext* ctx,
        Aes* aes, int enc,
        const uint8_t* in, uint32_t len,
        uint8_t* out);

int wh_Client_AesCbcDma(whClientContext* ctx, Aes* aes, int enc,
                        const uint8_t* in, uint32_t len, uint8_t* out);

int wh_Client_AesCbcRequest(whClientContext* ctx, Aes* aes, int enc,
                            const uint8_t* in, uint32_t len);

int wh_Client_AesCbcResponse(whClientContext* ctx, Aes* aes, uint8_t* out,
                             uint32_t* out_size);
```

```
int wh_Client_AesCbcDmaRequest(whClientContext* ctx, Aes* aes, int enc,
                               const uint8_t* in, uint32_t len, uint8_t* out);

int wh_Client_AesCbcDmaResponse(whClientContext* ctx, Aes* aes);
#endif /* HAVE_AES_CBC */

#ifdef HAVE_AESGCM
int wh_Client_AesGcm(whClientContext* ctx,
        Aes* aes, int enc,
        const uint8_t* in, uint32_t len,
        const uint8_t* iv, uint32_t iv_len,
        const uint8_t* authin, uint32_t authin_len,
        const uint8_t* dec_tag, uint8_t* enc_tag, uint32_t tag_len,
        uint8_t* out);

int wh_Client_AesGcmDma(whClientContext* ctx, Aes* aes, int enc,
                        const uint8_t* in, uint32_t len, const uint8_t* iv,
                        uint32_t iv_len, const uint8_t* authin,
                        uint32_t authin_len, const uint8_t* dec_tag,
                        uint8_t* enc_tag, uint32_t tag_len, uint8_t* out);

int wh_Client_AesGcmRequest(whClientContext* ctx, Aes* aes, int enc,
                            const uint8_t* in, uint32_t len, const uint8_t* iv,
                            uint32_t iv_len, const uint8_t* authin,
                            uint32_t authin_len, const uint8_t* dec_tag,
                            uint32_t tag_len);

int wh_Client_AesGcmResponse(whClientContext* ctx, Aes* aes, uint8_t* out,
                             uint32_t out_capacity, uint32_t* out_size,
                             uint8_t* enc_tag, uint32_t tag_len);

int wh_Client_AesGcmDmaRequest(whClientContext* ctx, Aes* aes, int enc,
                               const uint8_t* in, uint32_t len, uint8_t* out,
                               const uint8_t* iv, uint32_t iv_len,
                               const uint8_t* authin, uint32_t authin_len,
                               const uint8_t* dec_tag, uint32_t tag_len);

int wh_Client_AesGcmDmaResponse(whClientContext* ctx, Aes* aes,
                                uint8_t* enc_tag, uint32_t tag_len);
#endif /* HAVE_AESGCM */

#endif /* !NO_AES */


#ifdef WOLFSSL_CMAC

int wh_Client_Cmac(whClientContext* ctx, Cmac* cmac, CmacType type,
                   const uint8_t* key, uint32_t keyLen, const uint8_t* in,
                   uint32_t inLen, uint8_t* outMac, uint32_t* outMacLen);

int wh_Client_CmacGenerateRequest(whClientContext* ctx, Cmac* cmac,
                                  CmacType type, const uint8_t* key,
                                  uint32_t keyLen, const uint8_t* in,
                                  uint32_t inLen, uint32_t outMacLen);
```

```
int wh_Client_CmacGenerateResponse(whClientContext* ctx, Cmac* cmac,
                                   uint8_t* outMac, uint32_t* outMacLen);

int wh_Client_CmacUpdateRequest(whClientContext* ctx, Cmac* cmac, CmacType
 ↪  type,
                                const uint8_t* key, uint32_t keyLen,
                                const uint8_t* in, uint32_t inLen,
                                bool* requestSent);

int wh_Client_CmacUpdateResponse(whClientContext* ctx, Cmac* cmac);

int wh_Client_CmacFinalRequest(whClientContext* ctx, Cmac* cmac);

int wh_Client_CmacFinalResponse(whClientContext* ctx, Cmac* cmac,
                                uint8_t* outMac, uint32_t* outMacLen);


int wh_Client_CmacSetKeyId(Cmac* key, whNvmId keyId);

int wh_Client_CmacGetKeyId(Cmac* key, whNvmId* outId);

#ifdef WOLFHSM_CFG_DMA
int wh_Client_CmacDma(whClientContext* ctx, Cmac* cmac, CmacType type,
                      const uint8_t* key, uint32_t keyLen, const uint8_t* in,
                      uint32_t inLen, uint8_t* outMac, uint32_t* outMacLen);

int wh_Client_CmacGenerateDmaRequest(whClientContext* ctx, Cmac* cmac,
                                     CmacType type, const uint8_t* key,
                                     uint32_t keyLen, const uint8_t* in,
                                     uint32_t inLen, uint32_t outMacLen);

int wh_Client_CmacGenerateDmaResponse(whClientContext* ctx, Cmac* cmac,
                                      uint8_t* outMac, uint32_t* outMacLen);

int wh_Client_CmacDmaUpdateRequest(whClientContext* ctx, Cmac* cmac,
                                   CmacType type, const uint8_t* key,
                                   uint32_t keyLen, const uint8_t* in,
                                   uint32_t inLen, bool* requestSent);

int wh_Client_CmacDmaUpdateResponse(whClientContext* ctx, Cmac* cmac);

int wh_Client_CmacDmaFinalRequest(whClientContext* ctx, Cmac* cmac);

int wh_Client_CmacDmaFinalResponse(whClientContext* ctx, Cmac* cmac,
                                   uint8_t* outMac, uint32_t* outMacLen);
#endif /* WOLFHSM_CFG_DMA */

#endif /* WOLFSSL_CMAC */

#ifndef NO_SHA256

int wh_Client_Sha256(whClientContext* ctx, wc_Sha256* sha, const uint8_t* in,
                     uint32_t inLen, uint8_t* out);
```

```
int wh_Client_Sha256UpdateRequest(whClientContext* ctx, wc_Sha256* sha,
                                  const uint8_t* in, uint32_t inLen,
                                  bool* requestSent);

int wh_Client_Sha256UpdateResponse(whClientContext* ctx, wc_Sha256* sha);

int wh_Client_Sha256FinalRequest(whClientContext* ctx, wc_Sha256* sha);

int wh_Client_Sha256FinalResponse(whClientContext* ctx, wc_Sha256* sha,
                                  uint8_t* out);

int wh_Client_Sha256Dma(whClientContext* ctx, wc_Sha256* sha, const uint8_t*
 ↪  in,
                        uint32_t inLen, uint8_t* out);

#ifdef WOLFHSM_CFG_DMA
int wh_Client_Sha256DmaUpdateRequest(whClientContext* ctx, wc_Sha256* sha,
                                     const uint8_t* in, uint32_t inLen,
                                     bool* requestSent);

int wh_Client_Sha256DmaUpdateResponse(whClientContext* ctx, wc_Sha256* sha);

int wh_Client_Sha256DmaFinalRequest(whClientContext* ctx, wc_Sha256* sha);

int wh_Client_Sha256DmaFinalResponse(whClientContext* ctx, wc_Sha256* sha,
                                     uint8_t* out);
#endif /* WOLFHSM_CFG_DMA */

#endif /* !NO_SHA256 */

#if defined(WOLFSSL_SHA224)
int wh_Client_Sha224(whClientContext* ctx, wc_Sha224* sha, const uint8_t* in,
                     uint32_t inLen, uint8_t* out);

int wh_Client_Sha224UpdateRequest(whClientContext* ctx, wc_Sha224* sha,
                                  const uint8_t* in, uint32_t inLen,
                                  bool* requestSent);

int wh_Client_Sha224UpdateResponse(whClientContext* ctx, wc_Sha224* sha);

int wh_Client_Sha224FinalRequest(whClientContext* ctx, wc_Sha224* sha);

int wh_Client_Sha224FinalResponse(whClientContext* ctx, wc_Sha224* sha,
                                  uint8_t* out);

int wh_Client_Sha224Dma(whClientContext* ctx, wc_Sha224* sha, const uint8_t*
 ↪  in,
                        uint32_t inLen, uint8_t* out);

#ifdef WOLFHSM_CFG_DMA
int wh_Client_Sha224DmaUpdateRequest(whClientContext* ctx, wc_Sha224* sha,
                                     const uint8_t* in, uint32_t inLen,
```

```
                                    bool* requestSent);
int wh_Client_Sha224DmaUpdateResponse(whClientContext* ctx, wc_Sha224* sha);
int wh_Client_Sha224DmaFinalRequest(whClientContext* ctx, wc_Sha224* sha);
int wh_Client_Sha224DmaFinalResponse(whClientContext* ctx, wc_Sha224* sha,
                                    uint8_t* out);
#endif /* WOLFHSM_CFG_DMA */

#endif /* WOLFSSL_SHA224 */

#if defined(WOLFSSL_SHA384)
int wh_Client_Sha384(whClientContext* ctx, wc_Sha384* sha, const uint8_t* in,
                     uint32_t inLen, uint8_t* out);

int wh_Client_Sha384UpdateRequest(whClientContext* ctx, wc_Sha384* sha,
                                  const uint8_t* in, uint32_t inLen,
                                  bool* requestSent);

int wh_Client_Sha384UpdateResponse(whClientContext* ctx, wc_Sha384* sha);

int wh_Client_Sha384FinalRequest(whClientContext* ctx, wc_Sha384* sha);

int wh_Client_Sha384FinalResponse(whClientContext* ctx, wc_Sha384* sha,
                                  uint8_t* out);

int wh_Client_Sha384Dma(whClientContext* ctx, wc_Sha384* sha, const uint8_t*
 ↪  in,
                        uint32_t inLen, uint8_t* out);

#ifdef WOLFHSM_CFG_DMA
int wh_Client_Sha384DmaUpdateRequest(whClientContext* ctx, wc_Sha384* sha,
                                    const uint8_t* in, uint32_t inLen,
                                    bool* requestSent);
int wh_Client_Sha384DmaUpdateResponse(whClientContext* ctx, wc_Sha384* sha);
int wh_Client_Sha384DmaFinalRequest(whClientContext* ctx, wc_Sha384* sha);
int wh_Client_Sha384DmaFinalResponse(whClientContext* ctx, wc_Sha384* sha,
                                    uint8_t* out);
#endif /* WOLFHSM_CFG_DMA */

#endif /* WOLFSSL_SHA384 */

#if defined(WOLFSSL_SHA512)
int wh_Client_Sha512(whClientContext* ctx, wc_Sha512* sha, const uint8_t* in,
                     uint32_t inLen, uint8_t* out);

int wh_Client_Sha512UpdateRequest(whClientContext* ctx, wc_Sha512* sha,
                                  const uint8_t* in, uint32_t inLen,
                                  bool* requestSent);

int wh_Client_Sha512UpdateResponse(whClientContext* ctx, wc_Sha512* sha);

int wh_Client_Sha512FinalRequest(whClientContext* ctx, wc_Sha512* sha);

int wh_Client_Sha512FinalResponse(whClientContext* ctx, wc_Sha512* sha,
                                  uint8_t* out);
```

```
int wh_Client_Sha512Dma(whClientContext* ctx, wc_Sha512* sha, const uint8_t*
 ↪  in,
                        uint32_t inLen, uint8_t* out);

#ifdef WOLFHSM_CFG_DMA
int wh_Client_Sha512DmaUpdateRequest(whClientContext* ctx, wc_Sha512* sha,
                                     const uint8_t* in, uint32_t inLen,
                                     bool* requestSent);
int wh_Client_Sha512DmaUpdateResponse(whClientContext* ctx, wc_Sha512* sha);
int wh_Client_Sha512DmaFinalRequest(whClientContext* ctx, wc_Sha512* sha);
int wh_Client_Sha512DmaFinalResponse(whClientContext* ctx, wc_Sha512* sha,
                                     uint8_t* out);
#endif /* WOLFHSM_CFG_DMA */

#endif /* WOLFSSL_SHA512 */

#ifdef WOLFSSL_HAVE_MLDSA

int wh_Client_MlDsaSetKeyId(wc_MlDsaKey* key, whKeyId keyId);

int wh_Client_MlDsaGetKeyId(wc_MlDsaKey* key, whKeyId* outId);

int wh_Client_MlDsaImportKey(whClientContext* ctx, wc_MlDsaKey* key,
                             whKeyId* inout_keyId, whNvmFlags flags,
                             uint16_t label_len, uint8_t* label);

int wh_Client_MlDsaExportKey(whClientContext* ctx, whKeyId keyId, wc_MlDsaKey*
 ↪  key,
                             uint16_t label_len, uint8_t* label);

int wh_Client_MlDsaExportPublicKey(whClientContext* ctx, whKeyId keyId,
                                   wc_MlDsaKey* key, uint16_t label_len,
                                   uint8_t* label);

int wh_Client_MlDsaMakeExportKey(whClientContext* ctx, int level, int size,
                                 wc_MlDsaKey* key);
int wh_Client_MlDsaMakeCacheKey(whClientContext* ctx, int size, int level,
                                whKeyId* inout_key_id, whNvmFlags flags,
                                uint16_t label_len, uint8_t* label);
int wh_Client_MlDsaSign(whClientContext* ctx, const byte* in, word32 in_len,
                            byte* out, word32* out_len, wc_MlDsaKey* key,
                            const byte* context, byte contextLen,
                            word32 preHashType);
int wh_Client_MlDsaVerify(whClientContext* ctx, const byte* sig,
                              word32 sig_len, const byte* msg, word32 msg_len,
                              int* res, wc_MlDsaKey* key, const byte* context,
                              byte contextLen, word32 preHashType);

int wh_Client_MlDsaCheckPrivKey(whClientContext* ctx, wc_MlDsaKey* key,
                                const byte* pubKey, word32 pubKeySz);


#ifdef WOLFHSM_CFG_DMA
```

```
int wh_Client_MlDsaImportKeyDma(whClientContext* ctx, wc_MlDsaKey* key,
                                whKeyId* inout_keyId, whNvmFlags flags,
                                uint16_t label_len, uint8_t* label);

int wh_Client_MlDsaExportKeyDma(whClientContext* ctx, whKeyId keyId,
                                wc_MlDsaKey* key, uint16_t label_len,
                                uint8_t* label);

int wh_Client_MlDsaExportPublicKeyDma(whClientContext* ctx, whKeyId keyId,
                                      wc_MlDsaKey* key, uint16_t label_len,
                                      uint8_t* label);

int wh_Client_MlDsaMakeExportKeyDma(whClientContext* ctx, int level,
                                    wc_MlDsaKey* key);


int wh_Client_MlDsaSignDma(whClientContext* ctx, const byte* in,
                           word32 in_len, byte* out, word32* out_len,
                           wc_MlDsaKey* key, const byte* context,
                           byte contextLen, word32 preHashType);

int wh_Client_MlDsaVerifyDma(whClientContext* ctx, const byte* sig,
                             word32 sig_len, const byte* msg,
                             word32 msg_len, int* res, wc_MlDsaKey* key,
                             const byte* context, byte contextLen,
                             word32 preHashType);

int wh_Client_MlDsaCheckPrivKeyDma(whClientContext* ctx, wc_MlDsaKey* key,
                                   const byte* pubKey, word32 pubKeySz);

#endif /* WOLFHSM_CFG_DMA */

#endif /* WOLFSSL_HAVE_MLDSA */

#ifdef WOLFSSL_HAVE_MLKEM

int wh_Client_MlKemSetKeyId(MlKemKey* key, whKeyId keyId);

int wh_Client_MlKemGetKeyId(MlKemKey* key, whKeyId* outId);

int wh_Client_MlKemImportKey(whClientContext* ctx, MlKemKey* key,
                             whKeyId* inout_keyId, whNvmFlags flags,
                             uint16_t label_len, uint8_t* label);

int wh_Client_MlKemExportKey(whClientContext* ctx, whKeyId keyId, MlKemKey*
 ↪  key,
                             uint16_t label_len, uint8_t* label);

int wh_Client_MlKemExportPublicKey(whClientContext* ctx, whKeyId keyId,
                                   MlKemKey* key, uint16_t label_len,
                                   uint8_t* label);

int wh_Client_MlKemMakeExportKey(whClientContext* ctx, int level,
                                 MlKemKey* key);
```

```
int wh_Client_MlKemMakeCacheKey(whClientContext* ctx, int level,
                                whKeyId* inout_key_id, whNvmFlags flags,
                                uint16_t label_len, uint8_t* label);

int wh_Client_MlKemEncapsulate(whClientContext* ctx, MlKemKey* key,
                               uint8_t* ct, uint32_t* inout_ct_len,
                               uint8_t* ss, uint32_t* inout_ss_len);

int wh_Client_MlKemDecapsulate(whClientContext* ctx, MlKemKey* key,
                               const uint8_t* ct, uint32_t ct_len, uint8_t* ss,
                               uint32_t* inout_ss_len);

#ifdef WOLFHSM_CFG_DMA

int wh_Client_MlKemImportKeyDma(whClientContext* ctx, MlKemKey* key,
                                whKeyId* inout_keyId, whNvmFlags flags,
                                uint16_t label_len, uint8_t* label);

int wh_Client_MlKemExportKeyDma(whClientContext* ctx, whKeyId keyId,
                                MlKemKey* key, uint16_t label_len,
                                uint8_t* label);

int wh_Client_MlKemExportPublicKeyDma(whClientContext* ctx, whKeyId keyId,
                                      MlKemKey* key, uint16_t label_len,
                                      uint8_t* label);

int wh_Client_MlKemMakeExportKeyDma(whClientContext* ctx, int level,
                                    MlKemKey* key);

int wh_Client_MlKemEncapsulateDma(whClientContext* ctx, MlKemKey* key,
                                  uint8_t* ct, uint32_t* inout_ct_len,
                                  uint8_t* ss, uint32_t* inout_ss_len);

int wh_Client_MlKemDecapsulateDma(whClientContext* ctx, MlKemKey* key,
                                  const uint8_t* ct, uint32_t ct_len,
                                  uint8_t* ss, uint32_t* inout_ss_len);
#endif /* WOLFHSM_CFG_DMA */

#endif /* WOLFSSL_HAVE_MLKEM */

#if defined(WOLFSSL_HAVE_LMS) || defined(WOLFSSL_HAVE_XMSS)
#ifdef WOLFHSM_CFG_DMA

/* The raw public key is returned via DMA at keygen time (see the MakeKey
 * functions below). To retrieve it again later from just a keyId, use the
 * generic wh_Client_KeyExportPublic(ctx, keyId, WH_KEY_ALGO_LMS or
 * WH_KEY_ALGO_XMSS, ...). The private state is non-exportable and cannot be
 * read back by any path. */

#ifdef WOLFSSL_HAVE_LMS

int wh_Client_LmsSetKeyId(LmsKey* key, whKeyId keyId);
```

```
int wh_Client_LmsGetKeyId(LmsKey* key, whKeyId* outId);

int wh_Client_LmsMakeKeyDma(whClientContext* ctx, LmsKey* key,
                            whKeyId* inout_key_id, whNvmFlags flags,
                            uint16_t label_len, uint8_t* label);

int wh_Client_LmsMakeExportKeyDma(whClientContext* ctx, LmsKey* key);

int wh_Client_LmsSignDma(whClientContext* ctx, const byte* msg, word32 msgSz,
                         byte* sig, word32* sigSz, LmsKey* key);

int wh_Client_LmsVerifyDma(whClientContext* ctx, const byte* sig, word32 sigSz,
                           const byte* msg, word32 msgSz, int* res,
                           LmsKey* key);

int wh_Client_LmsSigsLeftDma(whClientContext* ctx, LmsKey* key);

int wh_Client_LmsImportPubKey(whClientContext* ctx, LmsKey* key,
                              whKeyId* inout_keyId, whNvmFlags flags,
                              uint16_t label_len, uint8_t* label);

#endif /* WOLFSSL_HAVE_LMS */

#ifdef WOLFSSL_HAVE_XMSS

int wh_Client_XmssSetKeyId(XmssKey* key, whKeyId keyId);

int wh_Client_XmssGetKeyId(XmssKey* key, whKeyId* outId);

int wh_Client_XmssMakeKeyDma(whClientContext* ctx, XmssKey* key,
                             whKeyId* inout_key_id, whNvmFlags flags,
                             uint16_t label_len, uint8_t* label);

int wh_Client_XmssMakeExportKeyDma(whClientContext* ctx, XmssKey* key);

int wh_Client_XmssSignDma(whClientContext* ctx, const byte* msg, word32 msgSz,
                          byte* sig, word32* sigSz, XmssKey* key);

int wh_Client_XmssVerifyDma(whClientContext* ctx, const byte* sig,
                            word32 sigSz, const byte* msg, word32 msgSz,
                            int* res, XmssKey* key);

int wh_Client_XmssSigsLeftDma(whClientContext* ctx, XmssKey* key);

int wh_Client_XmssImportPubKey(whClientContext* ctx, XmssKey* key,
                               whKeyId* inout_keyId, whNvmFlags flags,
                               uint16_t label_len, uint8_t* label);

#endif /* WOLFSSL_HAVE_XMSS */

#endif /* WOLFHSM_CFG_DMA */
#endif /* WOLFSSL_HAVE_LMS || WOLFSSL_HAVE_XMSS */

#endif /* !WOLFHSM_CFG_NO_CRYPTO */
```

```
#endif /* !WOLFHSM_WH_CLIENT_CRYPTO_H_ */
```

A.3 wolfhsm/wh_client_she.h

A.3.1 Functions

	Name
int	**wh_Client_ShePreProgramKey**(whClientContext * c, whNvmId keyId, whNvmFlags flags, uint8_t * key, whNvmSize keySz)Pre-programs a SHE key directly into NVM, bypassing the key update protocol.
int	**wh_Client_SheSetUidRequest**(whClientContext * c, uint8_t * uid, uint32_t uidSz)Sends a request to set the ECU UID (wolfHSM-specific).
int	**wh_Client_SheSetUidResponse**(whClientContext * c)Receives the response to a set UID request.
int	**wh_Client_SheSetUid**(whClientContext * c, uint8_t * uid, uint32_t uidSz)Sets the ECU UID with a blocking call (wolfHSM-specific).
int	**wh_Client_SheSecureBoot**(whClientContext * c, uint8_t * bootloader, uint32_t bootloaderLen)Runs the SHE secure boot measurement with a blocking call (CMD_SECURE_BOOT).
int	**wh_Client_SheGetStatusRequest**(whClientContext * c)Sends a request to read the SHE status register (CMD_GET_STATUS).
int	**wh_Client_SheGetStatusResponse**(whClientContext * c, uint8_t * sreg)Receives the SHE status register response (CMD_GET_STATUS).
int	**wh_Client_SheGetStatus**(whClientContext * c, uint8_t * sreg)Reads the SHE status register with a blocking call (CMD_GET_STATUS).
int	**wh_Client_SheLoadKeyRequest**(whClientContext * c, uint8_t * messageOne, uint8_t * messageTwo, uint8_t * messageThree)Sends an encrypted key update request (CMD_LOAD_KEY).
int	**wh_Client_SheLoadKeyResponse**(whClientContext * c, uint8_t * messageFour, uint8_t * messageFive)Receives the encrypted key update response (CMD_LOAD_KEY).
int	**wh_Client_SheLoadKey**(whClientContext * c, uint8_t * messageOne, uint8_t * messageTwo, uint8_t * messageThree, uint8_t * messageFour, uint8_t * messageFive)Performs an encrypted SHE key update with a blocking call (CMD_LOAD_KEY).
int	**wh_Client_SheLoadPlainKeyRequest**(whClientContext * c, uint8_t * key, uint32_t keySz)Sends a request to load the RAM key in plaintext (CMD_LOAD_PLAIN_KEY).

	Name
int	**wh_Client_SheLoadPlainKeyResponse**(whClientContext * c)Receives the response to a load plain key request (CMD_LOAD_PLAIN_KEY).
int	**wh_Client_SheLoadPlainKey**(whClientContext * c, uint8_t * key, uint32_t keySz)Loads the RAM key in plaintext with a blocking call (CMD_LOAD_PLAIN_KEY).
int	**wh_Client_SheExportRamKeyRequest**(whClientContext * c)Sends a request to export the RAM key (CMD_EXPORT_RAM_KEY).
int	**wh_Client_SheExportRamKeyResponse**(whClientContext * c, uint8_t * messageOne, uint8_t * messageTwo, uint8_t * messageThree, uint8_t * messageFour, uint8_t * messageFive)Receives the exported RAM key as an M1-M5 blob (CMD_EXPORT_RAM_KEY).
int	**wh_Client_SheExportRamKey**(whClientContext * c, uint8_t * messageOne, uint8_t * messageTwo, uint8_t * messageThree, uint8_t * messageFour, uint8_t * messageFive)Exports the RAM key as an M1-M5 blob with a blocking call (CMD_EXPORT_RAM_KEY).
int	**wh_Client_SheInitRndRequest**(whClientContext * c)Sends a request to initialize the SHE PRNG (CMD_INIT_RNG).
int	**wh_Client_SheInitRndResponse**(whClientContext * c)Receives the response to a PRNG initialization request (CMD_INIT_RNG).
int	**wh_Client_SheInitRnd**(whClientContext * c)Initializes the SHE PRNG with a blocking call (CMD_INIT_RNG).
int	**wh_Client_SheRndRequest**(whClientContext * c)Sends a request to draw random bytes from the SHE PRNG (CMD_RND).
int	**wh_Client_SheRndResponse**(whClientContext * c, uint8_t * out, uint32_t * outSz)Receives random bytes from the SHE PRNG (CMD_RND).
int	**wh_Client_SheRnd**(whClientContext * c, uint8_t * out, uint32_t * outSz)Draws random bytes from the SHE PRNG with a blocking call (CMD_RND).
int	**wh_Client_SheExtendSeedRequest**(whClientContext * c, uint8_t * entropy, uint32_t entropySz)Sends a request to reseed the SHE PRNG (CMD_EXTEND_SEED).
int	**wh_Client_SheExtendSeedResponse**(whClientContext * c)Receives the response to a PRNG reseed request (CMD_EXTEND_SEED).
int	**wh_Client_SheExtendSeed**(whClientContext * c, uint8_t * entropy, uint32_t entropySz)Reseeds the SHE PRNG with a blocking call (CMD_EXTEND_SEED).

	Name
int	**wh_Client_SheEncEcbRequest**(whClientContext * c, uint8_t keyId, uint8_t * in, uint32_t sz)Sends an AES-ECB encryption request (CMD_ENC_ECB).
int	**wh_Client_SheEncEcbResponse**(whClientContext * c, uint8_t * out, uint32_t sz)Receives the AES-ECB ciphertext (CMD_ENC_ECB).
int	**wh_Client_SheEncEcb**(whClientContext * c, uint8_t keyId, uint8_t * in, uint8_t * out, uint32_t sz)Encrypts data with AES-ECB using a blocking call (CMD_ENC_ECB).
int	**wh_Client_SheEncCbcRequest**(whClientContext * c, uint8_t keyId, uint8_t * iv, uint32_t ivSz, uint8_t * in, uint32_t sz)Sends an AES-CBC encryption request (CMD_ENC_CBC).
int	**wh_Client_SheEncCbcResponse**(whClientContext * c, uint8_t * out, uint32_t sz)Receives the AES-CBC ciphertext (CMD_ENC_CBC).
int	**wh_Client_SheEncCbc**(whClientContext * c, uint8_t keyId, uint8_t * iv, uint32_t ivSz, uint8_t * in, uint8_t * out, uint32_t sz)Encrypts data with AES-CBC using a blocking call (CMD_ENC_CBC).
int	**wh_Client_SheDecEcbRequest**(whClientContext * c, uint8_t keyId, uint8_t * in, uint32_t sz)Sends an AES-ECB decryption request (CMD_DEC_ECB).
int	**wh_Client_SheDecEcbResponse**(whClientContext * c, uint8_t * out, uint32_t sz)Receives the AES-ECB plaintext (CMD_DEC_ECB).
int	**wh_Client_SheDecEcb**(whClientContext * c, uint8_t keyId, uint8_t * in, uint8_t * out, uint32_t sz)Decrypts data with AES-ECB using a blocking call (CMD_DEC_ECB).
int	**wh_Client_SheDecCbcRequest**(whClientContext * c, uint8_t keyId, uint8_t * iv, uint32_t ivSz, uint8_t * in, uint32_t sz)Sends an AES-CBC decryption request (CMD_DEC_CBC).
int	**wh_Client_SheDecCbcResponse**(whClientContext * c, uint8_t * out, uint32_t sz)Receives the AES-CBC plaintext (CMD_DEC_CBC).
int	**wh_Client_SheDecCbc**(whClientContext * c, uint8_t keyId, uint8_t * iv, uint32_t ivSz, uint8_t * in, uint8_t * out, uint32_t sz)Decrypts data with AES-CBC using a blocking call (CMD_DEC_CBC).
int	**wh_Client_SheGenerateMacRequest**(whClientContext * c, uint8_t keyId, uint8_t * in, uint32_t sz)Sends a CMAC generation request (CMD_GENERATE_MAC).

	Name
int	**wh_Client_SheGenerateMacResponse**(whClientContext * c, uint8_t * out, uint32_t sz)Receives the generated CMAC (CMD_GENERATE_MAC).
int	**wh_Client_SheGenerateMac**(whClientContext * c, uint8_t keyId, uint8_t * in, uint32_t inSz, uint8_t * out, uint32_t outSz)Generates a CMAC with a blocking call (CMD_GENERATE_MAC).
int	**wh_Client_SheVerifyMacRequest**(whClientContext * c, uint8_t keyId, uint8_t * message, uint32_t messageLen, uint8_t * mac, uint32_t macLen)Sends a CMAC verification request (CMD_VERIFY_MAC).
int	**wh_Client_SheVerifyMacResponse**(whClientContext * c, uint8_t * outStatus)Receives the CMAC verification result (CMD_VERIFY_MAC).
int	**wh_Client_SheVerifyMac**(whClientContext * c, uint8_t keyId, uint8_t * message, uint32_t messageLen, uint8_t * mac, uint32_t macLen, uint8_t * outStatus)Verifies a CMAC with a blocking call (CMD_VERIFY_MAC).

A.3.2 Functions Documentation

A.3.2.1 function wh_Client_ShePreProgramKey

```
int wh_Client_ShePreProgramKey(
    whClientContext * c,
    whNvmId keyId,
    whNvmFlags flags,
    uint8_t * key,
    whNvmSize keySz
)
```

Pre-programs a SHE key directly into NVM, bypassing the key update protocol.

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to write (0-15, e.g. WH_SHE_MASTER_ECU_KEY_ID).
- **flags** SHE key protection flags to store with the key (WH_SHE_FLAG_WRITE_PROTECT, WH_SHE_FLAG_BOOT_PROTECT, etc.).
- **key** Pointer to the key material to store.
- **keySz** Length of the key material in bytes (WH_SHE_KEY_SZ, 16).

Return: int Returns 0 on success, or a negative error code on failure.

SHE provisioning and identity functions

This is a wolfHSM-specific provisioning helper that has no equivalent in the AUTOSAR SHE command set. It writes key straight into the SHE NVM slot keyId with an update counter of zero, skipping the encrypted M1-M5 CMD_LOAD_KEY protocol. It is intended for the initial provisioning of a blank device (for example installing the MASTER_ECU_KEY or BOOT_MAC at production) before any key-update authorization key exists; subsequent in-field updates should use the spec-compliant wh_Client_SheLoadKey(). The key is scoped to the calling client via the keyId USER field.

A.3.2.2 function wh_Client_SheSetUidRequest

```
int wh_Client_SheSetUidRequest(
    whClientContext * c,
    uint8_t * uid,
    uint32_t uidSz
)
```

Sends a request to set the ECU UID (wolfHSM-specific).

Parameters:

- **c** Pointer to the client context.
- **uid** Pointer to the UID bytes to install.
- **uidSz** Length of uid; must be at least WH_SHE_UID_SZ (15).

Return: int Returns 0 on success, or a negative error code on failure.

Sends a request carrying the 15-byte unique identifier. This command has no AUTOSAR SHE equivalent: the spec assumes the UID is fused in hardware, but a software SHE module must be told its UID. The server rejects most SHE commands until the UID has been set, and the key update protocol binds the M1/M4 messages against this value.

A.3.2.3 function wh_Client_SheSetUidResponse

```
int wh_Client_SheSetUidResponse(
    whClientContext * c
)
```

Receives the response to a set UID request.

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

A.3.2.4 function wh_Client_SheSetUid

```
int wh_Client_SheSetUid(
    whClientContext * c,
    uint8_t * uid,
    uint32_t uidSz
)
```

Sets the ECU UID with a blocking call (wolfHSM-specific).

Parameters:

- **c** Pointer to the client context.
- **uid** Pointer to the UID bytes to install.
- **uidSz** Length of uid; must be at least WH_SHE_UID_SZ (15).

Return: int Returns 0 on success, or a negative error code on failure.

Sends a set UID request and busy-polls for the response. See wh_Client_SheSetUidRequest() for why this command exists outside the AUTOSAR SHE command set. The UID must be set before most other SHE commands will be accepted by the server.

A.3.2.5 function wh_Client_SheSecureBoot

```
int wh_Client_SheSecureBoot(
    whClientContext * c,
    uint8_t * bootloader,
    uint32_t bootloaderLen
)
```

Runs the SHE secure boot measurement with a blocking call (CMD_SECURE_BOOT).

Parameters:

- **c** Pointer to the client context.
- **bootloader** Pointer to the bootloader image to measure.
- **bootloaderLen** Length of the bootloader image in bytes.

Return: int Returns 0 on success, or a negative error code on failure. A boot measurement mismatch is reported via the status register (wh_Client_SheGetStatus), not as an error return.

SHE secure boot functions

Implements the AUTOSAR SHE CMD_SECURE_BOOT command by driving the three-phase INIT/UPDATE/FINISH state machine in a single blocking call. The server initializes a CMAC with the bootloader length, streams `bootloader` into it in chunks of up to WOLFHSM_CFG_COMM_DATA_LEN (repeating the UPDATE phase as needed), then finalizes the CMAC using BOOT_MAC_KEY (slot 2) and compares it against the stored BOOT_MAC (slot 3). The outcome is reported through the status register: a match sets WH_SHE_SREG_BOOT_OK while a mismatch leaves it clear; either way WH_SHE_SREG_BOOT_FINISHED is set. Until secure boot succeeds the server refuses most non-boot SHE commands.

A.3.2.6 function wh_Client_SheGetStatusRequest

```
int wh_Client_SheGetStatusRequest(
    whClientContext * c
)
```

Sends a request to read the SHE status register (CMD_GET_STATUS).

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

SHE status functions

Sends an AUTOSAR SHE CMD_GET_STATUS request. This is a zero-length message.

A.3.2.7 function wh_Client_SheGetStatusResponse

```
int wh_Client_SheGetStatusResponse(
    whClientContext * c,
    uint8_t * sreg
)
```

Receives the SHE status register response (CMD_GET_STATUS).

Parameters:

- **c** Pointer to the client context.
- **sreg** Pointer to a byte that receives the status register value.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

Consumes a CMD_GET_STATUS response and writes the 8-bit status register (SREG) to `sreg`. The SREG bits are defined in wolfhsm/wh_she_common.h (WH_SHE_SREG_SECURE_BOOT, WH_SHE_SREG_BOOT_FINISHED, WH_SHE_SREG_BOOT_OK, WH_SHE_SREG_RND_INIT, ...).

A.3.2.8 function wh_Client_SheGetStatus

```
int wh_Client_SheGetStatus(
    whClientContext * c,
    uint8_t * sreg
)
```

Reads the SHE status register with a blocking call (CMD_GET_STATUS).

Parameters:

- **c** Pointer to the client context.
- **sreg** Pointer to a byte that receives the status register value.

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_GET_STATUS request and busy-polls for the response, writing the 8-bit status register (SREG) to `sreg`. See wolfhsm/wh_she_common.h for the SREG bit definitions.

A.3.2.9 function wh_Client_SheLoadKeyRequest

```
int wh_Client_SheLoadKeyRequest(
    whClientContext * c,
    uint8_t * messageOne,
    uint8_t * messageTwo,
    uint8_t * messageThree
)
```

Sends an encrypted key update request (CMD_LOAD_KEY).

Parameters:

- **c** Pointer to the client context.
- **messageOne** M1: WH_SHE_M1_SZ (16) bytes - UID, key ID, auth ID.
- **messageTwo** M2: WH_SHE_M2_SZ (32) bytes - encrypted counter, flags, and the new key.
- **messageThree** M3: WH_SHE_M3_SZ (16) bytes - CMAC over M1 | M2.

Return: int Returns 0 on success, or a negative error code on failure.

SHE key management functions

Sends the three AUTOSAR SHE key update messages. M1, M2, and M3 encode the target slot, authorization key, new key, update counter, and protection flags, encrypted and CMAC-protected under keys derived from the authorization key using the spec's Miyaguchi-Preneel construction.

A.3.2.10 function wh_Client_SheLoadKeyResponse

```
int wh_Client_SheLoadKeyResponse(
    whClientContext * c,
    uint8_t * messageFour,
    uint8_t * messageFive
)
```

Receives the encrypted key update response (CMD_LOAD_KEY).

Parameters:

- **c** Pointer to the client context.
- **messageFour** M4: WH_SHE_M4_SZ (32) bytes - proof of storage.
- **messageFive** M5: WH_SHE_M5_SZ (16) bytes - CMAC over M4.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure (e.g. WH_SHE_ERC_KEY_UPDATE_ERROR, WH_SHE_ERC_WRITE_PROTECTED).

Consumes the M4 and M5 verification messages the server returns after storing the new key. M4 proves the new key and counter were stored, and M5 is a CMAC over M4; both are derived from the newly loaded key so the client can confirm the update was applied correctly.

A.3.2.11 function wh_Client_SheLoadKey

```
int wh_Client_SheLoadKey(
    whClientContext * c,
    uint8_t * messageOne,
    uint8_t * messageTwo,
    uint8_t * messageThree,
    uint8_t * messageFour,
    uint8_t * messageFive
)
```

Performs an encrypted SHE key update with a blocking call (CMD_LOAD_KEY).

Parameters:

- **c** Pointer to the client context.
- **messageOne** M1: WH_SHE_M1_SZ (16) bytes.
- **messageTwo** M2: WH_SHE_M2_SZ (32) bytes.
- **messageThree** M3: WH_SHE_M3_SZ (16) bytes.
- **messageFour** M4: WH_SHE_M4_SZ (32) bytes.
- **messageFive** M5: WH_SHE_M5_SZ (16) bytes.

Return: int Returns 0 on success, or a negative error code on failure.

Sends the M1/M2/M3 key update messages and busy-polls for the M4/M5 verification response. This is the AUTOSAR SHE protocol for installing or updating any key slot other than RAM_KEY: the server derives the authorization keys, verifies M3, decrypts M2, enforces the strictly increasing update counter and the WRITE_PROTECT / UID / WILDCARD constraints, stores the new key, and returns M4/M5. Use wh_Client_SheLoadPlainKey() to load the volatile RAM_KEY in plaintext.

A.3.2.12 function wh_Client_SheLoadPlainKeyRequest

```
int wh_Client_SheLoadPlainKeyRequest(
    whClientContext * c,
    uint8_t * key,
    uint32_t keySz
)
```

Sends a request to load the RAM key in plaintext (CMD_LOAD_PLAIN_KEY).

Parameters:

- **c** Pointer to the client context.
- **key** Pointer to the plaintext key material.
- **keySz** Length of key; must be at least WH_SHE_KEY_SZ (16).

Return: int Returns 0 on success, or a negative error code on failure.

Sends an AUTOSAR SHE CMD_LOAD_PLAIN_KEY request, which installs a plaintext key directly into the volatile RAM_KEY slot (ID 14) without the encrypted M1-M5 protocol. RAM_KEY lives only in the server key cache and is lost on power cycle.

A.3.2.13 function wh_Client_SheLoadPlainKeyResponse

```
int wh_Client_SheLoadPlainKeyResponse(
    whClientContext * c
)
```

Receives the response to a load plain key request (CMD_LOAD_PLAIN_KEY).

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

A.3.2.14 function wh_Client_SheLoadPlainKey

```
int wh_Client_SheLoadPlainKey(
    whClientContext * c,
    uint8_t * key,
    uint32_t keySz
)
```

Loads the RAM key in plaintext with a blocking call (CMD_LOAD_PLAIN_KEY).

Parameters:

- **c** Pointer to the client context.
- **key** Pointer to the plaintext key material.
- **keySz** Length of key; must be at least WH_SHE_KEY_SZ (16).

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_LOAD_PLAIN_KEY request and busy-polls for the response, installing key into the volatile RAM_KEY slot (ID 14), which is lost on power cycle.

A.3.2.15 function wh_Client_SheExportRamKeyRequest

```
int wh_Client_SheExportRamKeyRequest(
    whClientContext * c
)
```

Sends a request to export the RAM key (CMD_EXPORT_RAM_KEY).

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

Sends an AUTOSAR SHE CMD_EXPORT_RAM_KEY request. This is a zero-length message.

A.3.2.16 function wh_Client_SheExportRamKeyResponse

```
int wh_Client_SheExportRamKeyResponse(
    whClientContext * c,
    uint8_t * messageOne,
    uint8_t * messageTwo,
    uint8_t * messageThree,
    uint8_t * messageFour,
    uint8_t * messageFive
)
```

Receives the exported RAM key as an M1-M5 blob (CMD_EXPORT_RAM_KEY).

Parameters:

- **c** Pointer to the client context.
- **messageOne** M1: WH_SHE_M1_SZ (16) bytes.
- **messageTwo** M2: WH_SHE_M2_SZ (32) bytes.
- **messageThree** M3: WH_SHE_M3_SZ (16) bytes.
- **messageFour** M4: WH_SHE_M4_SZ (32) bytes.
- **messageFive** M5: WH_SHE_M5_SZ (16) bytes.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

Consumes the CMD_EXPORT_RAM_KEY response, which returns the current RAM_KEY (ID 14) packaged as the five key update messages bound to the MASTER_ECU_KEY. The resulting M1-M5 blob can be transferred to a peer SHE module and installed there via the CMD_LOAD_KEY protocol.

A.3.2.17 function wh_Client_SheExportRamKey

```
int wh_Client_SheExportRamKey(
    whClientContext * c,
    uint8_t * messageOne,
    uint8_t * messageTwo,
    uint8_t * messageThree,
    uint8_t * messageFour,
    uint8_t * messageFive
)
```

Exports the RAM key as an M1-M5 blob with a blocking call (CMD_EXPORT_RAM_KEY).

Parameters:

- **c** Pointer to the client context.
- **messageOne** M1: WH_SHE_M1_SZ (16) bytes.
- **messageTwo** M2: WH_SHE_M2_SZ (32) bytes.
- **messageThree** M3: WH_SHE_M3_SZ (16) bytes.
- **messageFour** M4: WH_SHE_M4_SZ (32) bytes.
- **messageFive** M5: WH_SHE_M5_SZ (16) bytes.

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_EXPORT_RAM_KEY request and busy-polls for the response. The RAM_KEY is returned as the five key update messages (M1-M5) bound to the MASTER_ECU_KEY, suitable for loading into a peer with wh_Client_SheLoadKey().

A.3.2.18 function wh_Client_SheInitRndRequest

```
int wh_Client_SheInitRndRequest(
    whClientContext * c
)
```

Sends a request to initialize the SHE PRNG (CMD_INIT_RNG).

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

SHE PRNG functions

Sends an AUTOSAR SHE CMD_INIT_RNG request. This is a zero-length message.

A.3.2.19 function wh_Client_SheInitRndResponse

```
int wh_Client_SheInitRndResponse(
    whClientContext * c
)
```

Receives the response to a PRNG initialization request (CMD_INIT_RNG).

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

On success the server has derived the PRNG key and state from SECRET_KEY and PRNG_SEED, advanced and persisted the seed, and set WH_SHE_SREG_RND_INIT in the status register.

A.3.2.20 function wh_Client_SheInitRnd

```
int wh_Client_SheInitRnd(
    whClientContext * c
)
```

Initializes the SHE PRNG with a blocking call (CMD_INIT_RNG).

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_INIT_RNG request and busy-polls for the response. The deterministic PRNG must be initialized with this command before wh_Client_SheRnd() can draw random data.

A.3.2.21 function wh_Client_SheRndRequest

```
int wh_Client_SheRndRequest(
    whClientContext * c
)
```

Sends a request to draw random bytes from the SHE PRNG (CMD_RND).

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, or a negative error code on failure.

Sends an AUTOSAR SHE CMD_RND request. This is a zero-length message. The PRNG must have been initialized with wh_Client_SheInitRnd().

A.3.2.22 function wh_Client_SheRndResponse

```
int wh_Client_SheRndResponse(
    whClientContext * c,
    uint8_t * out,
    uint32_t * outSz
)
```

Receives random bytes from the SHE PRNG (CMD_RND).

Parameters:

- **c** Pointer to the client context.
- **out** Buffer that receives the random bytes.
- **outSz** On input, the size of out (must be at least WH_SHE_KEY_SZ); on output, the number of bytes written (WH_SHE_KEY_SZ).

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

Consumes a CMD_RND response and copies the generated random block into out. SHE produces exactly WH_SHE_KEY_SZ (16) bytes per call.

A.3.2.23 function wh_Client_SheRnd

```
int wh_Client_SheRnd(
    whClientContext * c,
    uint8_t * out,
    uint32_t * outSz
)
```

Draws random bytes from the SHE PRNG with a blocking call (CMD_RND).

Parameters:

- **c** Pointer to the client context.
- **out** Buffer that receives the random bytes.
- **outSz** On input, the size of out (must be at least WH_SHE_KEY_SZ); on output, the number of bytes written.

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_RND request and busy-polls for the response, copying WH_SHE_KEY_SZ (16) bytes of PRNG output into out. wh_Client_SheInitRnd() must have been called first.

A.3.2.24 function wh_Client_SheExtendSeedRequest

```
int wh_Client_SheExtendSeedRequest(
    whClientContext * c,
    uint8_t * entropy,
    uint32_t entropySz
)
```

Sends a request to reseed the SHE PRNG (CMD_EXTEND_SEED).

Parameters:

- **c** Pointer to the client context.
- **entropy** Pointer to the additional entropy to mix in.
- **entropySz** Length of entropy; must equal WH_SHE_KEY_SZ (16).

Return: int Returns 0 on success, or a negative error code on failure.

Sends an AUTOSAR SHE CMD_EXTEND_SEED request, mixing caller-supplied entropy into the PRNG state.

A.3.2.25 function wh_Client_SheExtendSeedResponse

```
int wh_Client_SheExtendSeedResponse(
    whClientContext * c
)
```

Receives the response to a PRNG reseed request (CMD_EXTEND_SEED).

Parameters:

- **c** Pointer to the client context.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

On success the server has folded the supplied entropy into the PRNG state and written the updated PRNG_SEED back to NVM so the reseed survives reboots.

A.3.2.26 function wh_Client_SheExtendSeed

```
int wh_Client_SheExtendSeed(
    whClientContext * c,
    uint8_t * entropy,
    uint32_t entropySz
)
```

Reseeds the SHE PRNG with a blocking call (CMD_EXTEND_SEED).

Parameters:

- **c** Pointer to the client context.
- **entropy** Pointer to the additional entropy to mix in.
- **entropySz** Length of entropy; must equal WH_SHE_KEY_SZ (16).

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_EXTEND_SEED request and busy-polls for the response, mixing WH_SHE_KEY_SZ (16) bytes of caller-supplied entropy into the PRNG state and persisting the updated seed.

A.3.2.27 function wh_Client_SheEncEcbRequest

```
int wh_Client_SheEncEcbRequest(
    whClientContext * c,
    uint8_t keyId,
    uint8_t * in,
    uint32_t sz
)
```

Sends an AES-ECB encryption request (CMD_ENC_ECB).

Parameters:

- **c** Pointer to the client context.

- **keyId** SHE key slot to encrypt with.
- **in** Pointer to the plaintext.
- **sz** Length of the plaintext in bytes; must be at least WH_SHE_KEY_SZ (16) and fit within WOLFHSM_CFG_COMM_DATA_LEN.

Return: int Returns 0 on success, or a negative error code on failure.

SHE cipher functions

Sends an AUTOSAR SHE CMD_ENC_ECB request to encrypt `in` under the key in slot `keyId`.

A.3.2.28 function wh_Client_SheEncEcbResponse

```
int wh_Client_SheEncEcbResponse(
    whClientContext * c,
    uint8_t * out,
    uint32_t sz
)
```

Receives the AES-ECB ciphertext (CMD_ENC_ECB).

Parameters:

- **c** Pointer to the client context.
- **out** Buffer that receives the ciphertext.
- **sz** Size of `out`; must be at least the input size.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

A.3.2.29 function wh_Client_SheEncEcb

```
int wh_Client_SheEncEcb(
    whClientContext * c,
    uint8_t keyId,
    uint8_t * in,
    uint8_t * out,
    uint32_t sz
)
```

Encrypts data with AES-ECB using a blocking call (CMD_ENC_ECB).

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to encrypt with.
- **in** Pointer to the plaintext.
- **out** Buffer that receives the ciphertext (at least `sz` bytes).
- **sz** Length of the data in bytes; must be at least WH_SHE_KEY_SZ.

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_ENC_ECB request and busy-polls for the response, encrypting `in` under the key in slot `keyId` and writing the ciphertext to `out`.

A.3.2.30 function wh_Client_SheEncCbcRequest

```
int wh_Client_SheEncCbcRequest(
    whClientContext * c,
    uint8_t keyId,
```

```
    uint8_t * iv,
    uint32_t ivSz,
    uint8_t * in,
    uint32_t sz
)
```

Sends an AES-CBC encryption request (CMD_ENC_CBC).

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to encrypt with.
- **iv** Pointer to the initialization vector.
- **ivSz** Length of `iv`; must be at least WH_SHE_KEY_SZ (16).
- **in** Pointer to the plaintext.
- **sz** Length of the plaintext in bytes; must be at least WH_SHE_KEY_SZ (16) and fit within WOLFHSM_CFG_COMM_DATA_LEN.

Return: int Returns 0 on success, or a negative error code on failure.

Sends an AUTOSAR SHE CMD_ENC_CBC request to encrypt `in` under the key in slot `keyId` using the supplied initialization vector.

A.3.2.31 function wh_Client_SheEncCbcResponse

```
int wh_Client_SheEncCbcResponse(
    whClientContext * c,
    uint8_t * out,
    uint32_t sz
)
```

Receives the AES-CBC ciphertext (CMD_ENC_CBC).

Parameters:

- **c** Pointer to the client context.
- **out** Buffer that receives the ciphertext.
- **sz** Size of `out`; must be at least the input size.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

A.3.2.32 function wh_Client_SheEncCbc

```
int wh_Client_SheEncCbc(
    whClientContext * c,
    uint8_t keyId,
    uint8_t * iv,
    uint32_t ivSz,
    uint8_t * in,
    uint8_t * out,
    uint32_t sz
)
```

Encrypts data with AES-CBC using a blocking call (CMD_ENC_CBC).

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to encrypt with.

- **iv** Pointer to the initialization vector.
- **ivSz** Length of `iv`; must be at least WH_SHE_KEY_SZ (16).
- **in** Pointer to the plaintext.
- **out** Buffer that receives the ciphertext (at least `sz` bytes).
- **sz** Length of the data in bytes; must be at least WH_SHE_KEY_SZ.

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_ENC_CBC request and busy-polls for the response, encrypting `in` under the key in slot `keyId` with the supplied `iv`.

A.3.2.33 function wh_Client_SheDecEcbRequest

```
int wh_Client_SheDecEcbRequest(
    whClientContext * c,
    uint8_t keyId,
    uint8_t * in,
    uint32_t sz
)
```

Sends an AES-ECB decryption request (CMD_DEC_ECB).

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to decrypt with.
- **in** Pointer to the ciphertext.
- **sz** Length of the ciphertext in bytes; must be at least WH_SHE_KEY_SZ (16) and fit within WOLFHSM_CFG_COMM_DATA_LEN.

Return: int Returns 0 on success, or a negative error code on failure.

Sends an AUTOSAR SHE CMD_DEC_ECB request to decrypt `in` under the key in slot `keyId`.

A.3.2.34 function wh_Client_SheDecEcbResponse

```
int wh_Client_SheDecEcbResponse(
    whClientContext * c,
    uint8_t * out,
    uint32_t sz
)
```

Receives the AES-ECB plaintext (CMD_DEC_ECB).

Parameters:

- **c** Pointer to the client context.
- **out** Buffer that receives the plaintext.
- **sz** Size of `out`; must be at least the input size.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

A.3.2.35 function wh_Client_SheDecEcb

```
int wh_Client_SheDecEcb(
    whClientContext * c,
    uint8_t keyId,
    uint8_t * in,
    uint8_t * out,
```

```
    uint32_t sz
)
```

Decrypts data with AES-ECB using a blocking call (CMD_DEC_ECB).

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to decrypt with.
- **in** Pointer to the ciphertext.
- **out** Buffer that receives the plaintext (at least `sz` bytes).
- **sz** Length of the data in bytes; must be at least WH_SHE_KEY_SZ.

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_DEC_ECB request and busy-polls for the response, decrypting `in` under the key in slot `keyId` and writing the plaintext to `out`.

A.3.2.36 function wh_Client_SheDecCbcRequest

```
int wh_Client_SheDecCbcRequest(
    whClientContext * c,
    uint8_t keyId,
    uint8_t * iv,
    uint32_t ivSz,
    uint8_t * in,
    uint32_t sz
)
```

Sends an AES-CBC decryption request (CMD_DEC_CBC).

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to decrypt with.
- **iv** Pointer to the initialization vector.
- **ivSz** Length of `iv`; must be at least WH_SHE_KEY_SZ (16).
- **in** Pointer to the ciphertext.
- **sz** Length of the ciphertext in bytes; must be at least WH_SHE_KEY_SZ (16) and fit within WOLFHSM_CFG_COMM_DATA_LEN.

Return: int Returns 0 on success, or a negative error code on failure.

Sends an AUTOSAR SHE CMD_DEC_CBC request to decrypt `in` under the key in slot `keyId` using the supplied initialization vector.

A.3.2.37 function wh_Client_SheDecCbcResponse

```
int wh_Client_SheDecCbcResponse(
    whClientContext * c,
    uint8_t * out,
    uint32_t sz
)
```

Receives the AES-CBC plaintext (CMD_DEC_CBC).

Parameters:

- **c** Pointer to the client context.
- **out** Buffer that receives the plaintext.
- **sz** Size of `out`; must be at least the input size.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

A.3.2.38 function wh_Client_SheDecCbc

```
int wh_Client_SheDecCbc(
    whClientContext * c,
    uint8_t keyId,
    uint8_t * iv,
    uint32_t ivSz,
    uint8_t * in,
    uint8_t * out,
    uint32_t sz
)
```

Decrypts data with AES-CBC using a blocking call (CMD_DEC_CBC).

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to decrypt with.
- **iv** Pointer to the initialization vector.
- **ivSz** Length of `iv`; must be at least WH_SHE_KEY_SZ (16).
- **in** Pointer to the ciphertext.
- **out** Buffer that receives the plaintext (at least `sz` bytes).
- **sz** Length of the data in bytes; must be at least WH_SHE_KEY_SZ.

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_DEC_CBC request and busy-polls for the response, decrypting `in` under the key in slot `keyId` with the supplied `iv`.

A.3.2.39 function wh_Client_SheGenerateMacRequest

```
int wh_Client_SheGenerateMacRequest(
    whClientContext * c,
    uint8_t keyId,
    uint8_t * in,
    uint32_t sz
)
```

Sends a CMAC generation request (CMD_GENERATE_MAC).

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to MAC with.
- **in** Pointer to the message to authenticate.
- **sz** Length of the message in bytes; must be at least WH_SHE_KEY_SZ (16) and fit within WOLFHSM_CFG_COMM_DATA_LEN.

Return: int Returns 0 on success, or a negative error code on failure.

SHE MAC functions

Sends an AUTOSAR SHE CMD_GENERATE_MAC request to compute a CMAC over `in` using the key in slot `keyId`.

A.3.2.40 function wh_Client_SheGenerateMacResponse

```
int wh_Client_SheGenerateMacResponse(
    whClientContext * c,
    uint8_t * out,
    uint32_t sz
)
```

Receives the generated CMAC (CMD_GENERATE_MAC).

Parameters:

- **c** Pointer to the client context.
- **out** Buffer that receives the WH_SHE_KEY_SZ (16) byte CMAC.
- **sz** Size of out; must be at least WH_SHE_KEY_SZ (16).

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

A.3.2.41 function wh_Client_SheGenerateMac

```
int wh_Client_SheGenerateMac(
    whClientContext * c,
    uint8_t keyId,
    uint8_t * in,
    uint32_t inSz,
    uint8_t * out,
    uint32_t outSz
)
```

Generates a CMAC with a blocking call (CMD_GENERATE_MAC).

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to MAC with.
- **in** Pointer to the message to authenticate.
- **inSz** Length of the message in bytes; must be at least WH_SHE_KEY_SZ (16).
- **out** Buffer that receives the CMAC.
- **outSz** Size of out; must be at least WH_SHE_KEY_SZ (16).

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_GENERATE_MAC request and busy-polls for the response, computing a CMAC over in under the key in slot keyId and writing the 16-byte tag to out.

A.3.2.42 function wh_Client_SheVerifyMacRequest

```
int wh_Client_SheVerifyMacRequest(
    whClientContext * c,
    uint8_t keyId,
    uint8_t * message,
    uint32_t messageLen,
    uint8_t * mac,
    uint32_t macLen
)
```

Sends a CMAC verification request (CMD_VERIFY_MAC).

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to verify with.
- **message** Pointer to the message that was authenticated.
- **messageLen** Length of `message`; must be at least WH_SHE_KEY_SZ (16). The message plus MAC must fit within WOLFHSM_CFG_COMM_DATA_LEN.
- **mac** Pointer to the CMAC to verify.
- **macLen** Length of `mac`; must be at least WH_SHE_KEY_SZ (16).

Return: int Returns 0 on success, or a negative error code on failure.

Sends an AUTOSAR SHE CMD_VERIFY_MAC request to verify `mac` against a freshly computed CMAC of `message` using the key in slot `keyId`.

A.3.2.43 function wh_Client_SheVerifyMacResponse

```
int wh_Client_SheVerifyMacResponse(
    whClientContext * c,
    uint8_t * outStatus
)
```

Receives the CMAC verification result (CMD_VERIFY_MAC).

Parameters:

- **c** Pointer to the client context.
- **outStatus** Set to 0 if the MAC verified successfully, or 1 if it did not match.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if no response is available yet, or a negative error code on failure.

Consumes a CMD_VERIFY_MAC response and reports whether the MAC matched through `outStatus`. Per the SHE spec a verification mismatch is a normal result rather than an error: the function still returns success and signals the mismatch via `outStatus`.

A.3.2.44 function wh_Client_SheVerifyMac

```
int wh_Client_SheVerifyMac(
    whClientContext * c,
    uint8_t keyId,
    uint8_t * message,
    uint32_t messageLen,
    uint8_t * mac,
    uint32_t macLen,
    uint8_t * outStatus
)
```

Verifies a CMAC with a blocking call (CMD_VERIFY_MAC).

Parameters:

- **c** Pointer to the client context.
- **keyId** SHE key slot to verify with.
- **message** Pointer to the message that was authenticated.
- **messageLen** Length of `message`; must be at least WH_SHE_KEY_SZ.
- **mac** Pointer to the CMAC to verify.
- **macLen** Length of `mac`; must be at least WH_SHE_KEY_SZ (16).
- **outStatus** Set to 0 if the MAC verified successfully, or 1 if it did not match.

Return: int Returns 0 on success, or a negative error code on failure.

Sends a CMD_VERIFY_MAC request and busy-polls for the response, verifying `mac` against a CMAC of `message` under the key in slot `keyId`. A verification mismatch is reported through `outStatus`, not as an error return.

A.3.3 Source code

```
/*
 * Copyright (C) 2024 wolfSSL Inc.
 *
 * This file is part of wolfHSM.
 *
 * wolfHSM is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
 * wolfHSM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with wolfHSM.  If not, see <http://www.gnu.org/licenses/>.
 */
/*
 * wolfhsm/wh_client_she.h
 *
 * Client API for the AUTOSAR SHE (Secure Hardware Extension) subsystem.
 *
 * This header declares the client-side interface to the optional wolfHSM SHE
 * extension, which is compiled only when WOLFHSM_CFG_SHE_EXTENSION is defined.
 * The API maps one-to-one onto the AUTOSAR SHE command set (CMD_SECURE_BOOT,
 * CMD_LOAD_KEY, CMD_RND, ...): each spec command is exposed as a
 * wh_Client_She* function that marshals the request to the wolfHSM server,
 * which implements the spec-compliant SHE behavior on top of the wolfHSM
 * keystore, NVM, and crypto infrastructure.
 *
 * AUTOSAR SHE defines a small, fixed-function security module for automotive
 * ECUs: a set of 128-bit AES key slots with fixed roles, an encrypted key
 * update protocol (the M1-M5 messages), a CMAC-based secure boot measurement,
 * a deterministic PRNG, and an 8-bit status register (SREG). See the AUTOSAR
 * "Specification of Secure Hardware Extensions" for the authoritative command
 * and protocol definitions.
 *
 * SHE key slots have fixed roles (defined in wolfhsm/wh_she_common.h):
 *   ID 0      SECRET_KEY      master secret consumed by the PRNG derivation
 *   ID 1      MASTER_ECU_KEY  ECU identity / key-update authorization key
 *   ID 2      BOOT_MAC_KEY    key used to CMAC the bootloader at secure boot
 *   ID 3      BOOT_MAC        expected bootloader CMAC compared at secure boot
 *   ID 4-13   user key slots  general-purpose keys
 *   ID 14     RAM_KEY         volatile slot, lost on power cycle
 *   ID 15     PRNG_SEED       persistent PRNG seed state
 *
 * Client SHE commands return WH_SHE_ERC_NO_ERROR (defined to WH_ERROR_OK,
```

```
 * i.e. 0) on success. On a SHE protocol failure they return one of the
 * WH_SHE_ERC_* status codes from wolfhsm/wh_error.h. On a transport or
↪  argument
 * failure they return a negative wolfHSM error code such as WH_ERROR_BADARGS.
 *
 */

#ifndef WOLFHSM_WH_CLIENT_SHE_H_
#define WOLFHSM_WH_CLIENT_SHE_H_

/* Pick up compile-time configuration */
#include "wolfhsm/wh_settings.h"

#ifdef WOLFHSM_CFG_SHE_EXTENSION

/* System libraries */
#include <stdint.h>


#include "wolfhsm/wh_common.h"
#include "wolfhsm/wh_client.h"

int wh_Client_ShePreProgramKey(whClientContext* c, whNvmId keyId,
    whNvmFlags flags, uint8_t* key, whNvmSize keySz);

int wh_Client_SheSetUidRequest(whClientContext* c, uint8_t* uid,
    uint32_t uidSz);

int wh_Client_SheSetUidResponse(whClientContext* c);

int wh_Client_SheSetUid(whClientContext* c, uint8_t* uid, uint32_t uidSz);

int wh_Client_SheSecureBoot(whClientContext* c, uint8_t* bootloader,
    uint32_t bootloaderLen);

int wh_Client_SheGetStatusRequest(whClientContext* c);

int wh_Client_SheGetStatusResponse(whClientContext* c, uint8_t* sreg);

int wh_Client_SheGetStatus(whClientContext* c, uint8_t* sreg);

int wh_Client_SheLoadKeyRequest(whClientContext* c, uint8_t* messageOne,
    uint8_t* messageTwo, uint8_t* messageThree);

int wh_Client_SheLoadKeyResponse(whClientContext* c, uint8_t* messageFour,
    uint8_t* messageFive);

int wh_Client_SheLoadKey(whClientContext* c, uint8_t* messageOne,
    uint8_t* messageTwo, uint8_t* messageThree, uint8_t* messageFour,
    uint8_t* messageFive);

int wh_Client_SheLoadPlainKeyRequest(whClientContext* c, uint8_t* key,
    uint32_t keySz);
```

```
int wh_Client_SheLoadPlainKeyResponse(whClientContext* c);

int wh_Client_SheLoadPlainKey(whClientContext* c, uint8_t* key, uint32_t
 ↪ keySz);

int wh_Client_SheExportRamKeyRequest(whClientContext* c);

int wh_Client_SheExportRamKeyResponse(whClientContext* c, uint8_t* messageOne,
    uint8_t* messageTwo, uint8_t* messageThree, uint8_t* messageFour,
    uint8_t* messageFive);

int wh_Client_SheExportRamKey(whClientContext* c, uint8_t* messageOne,
    uint8_t* messageTwo, uint8_t* messageThree, uint8_t* messageFour,
    uint8_t* messageFive);

int wh_Client_SheInitRndRequest(whClientContext* c);

int wh_Client_SheInitRndResponse(whClientContext* c);

int wh_Client_SheInitRnd(whClientContext* c);

int wh_Client_SheRndRequest(whClientContext* c);

int wh_Client_SheRndResponse(whClientContext* c, uint8_t* out, uint32_t*
 ↪ outSz);

int wh_Client_SheRnd(whClientContext* c, uint8_t* out, uint32_t* outSz);

int wh_Client_SheExtendSeedRequest(whClientContext* c, uint8_t* entropy,
    uint32_t entropySz);

int wh_Client_SheExtendSeedResponse(whClientContext* c);

int wh_Client_SheExtendSeed(whClientContext* c, uint8_t* entropy,
    uint32_t entropySz);

int wh_Client_SheEncEcbRequest(whClientContext* c, uint8_t keyId, uint8_t* in,
    uint32_t sz);

int wh_Client_SheEncEcbResponse(whClientContext* c, uint8_t* out, uint32_t sz);

int wh_Client_SheEncEcb(whClientContext* c, uint8_t keyId, uint8_t* in,
    uint8_t* out, uint32_t sz);

int wh_Client_SheEncCbcRequest(whClientContext* c, uint8_t keyId, uint8_t* iv,
    uint32_t ivSz, uint8_t* in, uint32_t sz);

int wh_Client_SheEncCbcResponse(whClientContext* c, uint8_t* out, uint32_t sz);

int wh_Client_SheEncCbc(whClientContext* c, uint8_t keyId, uint8_t* iv,
    uint32_t ivSz, uint8_t* in, uint8_t* out, uint32_t sz);

int wh_Client_SheDecEcbRequest(whClientContext* c, uint8_t keyId, uint8_t* in,
    uint32_t sz);
```

```
int wh_Client_SheDecEcbResponse(whClientContext* c, uint8_t* out, uint32_t sz);

int wh_Client_SheDecEcb(whClientContext* c, uint8_t keyId, uint8_t* in,
    uint8_t* out, uint32_t sz);

int wh_Client_SheDecCbcRequest(whClientContext* c, uint8_t keyId, uint8_t* iv,
    uint32_t ivSz, uint8_t* in, uint32_t sz);

int wh_Client_SheDecCbcResponse(whClientContext* c, uint8_t* out, uint32_t sz);

int wh_Client_SheDecCbc(whClientContext* c, uint8_t keyId, uint8_t* iv,
    uint32_t ivSz, uint8_t* in, uint8_t* out, uint32_t sz);

int wh_Client_SheGenerateMacRequest(whClientContext* c, uint8_t keyId,
    uint8_t* in, uint32_t sz);

int wh_Client_SheGenerateMacResponse(whClientContext* c, uint8_t* out,
    uint32_t sz);

int wh_Client_SheGenerateMac(whClientContext* c, uint8_t keyId, uint8_t* in,
    uint32_t inSz, uint8_t* out, uint32_t outSz);

int wh_Client_SheVerifyMacRequest(whClientContext* c, uint8_t keyId,
    uint8_t* message, uint32_t messageLen, uint8_t* mac, uint32_t macLen);

int wh_Client_SheVerifyMacResponse(whClientContext* c, uint8_t* outStatus);

int wh_Client_SheVerifyMac(whClientContext* c, uint8_t keyId, uint8_t* message,
    uint32_t messageLen, uint8_t* mac, uint32_t macLen, uint8_t* outStatus);

#endif /* WOLFHSM_CFG_SHE_EXTENSION */

#endif /* !WOLFHSM_WH_CLIENT_SHE_H_ */
```

B Server API Reference

This chapter is the complete reference for the wolfHSM **server** API. It is generated directly from the documentation comments in the public server headers (`wolfhsm/wh_server.h`, `wolfhsm/wh_server_keystore.h`, `wolfhsm/wh_server_img_mgr.h`, `wolfhsm/wh_server_cert.h`, and `wolfhsm/wh_server_cert_cache.h`), so it always tracks the source. For a conceptual, feature-oriented walkthrough of what these functions are for, see Features; this chapter documents the precise signatures, parameters, and return values.

- **Server API** — server context lifecycle, configuration, and top-level request dispatch (`wolfhsm/wh_server.h`).
- **Server Keystore API** — server-side key cache and keystore operations: cache and evict slots, NVM commit, key export, and metadata access (`wolfhsm/wh_server_keystore.h`).
- **Server Image Manager API** — image manager configuration, verification, and verify-state queries (`wolfhsm/wh_server_img_mgr.h`).
- **Server Cert API** — server-side certificate manager: trusted root storage, chain verification, attribute certificate support, and the user-injectable verify callback (`wolfhsm/wh_server_cert.h`).
- **Server Cert Cache API** — trusted-cert verify-result cache: hash-based result lookup and invalidation (`wolfhsm/wh_server_cert_cache.h`).

B.1 wolfhsm/wh_server.h

B.1.1 Functions

	Name
int	**wh_Server_Init**(whServerContext * server, whServerConfig * config)Initializes the server context with the provided configuration.
int	**wh_Server_SetConnected**(whServerContext * server, whCommConnected connected)Sets the connection state of the server.
int	**wh_Server_SetConnectedCb**(void * s, whCommConnected connected)Sets a callback function that should be invoked by the underlying transport after it is initialized.
int	**wh_Server_GetConnected**(whServerContext * server, whCommConnected * out_connected)Gets the connection state of the server.
int	**wh_Server_HandleRequestMessage**(whServerContext * server)Handles incoming request messages and dispatches them to the appropriate handlers.
int	**wh_Server_Cleanup**(whServerContext * server)Cleans up the server context and associated resources.
int	**wh_Server_RegisterCustomCb**(whServerContext * server, uint16_t action, whServerCustomCb handler)Registers a custom callback handler for a specific action.

	Name
int	**wh_Server_HandleCustomCbRequest**(whServerContext * server, uint16_t magic, uint16_t action, uint16_t seq, uint16_t req_size, const void * req_packet, uint16_t * out_resp_size, void * resp_packet)Handles incoming custom callback requests.
int	**wh_Server_DmaRegisterCb**(struct whServerContext_t * server, whServerDmaClientMemCb cb)Registers a custom client DMA callback.
int	**wh_Server_DmaRegisterMemCopyCb**(whServerContext * server, whServerDmaMemCopyCb cb)Registers a custom memory copy callback for DMA operations. This function allows the server to register a callback that will be invoked during DMA memory copy operations. The callback overrides the use of memcpy when copying to and from client memory. This is useful if standard memcpy cannot be used to copy data back and forth between the client, even after client addresses are transformed through the standard DMA callbacks (e.g. if client memory can only be accessed though a hardware FIFO or register interface)
int	**wh_Server_DmaRegisterAllowList**(struct whServerContext_t * server, const whServerDmaAddrAllowList * allowlist)Registers the allowable client read/write addresses for DMA.
int	**wh_Server_DmaCheckMemOperAllowed**(const struct whServerContext_t * server, whServerDmaOper oper, void * addr, size_t size)Checks if a DMA memory operation is allowed based on the server's allowlist.
int	**wh_Server_DmaProcessClientAddress**(struct whServerContext_t * server, uintptr_t clientAddr, void ** serverPtr, size_t len, whServerDmaOper oper, whServerDmaFlags flags)Processes a client address for DMA operations, using the native pointer size of the system.
int	**wh_Server_NvmLock**(whServerContext * server)
int	**wh_Server_NvmUnlock**(whServerContext * server)

B.1.2 Functions Documentation

B.1.2.1 function wh_Server_Init

```
int wh_Server_Init(
    whServerContext * server,
```

```
    whServerConfig * config
)
```

Initializes the server context with the provided configuration.

Parameters:

- **server** Pointer to the server context.
- **config** Pointer to the server configuration.

Return: int Returns 0 on success, WH_ERROR_BADARGS if the arguments are invalid, or WH_ERROR_ABORTED if initialization fails.

Public server context functions

This function must be called before any other server functions are used on the supplied context. If a Crypto component is configured it MUST be initialized before calling this function.

The NVM component is OPTIONAL: config->nvm may be NULL. With no NVM backing the server still runs and crypto works through the key cache when keys are primed (cached directly or via wrapped keys). Keystore lookups that miss the cache return WH_ERROR_NOTFOUND (as if the key were absent from NVM), and operations that inherently require persistence (the NVM request API, certificate-chain verification against stored roots, counters, key commit, SHE key/seed persistence, image-signature loading) fail at runtime rather than crashing. If config->nvm is provided, behavior is unchanged.

B.1.2.2 function wh_Server_SetConnected

```
int wh_Server_SetConnected(
    whServerContext * server,
    whCommConnected connected
)
```

Sets the connection state of the server.

Parameters:

- **server** Pointer to the server context.
- **connected** The connection state to set.

Return: int Returns 0 on success, or WH_ERROR_BADARGS if the arguments are invalid.

The connection state indicates whether the server is ready to handle incoming requests. This function should be invoked when the underlying transport is ready for use.

B.1.2.3 function wh_Server_SetConnectedCb

```
int wh_Server_SetConnectedCb(
    void * s,
    whCommConnected connected
)
```

Sets a callback function that should be invoked by the underlying transport after it is initialized.

Parameters:

- **s** Pointer to the server context.
- **connected** The connection state to set.

Return: int Returns 0 on success.

The connection state indicates whether the server is ready to handle incoming requests. This function should be invoked when the underlying transport is ready for use.

B.1.2.4 function wh_Server_GetConnected

```
int wh_Server_GetConnected(
    whServerContext * server,
    whCommConnected * out_connected
)
```

Gets the connection state of the server.

Parameters:

- **server** Pointer to the server context.
- **out_connected** Pointer to store the connection state.

Return: int Returns 0 on success, or WH_ERROR_BADARGS if the arguments are invalid.

B.1.2.5 function wh_Server_HandleRequestMessage

```
int wh_Server_HandleRequestMessage(
    whServerContext * server
)
```

Handles incoming request messages and dispatches them to the appropriate handlers.

Parameters:

- **server** Pointer to the server context.

Return: int Returns 0 on success, WH_ERROR_BADARGS if the arguments are invalid, WH_ERROR_NOTREADY if the server is not connected or no data is available, or a negative error code on failure.

This function processes incoming request messages from the communication server in a non-blocking fashion. It determines the message group and action, and dispatches the request to the appropriate handler. The function also sends a response back to the client.

B.1.2.6 function wh_Server_Cleanup

```
int wh_Server_Cleanup(
    whServerContext * server
)
```

Cleans up the server context and associated resources.

Parameters:

- **server** Pointer to the server context.

Return: int Returns WH_ERROR_OK on success, or WH_ERROR_BADARGS if the arguments are invalid.

This function releases any resources associated with the server context, including communication server resources. It resets the server context to its initial state.

B.1.2.7 function wh_Server_RegisterCustomCb

```
int wh_Server_RegisterCustomCb(
    whServerContext * server,
    uint16_t action,
    whServerCustomCb handler
)
```

Registers a custom callback handler for a specific action.

Parameters:

- **server** Pointer to the server context.
- **actionId** The action ID for which the callback is being registered.
- **cb** The custom callback handler to register.

Return: int Returns WH_ERROR_OK on success, or WH_ERROR_BADARGS if the arguments are invalid.

Server custom callback functions

This function allows the server to register a custom callback handler for a specific action ID. The callback will be invoked when a request with the corresponding action ID is received.

B.1.2.8 function wh_Server_HandleCustomCbRequest

```
int wh_Server_HandleCustomCbRequest(
    whServerContext * server,
    uint16_t magic,
    uint16_t action,
    uint16_t seq,
    uint16_t req_size,
    const void * req_packet,
    uint16_t * out_resp_size,
    void * resp_packet
)
```

Handles incoming custom callback requests.

Parameters:

- **server** Pointer to the server context.
- **magic** The magic number for the request.
- **action** The action ID of the request.
- **seq** The sequence number of the request.
- **req_size** The size of the request packet.
- **req_packet** Pointer to the request packet data.
- **out_resp_size** Pointer to store the size of the response packet.
- **resp_packet** Pointer to store the response packet data.

Return: int Returns WH_ERROR_OK on success, WH_ERROR_BADARGS if the arguments are invalid, WH_ERROR_ABORTED if the request is malformed, or a negative error code on failure.

This function processes incoming custom callback requests by invoking the registered custom callback handler for the specified action. It translates the request and response messages and sends the appropriate response back to the client.

B.1.2.9 function wh_Server_DmaRegisterCb

```
int wh_Server_DmaRegisterCb(
    struct whServerContext_t * server,
    whServerDmaClientMemCb cb
)
```

Registers a custom client DMA callback.

Parameters:

- **server** Pointer to the server context.
- **cb** The custom DMA callback handler to register.

Return: int Returns WH_ERROR_OK on success, or WH_ERROR_BADARGS if the arguments are invalid.

Server DMA functions

This function allows the server to register a custom callback handler for processing client memory operations. The callback will be invoked during DMA operations to transform client addresses, manipulate caches, etc.

B.1.2.10 function wh_Server_DmaRegisterMemCopyCb

```
int wh_Server_DmaRegisterMemCopyCb(
    whServerContext * server,
    whServerDmaMemCopyCb cb
)
```

Registers a custom memory copy callback for DMA operations. This function allows the server to register a callback that will be invoked during DMA memory copy operations. The callback overrides the use of memcpy when copying to and from client memory. This is useful if standard memcpy cannot be used to copy data back and forth between the client, even after client addresses are transformed through the standard DMA callbacks (e.g. if client memory can only be accessed though a hardware FIFO or register interface)

Parameters:

- **server** Pointer to the server context.
- **cb** The custom memory copy callback handler to register.

Return: int Returns WH_ERROR_OK on success, or WH_ERROR_BADARGS if the arguments are invalid.

B.1.2.11 function wh_Server_DmaRegisterAllowList

```
int wh_Server_DmaRegisterAllowList(
    struct whServerContext_t * server,
    const whServerDmaAddrAllowList * allowlist
)
```

Registers the allowable client read/write addresses for DMA.

Parameters:

- **server** Pointer to the server context.
- **allowlist** Pointer to the list of allowable client addresses.

Return: int Returns WH_ERROR_OK on success, or WH_ERROR_BADARGS if the arguments are invalid.

This function allows the server to register a list of allowable client addresses for DMA read and write operations. The server will check these addresses during DMA operations to ensure they are within the allowed range for the client

B.1.2.12 function wh_Server_DmaCheckMemOperAllowed

```
int wh_Server_DmaCheckMemOperAllowed(
    const struct whServerContext_t * server,
    whServerDmaOper oper,
    void * addr,
    size_t size
)
```

Checks if a DMA memory operation is allowed based on the server's allowlist.

Parameters:

- **server** Pointer to the server context.
- **oper** The DMA operation type (e.g., read or write).

- **addr** The address to be checked.
- **size** The size of the memory operation.

Return: int Returns WH_ERROR_OK if the operation is allowed, WH_ERROR_BADARGS if the arguments are invalid, or WH_ERROR_ACCESS if the operation is not allowed.

This function verifies whether a specified DMA memory operation is permitted by checking the operation type and the address range against the server's registered allowlist. If no allowlist is registered, the operation is allowed.

B.1.2.13 function wh_Server_DmaProcessClientAddress

```
int wh_Server_DmaProcessClientAddress(
    struct whServerContext_t * server,
    uintptr_t clientAddr,
    void ** serverPtr,
    size_t len,
    whServerDmaOper oper,
    whServerDmaFlags flags
)
```

Processes a client address for DMA operations, using the native pointer size of the system.

Parameters:

- **server** Pointer to the server context.
- **clientAddr** The client address to be processed.
- **serverPtr** Pointer to store the transformed server address.
- **len** The length of the memory operation.
- **oper** The DMA operation type (e.g., read or write).
- **flags** Flags for the DMA operation.

Return: int Returns WH_ERROR_OK on success, WH_ERROR_BADARGS if the arguments are invalid, or a negative error code on failure.

This function transforms a client address for DMA operations. It performs user-supplied address transformations, cache manipulations, and checks the transformed address against the server's allowlist if registered.

B.1.2.14 function wh_Server_NvmLock

```
int wh_Server_NvmLock(
    whServerContext * server
)
```

Server NVM Locking API for handler-level thread safety

B.1.2.15 function wh_Server_NvmUnlock

```
int wh_Server_NvmUnlock(
    whServerContext * server
)
```

B.1.3 Source code

```
/*
 * Copyright (C) 2024 wolfSSL Inc.
 *
```

```
 * This file is part of wolfHSM.
 *
 * wolfHSM is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
 * wolfHSM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with wolfHSM.  If not, see <http://www.gnu.org/licenses/>.
 */
/*
 * wolfhsm/wh_server.h
 *
*
 */

#ifndef WOLFHSM_WH_SERVER_H_
#define WOLFHSM_WH_SERVER_H_

/* Pick up compile-time configuration */
#include "wolfhsm/wh_settings.h"

#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>

/* Forward declaration of the server structure so its elements can reference
 * itself  (e.g. server argument to custom callback) */
typedef struct whServerContext_t whServerContext;

#include "wolfhsm/wh_common.h"
#include "wolfhsm/wh_comm.h"
#include "wolfhsm/wh_keycache.h"
#include "wolfhsm/wh_server_cert_cache.h"
#include "wolfhsm/wh_nvm.h"
#ifdef WOLFHSM_CFG_ENABLE_AUTHENTICATION
#include "wolfhsm/wh_auth.h"
#endif /* WOLFHSM_CFG_ENABLE_AUTHENTICATION */
#ifdef WOLFHSM_CFG_HWKEYSTORE
#include "wolfhsm/wh_hwkeystore.h"
#endif /* WOLFHSM_CFG_HWKEYSTORE */
#include "wolfhsm/wh_message_customcb.h"
#include "wolfhsm/wh_log.h"
#ifdef WOLFHSM_CFG_DMA
#include "wolfhsm/wh_dma.h"
#endif /* WOLFHSM_CFG_DMA */

#ifndef WOLFHSM_CFG_NO_CRYPTO
#include "wolfssl/wolfcrypt/settings.h"
```

```
#include "wolfssl/wolfcrypt/types.h"
#include "wolfssl/wolfcrypt/random.h"
#include "wolfssl/wolfcrypt/rsa.h"
#include "wolfssl/wolfcrypt/ecc.h"
#include "wolfssl/wolfcrypt/curve25519.h"
#include "wolfssl/wolfcrypt/cryptocb.h"
#include "wolfssl/wolfcrypt/sha256.h"
#endif /* !WOLFHSM_CFG_NO_CRYPTO */

#ifdef WOLFHSM_CFG_SHE_EXTENSION
#include "wolfhsm/wh_she_common.h"
#include "wolfhsm/wh_server_she.h"
#endif


#ifndef WOLFHSM_CFG_NO_CRYPTO

typedef struct whServerCryptoContext {
#ifndef WC_NO_RNG
    WC_RNG rng[1];
#else
    /* Placeholder to prevent empty struct in C90 */
    uint8_t WH_PAD[1];
#endif
} whServerCryptoContext;


#endif /* !WOLFHSM_CFG_NO_CRYPTO */

/* Type definition for a custom server callback  */
typedef int (*whServerCustomCb)(
    whServerContext* server,             /* points to dispatching server ctx */
    const whMessageCustomCb_Request* req, /* request from client to callback */
    whMessageCustomCb_Response*      resp /* response from callback to client */
);


#ifdef WOLFHSM_CFG_DMA

/* Maintain existing naming for common DMA types */
typedef whDmaAddrAllowList whServerDmaAddrAllowList;
typedef whDmaOper          whServerDmaOper;
typedef whDmaFlags         whServerDmaFlags;
typedef whDmaAddr          whServerDmaAddr;
typedef whDmaAddrList      whServerDmaAddrList;
#ifdef WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY
typedef whDmaCopyOper whServerDmaCopyOper;
#endif /* WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY */

/* DMA callbacks invoked internally by wolfHSM before and after every client
 * memory operation. */
typedef int (*whServerDmaClientMemCb)(struct whServerContext_t* server,
                                      uintptr_t clientAddr, void** serverPtr,
                                      size_t len, whServerDmaOper oper,
```

```
                                    whServerDmaFlags flags);

#ifdef WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY
/* DMA callback invoked to copy from the client */
typedef int (*whServerDmaMemCopyCb)(struct whServerContext_t* server,
                                    uintptr_t clientAddr, uintptr_t serverPtr,
                                     size_t len, whServerDmaCopyOper oper,
                                     whServerDmaFlags flags);
#endif /* WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY */

/* Server DMA configuration struct for initializing a server */
typedef struct {
    whServerDmaClientMemCb          cb;               /* DMA callback */
#ifdef WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY
    whServerDmaMemCopyCb            memCopyCb;       /* DMA memory copy callback
 ↪  */
#endif /* WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY */
    const whServerDmaAddrAllowList* dmaAddrAllowList; /* allowed addresses */
} whServerDmaConfig;

typedef struct {
    whServerDmaClientMemCb          cb;               /* DMA callback */
#ifdef WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY
    whServerDmaMemCopyCb            memCopyCb;       /* DMA memory copy callback
 ↪  */
#endif /* WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY */
    const whServerDmaAddrAllowList* dmaAddrAllowList; /* allowed addresses */
} whServerDmaContext;
#endif /* WOLFHSM_CFG_DMA */

typedef struct whServerConfig_t {
    whCommServerConfig* comm_config;
    whNvmContext*       nvm; /* optional; NULL = no NVM backing */
#ifdef WOLFHSM_CFG_ENABLE_AUTHENTICATION
    whAuthContext* auth;
#endif /* WOLFHSM_CFG_ENABLE_AUTHENTICATION */
#ifdef WOLFHSM_CFG_HWKEYSTORE
    whHwKeystoreContext* hwKeystore; /* optional; NULL = no HW keystore */
#endif                                /* WOLFHSM_CFG_HWKEYSTORE */

#ifndef WOLFHSM_CFG_NO_CRYPTO
    whServerCryptoContext* crypto;
#ifdef WOLFHSM_CFG_SHE_EXTENSION
    whServerSheContext* she;
#endif /* WOLFHSM_CFG_SHE_EXTENSION */
#if defined WOLF_CRYPTO_CB
    int devId;
#endif /* WOLF_CRYPTO_CB */
#endif /* !WOLFHSM_CFG_NO_CRYPTO */
#ifdef WOLFHSM_CFG_DMA
    whServerDmaConfig* dmaConfig;
#endif /* WOLFHSM_CFG_DMA */
#ifdef WOLFHSM_CFG_LOGGING
    whLogConfig* logConfig;
```

```
#endif /* WOLFHSM_CFG_LOGGING */
#if defined(WOLFHSM_CFG_CERTIFICATE_MANAGER) && !defined(WOLFHSM_CFG_NO_CRYPTO)
    whServerCertConfig* certConfig; /* optional; NULL = no verify callback */
#endif /* WOLFHSM_CFG_CERTIFICATE_MANAGER && !WOLFHSM_CFG_NO_CRYPTO */
} whServerConfig;


/* Context structure to maintain the state of an HSM server */
struct whServerContext_t {
    whNvmContext* nvm;
#ifdef WOLFHSM_CFG_ENABLE_AUTHENTICATION
    whAuthContext* auth;
#endif /* WOLFHSM_CFG_ENABLE_AUTHENTICATION */
#ifdef WOLFHSM_CFG_HWKEYSTORE
    whHwKeystoreContext* hwKeystore;
#endif /* WOLFHSM_CFG_HWKEYSTORE */
    whCommServer  comm[1];
#ifndef WOLFHSM_CFG_NO_CRYPTO
    whServerCryptoContext* crypto;
    int                    devId;
    whKeyCacheContext      localCache; /* Unified cache structure */
#ifdef WOLFHSM_CFG_SHE_EXTENSION
    whServerSheContext* she;
#endif
#endif /* !WOLFHSM_CFG_NO_CRYPTO */
    whServerCustomCb   customHandlerTable[WOLFHSM_CFG_SERVER_CUSTOMCB_COUNT];
#ifdef WOLFHSM_CFG_DMA
    whServerDmaContext dma;
#endif /* WOLFHSM_CFG_DMA */
    int connected;
#ifdef WOLFHSM_CFG_LOGGING
    whLogContext log;
#endif /* WOLFHSM_CFG_LOGGING */
#if defined(WOLFHSM_CFG_CERTIFICATE_MANAGER) && !defined(WOLFHSM_CFG_NO_CRYPTO)
    whServerCertContext cert; /* verify callback + verify cache */
#endif /* WOLFHSM_CFG_CERTIFICATE_MANAGER && !WOLFHSM_CFG_NO_CRYPTO */
};


/* Initialize the comms and crypto cache components.
 * Note: Crypto components must be initialized prior to Server Init. NVM, if
 * provided, must also be initialized first; NVM is optional (see below).
 */

int wh_Server_Init(whServerContext* server, whServerConfig* config);

int wh_Server_SetConnected(whServerContext* server, whCommConnected connected);

int wh_Server_SetConnectedCb(void* s, whCommConnected connected);

int wh_Server_GetConnected(whServerContext* server,
                           whCommConnected* out_connected);

int wh_Server_HandleRequestMessage(whServerContext* server);
```

```
int wh_Server_Cleanup(whServerContext* server);

int wh_Server_RegisterCustomCb(whServerContext* server, uint16_t action,
                               whServerCustomCb handler);

int wh_Server_HandleCustomCbRequest(whServerContext* server, uint16_t magic,
                                    uint16_t action, uint16_t seq,
                                    uint16_t req_size, const void* req_packet,
                                   uint16_t* out_resp_size, void* resp_packet);

#ifdef WOLFHSM_CFG_DMA

int wh_Server_DmaRegisterCb(struct whServerContext_t* server,
                            whServerDmaClientMemCb    cb);


#ifdef WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY
int wh_Server_DmaRegisterMemCopyCb(whServerContext*     server,
                                   whServerDmaMemCopyCb cb);
#endif /* WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY */

int wh_Server_DmaRegisterAllowList(struct whServerContext_t*       server,
                                   const whServerDmaAddrAllowList* allowlist);

int wh_Server_DmaCheckMemOperAllowed(const struct whServerContext_t* server,
                                     whServerDmaOper oper, void* addr,
                                     size_t size);

int wh_Server_DmaProcessClientAddress(struct whServerContext_t* server,
                                      uintptr_t clientAddr, void** serverPtr,
                                      size_t len, whServerDmaOper oper,
                                      whServerDmaFlags flags);

int whServerDma_CopyFromClient(struct whServerContext_t* server,
                               void* serverPtr, uintptr_t clientAddr,
                               size_t len, whServerDmaFlags flags);

int whServerDma_CopyToClient(struct whServerContext_t* server,
                             uintptr_t clientAddr, void* serverPtr, size_t len,
                             whServerDmaFlags flags);
#endif /* WOLFHSM_CFG_DMA */

#ifdef WOLFHSM_CFG_THREADSAFE
int wh_Server_NvmLock(whServerContext* server);
int wh_Server_NvmUnlock(whServerContext* server);
#define WH_SERVER_NVM_LOCK(server) wh_Server_NvmLock(server)
#define WH_SERVER_NVM_UNLOCK(server) wh_Server_NvmUnlock(server)
#else
#define WH_SERVER_NVM_LOCK(server) (WH_ERROR_OK)
#define WH_SERVER_NVM_UNLOCK(server) (WH_ERROR_OK)
#endif

#endif /* !WOLFHSM_WH_SERVER_H_ */
```

B.2 wolfhsm/wh_server_keystore.h

B.2.1 Functions

	Name
int	**wh_Server_KeystoreGetUniqueId**(whServerContext * server, whNvmId * inout_id)Find a new unique key ID using the top bits of inout_id for user and type.
int	**wh_Server_KeystoreGetCacheSlot**(whServerContext * server, whKeyId keyId, uint16_t keySz, uint8_t ** outBuf, whNvmMetadata ** outMeta)Find an available cache slot for the specified key size.
int	**wh_Server_KeystoreGetCacheSlotChecked**(whServerContext * server, whKeyId keyId, uint16_t keySz, uint8_t ** outBuf, whNvmMetadata ** outMeta)
int	**wh_Server_KeystoreCacheKey**(whServerContext * server, whNvmMetadata * meta, uint8_t * in)Cache a key in server memory.
int	**wh_Server_KeystoreCacheKeyChecked**(whServerContext * server, whNvmMetadata * meta, uint8_t * in)Cache a key after enforcing keystore policy.
int	**wh_Server_KeystoreFreshenKey**(whServerContext * server, whKeyId keyId, uint8_t ** outBuf, whNvmMetadata ** outMeta)Ensure a key is in cache, loading it from NVM if necessary.
int	**wh_Server_KeystoreReadKey**(whServerContext * server, whKeyId keyId, whNvmMetadata * outMeta, uint8_t * out, uint32_t * outSz)Read a key from cache or NVM.
int	**wh_Server_KeystoreReadKeyChecked**(whServerContext * server, whKeyId keyId, whNvmMetadata * outMeta, uint8_t * out, uint32_t * outSz)Read a key with policy enforcement.
int	**wh_Server_KeystoreEvictKey**(whServerContext * server, whNvmId keyId)Remove a key from cache.
int	**wh_Server_KeystoreEvictKeyChecked**(whServerContext * server, whNvmId keyId)Evict a key with policy enforcement.
int	**wh_Server_KeystoreCommitKey**(whServerContext * server, whNvmId keyId)Commit a cached key to NVM storage.
int	**wh_Server_KeystoreCommitKeyChecked**(whServerContext * server, whNvmId keyId)Commit a cached key to NVM with policy enforcement.
int	**wh_Server_KeystoreEraseKey**(whServerContext * server, whNvmId keyId)Erase a key from both cache and NVM.
int	**wh_Server_KeystoreEraseKeyChecked**(whServerContext * server, whNvmId keyId)Erase a key with policy enforcement.

	Name
int	**wh_Server_KeystoreRevokeKey**(whServerContext * server, whKeyId keyId)Revoke a key (clears usage and marks non-modifiable)
int	**wh_Server_HandleKeyRequest**(whServerContext * server, uint16_t magic, uint16_t action, uint16_t req_size, const void * req_packet, uint16_t * out_resp_size, void * resp_packet)Handle key management requests from clients.
int	**wh_Server_KeystoreCacheKeyDma**(whServerContext * server, whNvmMetadata * meta, uint64_t keyAddr)Cache a key using DMA transfer.
int	**wh_Server_KeystoreCacheKeyDmaChecked**(whServerContext * server, whNvmMetadata * meta, uint64_t keyAddr)Cache a key with DMA after policy enforcement.
int	**wh_Server_KeystoreExportKeyDma**(whServerContext * server, whKeyId keyId, uint64_t keyAddr, uint64_t keySz, whNvmMetadata * outMeta)Export a key using DMA transfer.
int	**wh_Server_KeystoreExportKeyDmaChecked**(whServerContext * server, whKeyId keyId, uint64_t keyAddr, uint64_t keySz, whNvmMetadata * outMeta)Export a key with DMA after policy enforcement.
int	**wh_Server_KeystoreEnforceKeyUsage**(const whNvmMetadata * meta, whNvmFlags requiredUsage)Enforce key usage policy given metadata.
int	**wh_Server_KeystoreFindEnforceKeyUsage**(whServerContext * server, whKeyId keyId, whNvmFlags requiredUsage)

B.2.2 Functions Documentation

B.2.2.1 function wh_Server_KeystoreGetUniqueId

```
int wh_Server_KeystoreGetUniqueId(
    whServerContext * server,
    whNvmId * inout_id
)
```

Find a new unique key ID using the top bits of inout_id for user and type.

Parameters:

- **server** Server context
- **inout_id** Input: key ID with type and user set; Output: unique key ID

Return: 0 on success, error code on failure

Searches for an available key ID by checking against cache keys and NVM storage. The client_id and type should be set by caller on inout_id.

B.2.2.2 function wh_Server_KeystoreGetCacheSlot

```
int wh_Server_KeystoreGetCacheSlot(
    whServerContext * server,
    whKeyId keyId,
    uint16_t keySz,
    uint8_t ** outBuf,
    whNvmMetadata ** outMeta
)
```

Find an available cache slot for the specified key size.

Parameters:

- **server** Server context
- **keyId** Key ID (used to route to correct cache)
- **keySz** Size of the key in bytes
- **outBuf** Pointer to the cache buffer
- **outMeta** Pointer to the metadata structure

Return: 0 on success, error code on failure

Searches for an empty slot or a slot with a committed key that can be evicted. Returns the slot's buffer (zeroed) and metadata. Routes to the appropriate cache (global or local) based on keyId.

B.2.2.3 function wh_Server_KeystoreGetCacheSlotChecked

```
int wh_Server_KeystoreGetCacheSlotChecked(
    whServerContext * server,
    whKeyId keyId,
    uint16_t keySz,
    uint8_t ** outBuf,
    whNvmMetadata ** outMeta
)
```

B.2.2.4 function wh_Server_KeystoreCacheKey

```
int wh_Server_KeystoreCacheKey(
    whServerContext * server,
    whNvmMetadata * meta,
    uint8_t * in
)
```

Cache a key in server memory.

Parameters:

- **server** Server context
- **meta** Key metadata
- **in** Key data buffer

Return: 0 on success, error code on failure

Stores a key in the appropriate cache (regular or big) based on its size. Checks if the key is already committed to NVM.

B.2.2.5 function wh_Server_KeystoreCacheKeyChecked

```
int wh_Server_KeystoreCacheKeyChecked(
    whServerContext * server,
    whNvmMetadata * meta,
    uint8_t * in
)
```

Cache a key after enforcing keystore policy.

Runs policy checks (access/usage/etc.) before calling wh_Server_KeystoreCacheKey.

B.2.2.6 function wh_Server_KeystoreFreshenKey

```
int wh_Server_KeystoreFreshenKey(
    whServerContext * server,
    whKeyId keyId,
    uint8_t ** outBuf,
    whNvmMetadata ** outMeta
)
```

Ensure a key is in cache, loading it from NVM if necessary.

Parameters:

- **server** Server context
- **keyId** Key ID to freshen
- **outBuf** Pointer to the cached key buffer
- **outMeta** Pointer to the key metadata

Return: 0 on success, error code on failure

Tries to put the specified key into cache if it isn't already there. Returns pointers to the metadata and cached data.

B.2.2.7 function wh_Server_KeystoreReadKey

```
int wh_Server_KeystoreReadKey(
    whServerContext * server,
    whKeyId keyId,
    whNvmMetadata * outMeta,
    uint8_t * out,
    uint32_t * outSz
)
```

Read a key from cache or NVM.

Parameters:

- **server** Server context
- **keyId** Key ID to read
- **outMeta** Key metadata (can be NULL)
- **out** Buffer to store key data (can be NULL)
- **outSz** Input: size of out buffer; Output: actual key size

Return: 0 on success, error code on failure

Retrieves a key from cache or NVM storage and returns its metadata and data.

B.2.2.8 function wh_Server_KeystoreReadKeyChecked

```
int wh_Server_KeystoreReadKeyChecked(
    whServerContext * server,
    whKeyId keyId,
    whNvmMetadata * outMeta,
    uint8_t * out,
    uint32_t * outSz
)
```

Read a key with policy enforcement.

Performs keystore policy checks before reading from cache/NVM.

B.2.2.9 function wh_Server_KeystoreEvictKey

```
int wh_Server_KeystoreEvictKey(
    whServerContext * server,
    whNvmId keyId
)
```

Remove a key from cache.

Parameters:

- **server** Server context
- **keyId** Key ID to evict

Return: 0 on success, error code on failure

Marks the key as erased in the cache if present.

B.2.2.10 function wh_Server_KeystoreEvictKeyChecked

```
int wh_Server_KeystoreEvictKeyChecked(
    whServerContext * server,
    whNvmId keyId
)
```

Evict a key with policy enforcement.

Checks policy before removing the key from cache.

B.2.2.11 function wh_Server_KeystoreCommitKey

```
int wh_Server_KeystoreCommitKey(
    whServerContext * server,
    whNvmId keyId
)
```

Commit a cached key to NVM storage.

Parameters:

- **server** Server context
- **keyId** Key ID to commit

Return: 0 on success, error code on failure

Writes a key from cache to non-volatile memory and marks it as committed.

B.2.2.12 function wh_Server_KeystoreCommitKeyChecked

```
int wh_Server_KeystoreCommitKeyChecked(
    whServerContext * server,
    whNvmId keyId
)
```

Commit a cached key to NVM with policy enforcement.

Runs keystore policy checks before committing.

B.2.2.13 function wh_Server_KeystoreEraseKey

```
int wh_Server_KeystoreEraseKey(
    whServerContext * server,
    whNvmId keyId
)
```

Erase a key from both cache and NVM.

Parameters:

- **server** Server context
- **keyId** Key ID to erase

Return: 0 on success, error code on failure

Removes the key from cache if present and destroys it in NVM.

B.2.2.14 function wh_Server_KeystoreEraseKeyChecked

```
int wh_Server_KeystoreEraseKeyChecked(
    whServerContext * server,
    whNvmId keyId
)
```

Erase a key with policy enforcement.

Runs keystore policy checks before evicting/destroying.

B.2.2.15 function wh_Server_KeystoreRevokeKey

```
int wh_Server_KeystoreRevokeKey(
    whServerContext * server,
    whKeyId keyId
)
```

Revoke a key (clears usage and marks non-modifiable)

Placeholder implementation for key revocation.

B.2.2.16 function wh_Server_HandleKeyRequest

```
int wh_Server_HandleKeyRequest(
    whServerContext * server,
    uint16_t magic,
    uint16_t action,
    uint16_t req_size,
    const void * req_packet,
    uint16_t * out_resp_size,
```

```
    void * resp_packet
)
```

Handle key management requests from clients.

Parameters:

- **server** Server context
- **magic** Message magic number
- **action** Key operation to perform
- **req_size** Size of request packet
- **req_packet** Request packet data
- **out_resp_size** Size of response packet
- **resp_packet** Response packet data

Return: 0 on success, error code on failure

Processes various key operations including cache, export, evict, commit, and erase. Supports DMA operations if configured.

B.2.2.17 function wh_Server_KeystoreCacheKeyDma

```
int wh_Server_KeystoreCacheKeyDma(
    whServerContext * server,
    whNvmMetadata * meta,
    uint64_t keyAddr
)
```

Cache a key using DMA transfer.

Parameters:

- **server** Server context
- **meta** Key metadata
- **keyAddr** Client memory address containing key data

Return: 0 on success, error code on failure

Allocates a cache slot and copies key data from client memory using DMA.

B.2.2.18 function wh_Server_KeystoreCacheKeyDmaChecked

```
int wh_Server_KeystoreCacheKeyDmaChecked(
    whServerContext * server,
    whNvmMetadata * meta,
    uint64_t keyAddr
)
```

Cache a key with DMA after policy enforcement.

Performs policy checks before exporting a key via DMA.

B.2.2.19 function wh_Server_KeystoreExportKeyDma

```
int wh_Server_KeystoreExportKeyDma(
    whServerContext * server,
    whKeyId keyId,
    uint64_t keyAddr,
    uint64_t keySz,
```

```
    whNvmMetadata * outMeta
)
```

Export a key using DMA transfer.

Parameters:

- **server** Server context
- **keyId** Key ID to export
- **keyAddr** Client memory address to receive key data
- **keySz** Size of client memory buffer
- **outMeta** Buffer to receive key metadata

Return: 0 on success, error code on failure

Copies key data from server cache to client memory using DMA.

B.2.2.20 function wh_Server_KeystoreExportKeyDmaChecked

```
int wh_Server_KeystoreExportKeyDmaChecked(
    whServerContext * server,
    whKeyId keyId,
    uint64_t keyAddr,
    uint64_t keySz,
    whNvmMetadata * outMeta
)
```

Export a key with DMA after policy enforcement.

Performs policy checks before exporting a key via DMA.

B.2.2.21 function wh_Server_KeystoreEnforceKeyUsage

```
int wh_Server_KeystoreEnforceKeyUsage(
    const whNvmMetadata * meta,
    whNvmFlags requiredUsage
)
```

Enforce key usage policy given metadata.

Parameters:

- **meta** Pointer to key metadata
- **requiredUsage** Required usage policy flags (e.g., WH_NVM_FLAGS_USAGE_ENCRYPT | WH_NVM_FLAGS_USAGE_DECRYPT)

Return:

- WH_ERROR_OK if the key has all required usage flags set
- WH_ERROR_USAGE if the key does not have the required flags
- WH_ERROR_BADARGS if meta is NULL

Validates that a key has the required usage policy flags set in its metadata. This is a pure policy check function that does not perform any key lookups. Use this when you already have the key metadata available to avoid duplicate key freshening operations.

B.2.2.22 function wh_Server_KeystoreFindEnforceKeyUsage

```
int wh_Server_KeystoreFindEnforceKeyUsage(
    whServerContext * server,
    whKeyId keyId,
```

```
    whNvmFlags requiredUsage
)
```

Parameters:

- **server** Pointer to the server context
- **keyId** The translated server keyId (after client keyId translation)
- **requiredUsage** The required usage policy flags (e.g., WH_NVM_FLAGS_USAGE_ENCRYPT | WH_NVM_FLAGS_USAGE_DECRYPT)

Return:

- WH_ERROR_OK if the key has all required usage flags set
- WH_ERROR_USAGE if the key does not have the required flags
- Other error codes if key metadata cannot be retrieved

Validates that a key has the required usage policy flags set

This function enforces key usage policies by checking that the specified key has all the required usage flags set in its metadata. It retrieves the key metadata from the cache or NVM storage and performs a bitwise check against the required flags.

B.2.3 Source code

```c
/*
 * Copyright (C) 2024 wolfSSL Inc.
 *
 * This file is part of wolfHSM.
 *
 * wolfHSM is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
 * wolfHSM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with wolfHSM.  If not, see <http://www.gnu.org/licenses/>.
 */
/*
 * wolfhsm/wh_server_keystore.h
 *
 */
#ifndef WOLFHSM_WH_SERVER_KEYSTORE_H_
#define WOLFHSM_WH_SERVER_KEYSTORE_H_

/* Pick up compile-time configuration */
#include "wolfhsm/wh_settings.h"

#include <stdint.h>

#include "wolfhsm/wh_common.h"
#include "wolfhsm/wh_server.h"
```

```
int wh_Server_KeystoreGetUniqueId(whServerContext* server, whNvmId* inout_id);

int wh_Server_KeystoreGetCacheSlot(whServerContext* server, whKeyId keyId,
                                   uint16_t keySz, uint8_t** outBuf,
                                   whNvmMetadata** outMeta);
int wh_Server_KeystoreGetCacheSlotChecked(whServerContext* server,
                                          whKeyId keyId, uint16_t keySz,
                                          uint8_t**       outBuf,
                                          whNvmMetadata** outMeta);

int wh_Server_KeystoreCacheKey(whServerContext* server, whNvmMetadata* meta,
                               uint8_t* in);

int wh_Server_KeystoreCacheKeyChecked(whServerContext* server,
                                      whNvmMetadata* meta, uint8_t* in);

int wh_Server_KeystoreFreshenKey(whServerContext* server, whKeyId keyId,
                                 uint8_t** outBuf, whNvmMetadata** outMeta);

int wh_Server_KeystoreReadKey(whServerContext* server, whKeyId keyId,
                              whNvmMetadata* outMeta, uint8_t* out,
                              uint32_t* outSz);

int wh_Server_KeystoreReadKeyChecked(whServerContext* server, whKeyId keyId,
                                     whNvmMetadata* outMeta, uint8_t* out,
                                     uint32_t* outSz);

int wh_Server_KeystoreEvictKey(whServerContext* server, whNvmId keyId);

int wh_Server_KeystoreEvictKeyChecked(whServerContext* server, whNvmId keyId);

int wh_Server_KeystoreCommitKey(whServerContext* server, whNvmId keyId);

int wh_Server_KeystoreCommitKeyChecked(whServerContext* server, whNvmId keyId);

int wh_Server_KeystoreEraseKey(whServerContext* server, whNvmId keyId);

int wh_Server_KeystoreEraseKeyChecked(whServerContext* server, whNvmId keyId);

int wh_Server_KeystoreRevokeKey(whServerContext* server, whKeyId keyId);

int wh_Server_HandleKeyRequest(whServerContext* server, uint16_t magic,
                               uint16_t action, uint16_t req_size,
                               const void* req_packet, uint16_t* out_resp_size,
                               void* resp_packet);

int wh_Server_KeystoreCacheKeyDma(whServerContext* server, whNvmMetadata* meta,
                                  uint64_t keyAddr);

int wh_Server_KeystoreCacheKeyDmaChecked(whServerContext* server,
                                         whNvmMetadata* meta, uint64_t keyAddr);
int wh_Server_KeystoreExportKeyDma(whServerContext* server, whKeyId keyId,
                                   uint64_t keyAddr, uint64_t keySz,
                                   whNvmMetadata* outMeta);
```

```
int wh_Server_KeystoreExportKeyDmaChecked(whServerContext* server,
                                          whKeyId keyId, uint64_t keyAddr,
                                          uint64_t       keySz,
                                          whNvmMetadata* outMeta);


int wh_Server_KeystoreEnforceKeyUsage(const whNvmMetadata* meta,
                                      whNvmFlags           requiredUsage);

int wh_Server_KeystoreFindEnforceKeyUsage(whServerContext* server,
                                          whKeyId          keyId,
                                          whNvmFlags       requiredUsage);

#endif /* !WOLFHSM_WH_SERVER_KEYSTORE_H_ */
```

B.3 wolfhsm/wh_server_img_mgr.h

B.3.1 Functions

	Name
int	**wh_Server_ImgMgrInit**(whServerImgMgrContext * context, const whServerImgMgrConfig * config)Initialize the image manager.
int	**wh_Server_ImgMgrVerifyAll**(whServerImgMgrContext * context, whServerImgMgrVerifyResult * outResults, size_t outResultsCount, size_t * outErrorIdx)Verify all registered images.
int	**wh_Server_ImgMgrVerifyImg**(whServerImgMgrContext * context, const whServerImgMgrImg * img, whServerImgMgrVerifyResult * outResult)Verify an image by reference.
int	**wh_Server_ImgMgrVerifyImgIdx**(whServerImgMgrContext * context, size_t imgIdx, whServerImgMgrVerifyResult * outResult)Verify an image by index.
int	**wh_Server_ImgMgrVerifyMethodEccWithSha256**(whServerImg * context, const whServerImgMgrImg * img, const uint8_t * key, size_t keySz, const uint8_t * sig, size_t sigSz)Default ECC P256 verification method.
int	**wh_Server_ImgMgrVerifyMethodAesCmac**(whServerImgMgrC * context, const whServerImgMgrImg * img, const uint8_t * key, size_t keySz, const uint8_t * sig, size_t sigSz)AES128 CMAC verification method.
int	**wh_Server_ImgMgrVerifyMethodRsaSslWithSha256**(whServer * context, const whServerImgMgrImg * img, const uint8_t * key, size_t keySz, const uint8_t * sig, size_t sigSz)RSA2048 signature verification method.

	Name
int	**wh_Server_ImgMgrVerifyMethodWolfBootRsa4096WithSha25** * context, const whServerImgMgrImg * img, const uint8_t * key, size_t keySz, const uint8_t * sig, size_t sigSz)wolfBoot RSA4096+SHA256 verification method
int	**wh_Server_ImgMgrVerifyMethodWolfBootCertChainRsa4096** * context, const whServerImgMgrImg * img, const uint8_t * key, size_t keySz, const uint8_t * sig, size_t sigSz)wolfBoot RSA4096+SHA256 cert chain verification method
int	**wh_Server_ImgMgrVerifyActionDefault**(whServerImgMgrCont * context, const whServerImgMgrImg * img, int verifyResult)Default verification action callback.

B.3.2 Functions Documentation

B.3.2.1 function wh_Server_ImgMgrInit

```
int wh_Server_ImgMgrInit(
    whServerImgMgrContext * context,
    const whServerImgMgrConfig * config
)
```

Initialize the image manager.

Parameters:

- **context** Image manager context to initialize
- **config** Configuration containing image list

Return: WH_ERROR_OK on success, negative error code on failure

Initializes the image manager context with the provided configuration. Registers the list of images to be managed.

B.3.2.2 function wh_Server_ImgMgrVerifyAll

```
int wh_Server_ImgMgrVerifyAll(
    whServerImgMgrContext * context,
    whServerImgMgrVerifyResult * outResults,
    size_t outResultsCount,
    size_t * outErrorIdx
)
```

Verify all registered images.

Parameters:

- **context** Image manager context
- **outResults** Array to store verification results for each image
- **outResultsCount** Number of result slots in the results array (must be == context->imageCount)
- **outErrorIdx** Holds the index of the image whose verification returned an error (indicated via this function's return value). Only valid if return value is not WH_ERROR_OK. If no error occurred, this value is not updated.

Return: WH_ERROR_OK on success, negative error code on failure.

Iterates through all registered images and verifies each one. Calls the verification method and action callbacks for each image. Populates detailed verification results for each image.

B.3.2.3 function wh_Server_ImgMgrVerifyImg

```
int wh_Server_ImgMgrVerifyImg(
    whServerImgMgrContext * context,
    const whServerImgMgrImg * img,
    whServerImgMgrVerifyResult * outResult
)
```

Verify an image by reference.

Parameters:

- **context** Image manager context
- **img** Image structure to verify
- **outResult** Verification result. Only valid when function returns WH_ERROR_OK.

Return: WH_ERROR_OK on success, negative error code on failure

Verifies a specific image by direct reference to its structure. Performs DMA operations, key loading, and signature verification. Populates detailed verification results.

B.3.2.4 function wh_Server_ImgMgrVerifyImgIdx

```
int wh_Server_ImgMgrVerifyImgIdx(
    whServerImgMgrContext * context,
    size_t imgIdx,
    whServerImgMgrVerifyResult * outResult
)
```

Verify an image by index.

Parameters:

- **context** Image manager context
- **imgIdx** Index of image to verify
- **outResult** Verification result. Only valid when function returns WH_ERROR_OK.

Return: WH_ERROR_OK on success, negative error code on failure

Verifies a specific image by its index in the registered image array. Populates detailed verification results.

B.3.2.5 function wh_Server_ImgMgrVerifyMethodEccWithSha256

```
int wh_Server_ImgMgrVerifyMethodEccWithSha256(
    whServerImgMgrContext * context,
    const whServerImgMgrImg * img,
    const uint8_t * key,
    size_t keySz,
    const uint8_t * sig,
    size_t sigSz
)
```

Default ECC P256 verification method.

Parameters:

- **context** Image manager context

- **img** Image structure containing verification parameters
- **key** ECC public key data
- **keySz** Size of key data
- **sig** Signature data
- **sigSz** Size of signature data

Return: WH_ERROR_OK on successful verification, negative error code on failure

Default implementation of image verification using ECC P256 signatures. Uses wolfCrypt to hash the image data and verify the signature.

B.3.2.6 function wh_Server_ImgMgrVerifyMethodAesCmac

```
int wh_Server_ImgMgrVerifyMethodAesCmac(
    whServerImgMgrContext * context,
    const whServerImgMgrImg * img,
    const uint8_t * key,
    size_t keySz,
    const uint8_t * sig,
    size_t sigSz
)
```

AES128 CMAC verification method.

Parameters:

- **context** Image manager context
- **img** Image structure containing verification parameters
- **key** AES128 key data (16 bytes)
- **keySz** Size of key data (must be 16)
- **sig** CMAC signature data (16 bytes)
- **sigSz** Size of signature data (must be 16)

Return: WH_ERROR_OK on successful verification, negative error code on failure

Implementation of image verification using AES128 CMAC. Uses wolfCrypt to compute CMAC of the image data and compare with signature.

B.3.2.7 function wh_Server_ImgMgrVerifyMethodRsaSslWithSha256

```
int wh_Server_ImgMgrVerifyMethodRsaSslWithSha256(
    whServerImgMgrContext * context,
    const whServerImgMgrImg * img,
    const uint8_t * key,
    size_t keySz,
    const uint8_t * sig,
    size_t sigSz
)
```

RSA2048 signature verification method.

Parameters:

- **context** Image manager context
- **img** Image structure containing verification parameters
- **key** RSA public key data (DER format)
- **keySz** Size of key data
- **sig** RSA signature data
- **sigSz** Size of signature data

Return: WH_ERROR_OK on successful verification, negative error code on failure

Implementation of image verification using RSA2048 signatures. Uses wolfCrypt wc_RsaSSL_Verify to verify RSA signatures against SHA256 hash.

B.3.2.8 function wh_Server_ImgMgrVerifyMethodWolfBootRsa4096WithSha256

```
int wh_Server_ImgMgrVerifyMethodWolfBootRsa4096WithSha256(
    whServerImgMgrContext * context,
    const whServerImgMgrImg * img,
    const uint8_t * key,
    size_t keySz,
    const uint8_t * sig,
    size_t sigSz
)
```

wolfBoot RSA4096+SHA256 verification method

Parameters:

- **context** Image manager context
- **img** Image structure with hdrAddr pointing to wolfBoot header (must be 2-byte aligned) and addr pointing to firmware payload
- **key** RSA4096 public key data (DER format)
- **keySz** Size of key data
- **sig** Unused (NULL), signature is read from wolfBoot header
- **sigSz** Unused (0)

Return: WH_ERROR_OK on successful verification, negative error code on failure

Verifies a wolfBoot image using RSA4096 signature with SHA256 hash. The public key is provided by the framework (loaded from keystore). The signature is extracted from the wolfBoot header at img->hdrAddr.

B.3.2.9 function wh_Server_ImgMgrVerifyMethodWolfBootCertChainRsa4096WithSha256

```
int wh_Server_ImgMgrVerifyMethodWolfBootCertChainRsa4096WithSha256(
    whServerImgMgrContext * context,
    const whServerImgMgrImg * img,
    const uint8_t * key,
    size_t keySz,
    const uint8_t * sig,
    size_t sigSz
)
```

wolfBoot RSA4096+SHA256 cert chain verification method

Parameters:

- **context** Image manager context
- **img** Image structure with hdrAddr pointing to wolfBoot header (must be 2-byte aligned) and addr pointing to firmware payload
- **key** Unused (NULL), leaf key is extracted from cert chain
- **keySz** Unused (0)
- **sig** Unused (NULL), signature is read from wolfBoot header
- **sigSz** Unused (0)

Return: WH_ERROR_OK on successful verification, negative error code on failure

Verifies a wolfBoot image using RSA4096 signature with SHA256 hash, where the signing key is validated through a certificate chain. The root CA cert NVM ID is read from img->sigNvmId. The cert chain and signature are extracted from the wolfBoot header.

B.3.2.10 function wh_Server_ImgMgrVerifyActionDefault

```
int wh_Server_ImgMgrVerifyActionDefault(
    whServerImgMgrContext * context,
    const whServerImgMgrImg * img,
    int verifyResult
)
```

Default verification action callback.

Parameters:

- **context** Image manager context
- **img** Image structure containing verification parameters
- **verifyResult** Result from the verification method

Return: verifyResult (passes through the verification result)

Default implementation of post-verification action. Simply returns the verification result without performing additional actions.

B.3.3 Source code

```
/*
 * Copyright (C) 2025 wolfSSL Inc.
 *
 * This file is part of wolfHSM.
 *
 * wolfHSM is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
 * wolfHSM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with wolfHSM.  If not, see <http://www.gnu.org/licenses/>.
 */
/*
 * wolfhsm/wh_server_img_mgr.h
 *
 */

#ifndef WOLFHSM_WH_SERVER_IMG_MGR_H_
#define WOLFHSM_WH_SERVER_IMG_MGR_H_

#include "wolfhsm/wh_settings.h"

#ifdef WOLFHSM_CFG_SERVER_IMG_MGR
```

```c
#include <stdint.h>
#include <stddef.h>

#include "wolfhsm/wh_common.h"
#include "wolfhsm/wh_server.h"

/* wolfBoot header constants */
#define WH_IMG_MGR_WOLFBOOT_MAGIC 0x464C4F57 /* "WOLF" LE */
#define WH_IMG_MGR_WOLFBOOT_HDR_OFFSET 8

/* wolfBoot TLV types */
#define WH_IMG_MGR_WOLFBOOT_HDR_SHA256 0x03
#define WH_IMG_MGR_WOLFBOOT_HDR_IMG_TYPE 0x04
#define WH_IMG_MGR_WOLFBOOT_HDR_PUBKEY 0x10
#define WH_IMG_MGR_WOLFBOOT_HDR_SIGNATURE 0x20
#define WH_IMG_MGR_WOLFBOOT_HDR_CERT_CHAIN 0x23
#define WH_IMG_MGR_WOLFBOOT_HDR_PADDING 0xFF

/* wolfBoot auth key types (from high byte of img_type) */
#define WH_IMG_MGR_WOLFBOOT_AUTH_RSA4096 0x04
#define WH_IMG_MGR_WOLFBOOT_HDR_IMG_TYPE_AUTH_MASK 0xFF00

/* Image type enum controlling how the framework loads keys/signatures */
typedef enum {
    WH_IMG_MGR_IMG_TYPE_RAW = 0,       /* Key from keystore, sig from NVM */
    WH_IMG_MGR_IMG_TYPE_WOLFBOOT,      /* Key from keystore, sig from header */
    WH_IMG_MGR_IMG_TYPE_WOLFBOOT_CERT, /* Root CA from NVM, cert chain + sig
                                          from header */
} whServerImgMgrImgType;

/* Forward declaration for callback function signatures */
typedef struct whServerImgMgrContext_t whServerImgMgrContext;

/* Forward declare so the callbacks can reference the parent struct */
struct whServerImgMgrImg;

typedef int (*whServerImgMgrVerifyMethod)(whServerImgMgrContext* context,
                                          const struct whServerImgMgrImg* img,
                                          const uint8_t* key, size_t keySz,
                                          const uint8_t* sig, size_t sigSz);

typedef int (*whServerImgMgrVerifyAction)(whServerImgMgrContext* context,
                                          const struct whServerImgMgrImg* img,
                                          int verifyResult);

/*
 * @brief Image structure for verification
 *
 * This structure defines an image to be verified, including its location,
 * the key and signature identifiers, and the callbacks for verification
 * and post-verification actions.
 */
typedef struct whServerImgMgrImg {
    uintptr_t addr;     /* RAW: image blob. WOLFBOOT: firmware payload */
```

```
    size_t    size;     /* RAW: blob size. WOLFBOOT: max payload size, actual
                           size obtained from manifest header */
    uintptr_t hdrAddr;  /* wolfBoot header address (unused for RAW). Must be
                           2-byte aligned for WOLFBOOT/WOLFBOOT_CERT. */
    size_t    hdrSize;  /* wolfBoot header size (unused for RAW) */
    whKeyId   keyId;    /* RAW/WOLFBOOT: verify key ID. WOLFBOOT_CERT: unused */
    whNvmId   sigNvmId; /* RAW: sig NVM ID. WOLFBOOT_CERT: root CA NVM ID */
    whServerImgMgrImgType imgType; /* Controls framework loading behavior */
    whServerImgMgrVerifyMethod verifyMethod; /* Verification callback */
    whServerImgMgrVerifyAction verifyAction; /* Post-verification action */
} whServerImgMgrImg;

/*
 * @brief Image manager configuration structure
 *
 * Configuration structure passed to the image manager during initialization.
 * Contains the list of images to manage and their count.
 */
typedef struct whServerImgMgrConfig {
    whServerImgMgrImg* images;     /* Pointer to array of images */
    size_t             imageCount; /* Number of images in array */
    whServerContext*   server;     /* Server context for NVM/DMA access */
} whServerImgMgrConfig;

/*
 * @brief Image verification result structure
 *
 * Structure containing detailed results from image verification operations.
 * Provides separate access to verification method and action results.
 */
typedef struct whServerImgMgrVerifyResult {
    int verifyMethodResult; /* Result from the verification method callback */
    int verifyActionResult; /* Result from the verification action callback */
} whServerImgMgrVerifyResult;

/*
 * @brief Image manager context structure
 *
 * Context structure that maintains the state of the image manager.
 * This is embedded in the main server context.
 */
struct whServerImgMgrContext_t {
    whServerImgMgrImg images[WOLFHSM_CFG_SERVER_IMG_MGR_MAX_IMG_COUNT];
    size_t            imageCount;
    whServerContext*  server; /* Server context for NVM/DMA access */
};

int wh_Server_ImgMgrInit(whServerImgMgrContext*      context,
                         const whServerImgMgrConfig* config);

int wh_Server_ImgMgrVerifyAll(whServerImgMgrContext*      context,
                              whServerImgMgrVerifyResult* outResults,
                              size_t outResultsCount, size_t* outErrorIdx);
```

```
int wh_Server_ImgMgrVerifyImg(whServerImgMgrContext*      context,
                              const whServerImgMgrImg*    img,
                              whServerImgMgrVerifyResult* outResult);

int wh_Server_ImgMgrVerifyImgIdx(whServerImgMgrContext* context, size_t imgIdx,
                                 whServerImgMgrVerifyResult* outResult);

/* Built-in verification method callbacks */

int wh_Server_ImgMgrVerifyMethodEccWithSha256(whServerImgMgrContext*   context,
                                              const whServerImgMgrImg* img,
                                              const uint8_t* key, size_t keySz,
                                              const uint8_t* sig, size_t sigSz);

int wh_Server_ImgMgrVerifyMethodAesCmac(whServerImgMgrContext*   context,
                                        const whServerImgMgrImg* img,
                                        const uint8_t* key, size_t keySz,
                                        const uint8_t* sig, size_t sigSz);

int wh_Server_ImgMgrVerifyMethodRsaSslWithSha256(
    whServerImgMgrContext* context, const whServerImgMgrImg* img,
    const uint8_t* key, size_t keySz, const uint8_t* sig, size_t sigSz);

int wh_Server_ImgMgrVerifyMethodWolfBootRsa4096WithSha256(
    whServerImgMgrContext* context, const whServerImgMgrImg* img,
    const uint8_t* key, size_t keySz, const uint8_t* sig, size_t sigSz);

int wh_Server_ImgMgrVerifyMethodWolfBootCertChainRsa4096WithSha256(
    whServerImgMgrContext* context, const whServerImgMgrImg* img,
    const uint8_t* key, size_t keySz, const uint8_t* sig, size_t sigSz);

int wh_Server_ImgMgrVerifyActionDefault(whServerImgMgrContext*   context,
                                        const whServerImgMgrImg* img,
                                        int                      verifyResult);

#endif /* WOLFHSM_CFG_SERVER_IMG_MGR */

#endif /* !WOLFHSM_WH_SERVER_IMG_MGR_H_ */
```

B.4 wolfhsm/wh_server_cert.h

B.4.1 Functions

	Name
int	**wh_Server_CertInit**(whServerContext * server)Initialize the certificate manager.
int	**wh_Server_CertAddTrusted**(whServerContext * server, whNvmId id, whNvmAccess access, whNvmFlags flags, const uint8_t * label, whNvmSize label_len, const uint8_t * cert, uint32_t cert_len)Add a trusted certificate to NVM storage.

	Name
int	**wh_Server_CertEraseTrusted**(whServerContext * server, whNvmId id)Delete a trusted certificate from NVM storage.
int	**wh_Server_CertReadTrusted**(whServerContext * server, whNvmId id, uint8_t * cert, uint32_t * inout_cert_len)Get a trusted certificate from NVM storage.
int	**wh_Server_CertVerify**(whServerContext * server, const uint8_t * cert, uint32_t cert_len, whNvmId trustedRootNvmId, whCertFlags flags, whNvmFlags cachedKeyFlags, whKeyId * inout_keyId)Verify a certificate against trusted certificates.
int	**wh_Server_CertVerifyMultiRoot**(whServerContext * server, const uint8_t * cert, uint32_t cert_len, const whNvmId * trustedRootNvmIds, uint16_t numRoots, whCertFlags flags, whNvmFlags cachedKeyFlags, whKeyId * inout_keyId)Verify a certificate chain against a set of trusted root anchors.
int	**wh_Server_CertSetVerifyCb**(whServerContext * server, VerifyCallback cb)Register a verify callback at runtime.
int	**wh_Server_CertVerifyAcert**(whServerContext * server, const uint8_t * cert, uint32_t cert_len, whNvmId trustedRootNvmId)Verifies an attribute certificate against a trusted root certificate.
int	**wh_Server_HandleCertRequest**(whServerContext * server, uint16_t magic, uint16_t action, uint16_t seq, uint16_t req_size, const void * req_packet, uint16_t * out_resp_size, void * resp_packet)Handle a certificate request and generate a response.

B.4.2 Functions Documentation

B.4.2.1 function wh_Server_CertInit

```
int wh_Server_CertInit(
    whServerContext * server
)
```

Initialize the certificate manager.

Parameters:

- **server** The server context

Return: WH_ERROR_OK on success, error code on failure

B.4.2.2 function wh_Server_CertAddTrusted

```
int wh_Server_CertAddTrusted(
    whServerContext * server,
```

```
    whNvmId id,
    whNvmAccess access,
    whNvmFlags flags,
    const uint8_t * label,
    whNvmSize label_len,
    const uint8_t * cert,
    uint32_t cert_len
)
```

Add a trusted certificate to NVM storage.

Parameters:

- **server** The server context
- **id** The NVM ID to store the certificate under
- **cert** The certificate data buffer
- **cert_len** Length of the certificate data
- **flags** NVM flags for the certificate storage

Return: WH_ERROR_OK on success, error code on failure

B.4.2.3 function wh_Server_CertEraseTrusted

```
int wh_Server_CertEraseTrusted(
    whServerContext * server,
    whNvmId id
)
```

Delete a trusted certificate from NVM storage.

Parameters:

- **server** The server context
- **id** The NVM ID of the certificate to delete

Return: WH_ERROR_OK on success, error code on failure

B.4.2.4 function wh_Server_CertReadTrusted

```
int wh_Server_CertReadTrusted(
    whServerContext * server,
    whNvmId id,
    uint8_t * cert,
    uint32_t * inout_cert_len
)
```

Get a trusted certificate from NVM storage.

Parameters:

- **server** The server context
- **id** The NVM ID of the certificate to read
- **cert** Buffer to store the certificate data
- **inout_cert_len** On input, size of cert buffer. On output, actual cert size

Return: WH_ERROR_OK on success, error code on failure. If certificate is too large for the buffer, WH_ERROR_BUFFER_SIZE will be returned and inout_cert_len will be updated to the actual certificate size.

B.4.2.5 function wh_Server_CertVerify

```
int wh_Server_CertVerify(
    whServerContext * server,
    const uint8_t * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId,
    whCertFlags flags,
    whNvmFlags cachedKeyFlags,
    whKeyId * inout_keyId
)
```

Verify a certificate against trusted certificates.

Parameters:

- **server** The server context
- **cert** The certificate data to verify
- **cert_len** Length of the certificate data
- **trustedRootNvmId** NVM ID of the trusted root certificate
- **flags** Flags for the certificate verification (see WH_CERT_FLAGS_* in wh_common.h)
- **cachedKeyFlags** NVM usage flags to apply when caching the leaf public key (only used if WH_CERT_FLAGS_CACHE_LEAF_PUBKEY is set)
- **inout_keyId** Only valid if WH_CERT_FLAGS_CACHE_LEAF_PUBKEY is set. On input, set to the keyId to use when caching the leaf public key. If set to WH_KEYID_ERASED then a new unique keyId will be generated. On output, holds the keyId used for the leaf certificate.

Return: WH_ERROR_OK on success, error code on failure

B.4.2.6 function wh_Server_CertVerifyMultiRoot

```
int wh_Server_CertVerifyMultiRoot(
    whServerContext * server,
    const uint8_t * cert,
    uint32_t cert_len,
    const whNvmId * trustedRootNvmIds,
    uint16_t numRoots,
    whCertFlags flags,
    whNvmFlags cachedKeyFlags,
    whKeyId * inout_keyId
)
```

Verify a certificate chain against a set of trusted root anchors.

Parameters:

- **server** Server context.
- **cert** Candidate certificate chain (DER).
- **cert_len** Length of cert in bytes.
- **trustedRootNvmIds** Array of root NVM IDs to load as anchors. Order is informational only.
- **numRoots** Number of entries in trustedRootNvmIds. Must be 1..WOLFHSM_CFG_CERT_MAX_VERIFY_ROOTS.
- **flags** See WH_CERT_FLAGS_*.
- **cachedKeyFlags** NVM flags applied to the cached leaf key (only used if WH_CERT_FLAGS_CACHE_LEAF_PUBKEY).
- **inout_keyId** Cached leaf key id (only used if WH_CERT_FLAGS_CACHE_LEAF_PUBKEY).

Return: WH_ERROR_OK on chain trust success. WH_ERROR_CERT_VERIFY if no loaded anchor matches the chain. WH_ERROR_NOTFOUND if every supplied root id is absent from NVM. WH_ERROR_BADARGS / other negative codes on argument or environment errors.

Loads each available root identified by trustedRootNvmIds into a freshly allocated cert manager and verifies the supplied chain once. Succeeds if the chain anchors to any loaded root. Roots whose NVM objects are absent are skipped silently; non-absent failures to read or load any root are reported.

B.4.2.7 function wh_Server_CertSetVerifyCb

```
int wh_Server_CertSetVerifyCb(
    whServerContext * server,
    VerifyCallback cb
)
```

Register a verify callback at runtime.

Parameters:

- **server** The server context.
- **cb** The callback to register, or NULL to unregister.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if server is NULL.

Replaces the callback previously set via whServerCertConfig.verifyCb (or by a prior call to this function). Pass NULL to unregister.

The callback is applied to the per-request WOLFSSL_CERT_MANAGER created by wh_Server_CertVerify, so it participates in chain verification the same way a callback registered with wolfSSL_CertManagerSetVerify would. Verify-cache hits (when WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE is enabled) bypass the callback because they bypass wolfSSL's verify path entirely.

B.4.2.8 function wh_Server_CertVerifyAcert

```
int wh_Server_CertVerifyAcert(
    whServerContext * server,
    const uint8_t * cert,
    uint32_t cert_len,
    whNvmId trustedRootNvmId
)
```

Verifies an attribute certificate against a trusted root certificate.

Parameters:

- **server** Pointer to the server context
- **cert** Pointer to the attribute certificate data to verify
- **cert_len** Length of the certificate data in bytes
- **trustedRootNvmId** NVM ID of the trusted root certificate to verify against

Return: int Returns 0 on success, or a negative error code on failure.

This function retrieves a trusted root certificate from NVM using the specified NVM ID and verifies the provided attribute certificate against it.

B.4.2.9 function wh_Server_HandleCertRequest

```
int wh_Server_HandleCertRequest(
    whServerContext * server,
    uint16_t magic,
    uint16_t action,
    uint16_t seq,
    uint16_t req_size,
    const void * req_packet,
```

```
    uint16_t * out_resp_size,
    void * resp_packet
)
```

Handle a certificate request and generate a response.

Parameters:

- **server** The server context
- **magic** Magic number for message validation
- **action** The certificate action to perform
- **seq** Sequence number for the request
- **req_size** Size of the request packet
- **req_packet** The request packet data
- **out_resp_size** Size of the response packet
- **resp_packet** Buffer to store the response packet

Return: WH_ERROR_OK on success, error code on failure

B.4.3 Source code

```
/*
 * Copyright (C) 2025 wolfSSL Inc.
 *
 * This file is part of wolfHSM.
 *
 * wolfHSM is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
 * wolfHSM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with wolfHSM.  If not, see <http://www.gnu.org/licenses/>.
 */
/*
 * wolfhsm/wh_server_cert.h
 */

#ifndef WOLFHSM_WH_SERVER_CERT_H_
#define WOLFHSM_WH_SERVER_CERT_H_

/* Pick up compile-time configuration */
#include "wolfhsm/wh_settings.h"

#include <stdint.h>

#include "wolfhsm/wh_server.h"
#include "wolfhsm/wh_nvm.h"

int wh_Server_CertInit(whServerContext* server);
```

```
int wh_Server_CertAddTrusted(whServerContext* server, whNvmId id,
                             whNvmAccess access, whNvmFlags flags,
                             const uint8_t* label, whNvmSize label_len,
                             const uint8_t* cert, uint32_t cert_len);

int wh_Server_CertEraseTrusted(whServerContext* server, whNvmId id);

int wh_Server_CertReadTrusted(whServerContext* server, whNvmId id,
                              uint8_t* cert, uint32_t* inout_cert_len);

int wh_Server_CertVerify(whServerContext* server, const uint8_t* cert,
                         uint32_t cert_len, whNvmId trustedRootNvmId,
                         whCertFlags flags, whNvmFlags cachedKeyFlags,
                         whKeyId* inout_keyId);

int wh_Server_CertVerifyMultiRoot(whServerContext* server, const uint8_t* cert,
                                  uint32_t       cert_len,
                                  const whNvmId* trustedRootNvmIds,
                                  uint16_t numRoots, whCertFlags flags,
                                  whNvmFlags cachedKeyFlags,
                                  whKeyId*   inout_keyId);

#if defined(WOLFHSM_CFG_CERTIFICATE_MANAGER) && !defined(WOLFHSM_CFG_NO_CRYPTO)
int wh_Server_CertSetVerifyCb(whServerContext* server, VerifyCallback cb);
#endif /* WOLFHSM_CFG_CERTIFICATE_MANAGER && !WOLFHSM_CFG_NO_CRYPTO */

#if defined(WOLFHSM_CFG_CERTIFICATE_MANAGER_ACERT)
int wh_Server_CertVerifyAcert(whServerContext* server, const uint8_t* cert,
                              uint32_t cert_len, whNvmId trustedRootNvmId);
#endif

int wh_Server_HandleCertRequest(whServerContext* server, uint16_t magic,
                                uint16_t action, uint16_t seq,
                                uint16_t req_size, const void* req_packet,
                                uint16_t* out_resp_size, void* resp_packet);

#endif /* !WOLFHSM_WH_SERVER_CERT_H_ */
```

B.5 wolfhsm/wh_server_cert_cache.h

B.5.1 Classes

	Name
struct	**whCertVerifyCacheSlot**
struct	**whCertVerifyCacheContext**

B.5.2 Types

Name
typedef struct whCertVerifyCacheSlot**

Name
typedef struct whCertVerifyCacheContext**

B.5.3 Functions

	Name
int	**wh_Server_CertVerifyCache_Lookup**(struct whServerContext_t * server, const whNvmId * rootNvmIds, uint16_t numRoots, const uint8_t * hash)Look up a cert hash in the verify cache against a set of loaded trusted roots.
void	**wh_Server_CertVerifyCache_Insert**(struct whServerContext_t * server, const whNvmId * rootNvmIds, uint16_t numRoots, const uint8_t * hash)Insert a cert hash into the verify cache, recording the supplied root set as the entry's binding.
int	**wh_Server_CertVerifyCache_Clear**(struct whServerContext_t * server)Clear all entries from the verify cache.
int	**wh_Server_CertVerifyCache_SetEnabled**(struct whServerContext_t * server, uint8_t enable)Enable or disable the trusted cert verify cache at runtime.
int	**wh_Server_CertVerifyCache_EvictRoot**(struct whServerContext_t * server, whNvmId rootNvmId)Evict every cache entry whose stored root set contains the supplied trusted-root NVM ID.

B.5.4 Types Documentation

B.5.4.1 typedef whCertVerifyCacheSlot

```
typedef struct whCertVerifyCacheSlot whCertVerifyCacheSlot;
```

B.5.4.2 typedef whCertVerifyCacheContext

```
typedef struct whCertVerifyCacheContext whCertVerifyCacheContext;
```

B.5.5 Functions Documentation

B.5.5.1 function wh_Server_CertVerifyCache_Lookup

```
int wh_Server_CertVerifyCache_Lookup(
    struct whServerContext_t * server,
    const whNvmId * rootNvmIds,
    uint16_t numRoots,
    const uint8_t * hash
)
```

Look up a cert hash in the verify cache against a set of loaded trusted roots.

Parameters:

- **server** The server context.
- **rootNvmIds** Array of trusted root NVM IDs currently loaded (presented set).
- **numRoots** Number of entries in rootNvmIds (must be > 0).
- **hash** Pointer to a SHA-256 (32-byte) digest of the DER cert.

Return: WH_ERROR_OK on hit, WH_ERROR_NOTFOUND on miss, WH_ERROR_BADARGS on invalid arguments.

Hits when there exists a committed slot whose stored root set is a subset of the supplied set and whose hash matches. By verify monotonicity, a previously successful verify under the slot's set remains valid under any superset, so the hit is sound.

B.5.5.2 function wh_Server_CertVerifyCache_Insert

```
void wh_Server_CertVerifyCache_Insert(
    struct whServerContext_t * server,
    const whNvmId * rootNvmIds,
    uint16_t numRoots,
    const uint8_t * hash
)
```

Insert a cert hash into the verify cache, recording the supplied root set as the entry's binding.

Parameters:

- **server** The server context.
- **rootNvmIds** Array of trusted root NVM IDs loaded for the verify.
- **numRoots** Number of entries in rootNvmIds (must be > 0 and <= WOLFHSM_CFG_CERT_MAX_VERIFY_ROOTS).
- **hash** Pointer to a SHA-256 (32-byte) digest of the DER cert.

No-op if a slot with the same hash and the same root set already exists. Uses FIFO ring overwrite when full.

The supplied set must be the set of roots actually loaded into the cert manager at the time of the verify (post-filtering of any roots absent from NVM); recording roots that were not actually loaded would widen the entry's required-trust set without justification.

B.5.5.3 function wh_Server_CertVerifyCache_Clear

```
int wh_Server_CertVerifyCache_Clear(
    struct whServerContext_t * server
)
```

Clear all entries from the verify cache.

Parameters:

- **server** The server context.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if server is invalid, or a lock error if the cache lock could not be acquired.

In per-client mode (default) clears this server's cache only. In global mode (WOLFHSM_CFG_CERTIFICATE_VERIFY_CACH clears the shared cache for every connected client.

B.5.5.4 function wh_Server_CertVerifyCache_SetEnabled

```
int wh_Server_CertVerifyCache_SetEnabled(
    struct whServerContext_t * server,
    uint8_t enable
)
```

Enable or disable the trusted cert verify cache at runtime.

Parameters:

- **server** The server context.
- **enable** 1 to enable caching, 0 to disable (and flush).

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if server is invalid.

When disabled, all existing entries are cleared and subsequent calls to Lookup miss / Insert are no-ops until the cache is re-enabled. Enabling an already-enabled cache (or disabling an already-disabled one) is a no-op aside from acquiring the lock.

In per-client mode (default) this affects this server's cache only. In global mode (WOLFHSM_CFG_CERTIFICATE_VERIFY_C this affects the shared cache observed by every connected client.

B.5.5.5 function wh_Server_CertVerifyCache_EvictRoot

```
int wh_Server_CertVerifyCache_EvictRoot(
    struct whServerContext_t * server,
    whNvmId rootNvmId
)
```

Evict every cache entry whose stored root set contains the supplied trusted-root NVM ID.

Parameters:

- **server** The server context.
- **rootNvmId** NVM ID of the trusted root whose cache entries to drop.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if server is invalid, or a lock error if the cache lock could not be acquired (in which case the caller must treat the cache state as suspect).

Must be invoked whenever the trusted root at rootNvmId changes (add or erase). Without this, re-using a freed ID for a different root would let stale cache hits short-circuit verifies under a trust anchor that is no longer present at that ID.

Entries whose stored set contains the evicted root are dropped entirely rather than stripped of that one root, because the original verify may have been anchored at the now-departed root.

B.5.6 Source code

```
/*
 * Copyright (C) 2025 wolfSSL Inc.
 *
 * This file is part of wolfHSM.
 *
 * wolfHSM is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or
 * (at your option) any later version.
 *
 * wolfHSM is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with wolfHSM.  If not, see <http://www.gnu.org/licenses/>.
 */

/*
 * wolfhsm/wh_server_cert_cache.h
 *
 * Server-side cert subsystem types embedded in whServerContext:
 *   - whServerCertContext / whServerCertConfig: hold the user-injectable
 *     verify callback and (optionally) the trusted-cert verify cache.
 *   - whCertVerifyCacheContext: trusted-cert verify-result cache. Records
 *     SHA-256 hashes of DER-encoded CA certificates that have already been
 *     successfully verified, scoped to the set of trusted-root NVM IDs
 *     that were loaded when the verify ran. Hits apply across clients
 *     but require the cached root set to be a subset of the caller's
 *     currently-loaded root set.
 *
 *     Only CA certs are inserted. Caching a leaf would let a future
 *     "leaf alone" verify falsely succeed via cache hit, because the
 *     cache hit bypasses the wolfSSL signature check that would otherwise
 *     have failed (the leaf's issuer is not in the cert manager when the
 *     leaf is supplied without its intermediates). CA caching is sound
 *     because the chain walk loads each verified CA into the cert manager
 *     before the next cert is processed.
 *
 *     Soundness of the subset rule rests on X.509 verify monotonicity:
 *     adding more trusted roots can never invalidate a previously
 *     successful verify, so a chain that validated under set S still
 *     validates under any superset T ⊇ S. A cache hit therefore implies
 *     the cached verify's anchor (whichever root in S actually closed
 *     the chain) is currently trusted, regardless of which element of S
 *     it was — every element of S is in T by hypothesis.
 *
 *     Both single-root and multi-root verifies populate the cache.
 *     Single-root entries have one-element sets (maximum reuse, since
 *     any later caller whose loaded set contains that root will hit).
 *     Multi-root entries have larger sets (narrower reuse — only later
 *     callers whose loaded set is a superset will hit) but capture
 *     verifies that pure single-root traffic would not generate.
 *
 * Lives in its own header to avoid circular dependencies between wh_server.h
 * and wh_server_cert.h.
 */

#ifndef WOLFHSM_WH_SERVER_CERT_CACHE_H_
#define WOLFHSM_WH_SERVER_CERT_CACHE_H_

/* Pick up compile-time configuration */
#include "wolfhsm/wh_settings.h"

#if defined(WOLFHSM_CFG_CERTIFICATE_MANAGER) && !defined(WOLFHSM_CFG_NO_CRYPTO)
```

```c
#include <stdint.h>

#include "wolfhsm/wh_common.h" /* for whNvmId */
#include "wolfhsm/wh_lock.h"   /* for whLock (global cache lock) */

#include "wolfssl/ssl.h" /* for VerifyCallback */

#ifdef WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE

#ifndef WOLFHSM_CFG_CERT_VERIFY_CACHE_COUNT
#define WOLFHSM_CFG_CERT_VERIFY_CACHE_COUNT 16
#endif

#define WH_CERT_VERIFY_CACHE_HASH_LEN 32 /* SHA-256 digest size */

typedef struct whCertVerifyCacheSlot {
    uint8_t  committed; /* 0 = empty, 1 = valid */
    uint8_t  WH_PAD[1];
    uint16_t numRoots; /* count of valid entries in rootNvmIds */
    /* Set of trusted root NVM IDs loaded when this cert was verified. A
     * lookup hits when this set is a subset of the caller's currently
     * loaded set (verify monotonicity makes the over-approximation safe). */
    whNvmId rootNvmIds[WOLFHSM_CFG_CERT_MAX_VERIFY_ROOTS];
    uint8_t hash[WH_CERT_VERIFY_CACHE_HASH_LEN];
} whCertVerifyCacheSlot;

typedef struct whCertVerifyCacheContext {
    whCertVerifyCacheSlot slots[WOLFHSM_CFG_CERT_VERIFY_CACHE_COUNT];
    uint16_t              writeIdx; /* FIFO ring write position */
    /* Runtime enable flag. When zero, Lookup misses and Insert is a no-op,
     * regardless of slot contents. Toggled by
     * wh_Server_CertVerifyCache_SetEnabled (also reachable from clients via
     * WH_MESSAGE_CERT_ACTION_VERIFY_CACHE_SET_ENABLED). Default is 1;
     * explicitly initialized at server / NVM init so a fresh zero-init context
     * is treated as disabled until init runs. */
    uint8_t enabled;
    uint8_t WH_PAD[5];
#if defined(WOLFHSM_CFG_THREADSAFE) && \
    defined(WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE_GLOBAL)
    /* Dedicated lock for the global verify cache. Independent from the NVM
     * lock so cert-cache operations do not serialize behind NVM I/O. Only
     * present when the cache lives in the shared NVM context. */
    whLock lock;
#endif
} whCertVerifyCacheContext;

#endif /* WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE */

/* Per-server cert subsystem config, supplied via whServerConfig.certConfig.
 * The verify callback signature matches wolfSSL's VerifyCallback so the same
 * callback registered with wolfSSL_CertManagerSetVerify can be used here. */
typedef struct {
    VerifyCallback verifyCb; /* user-supplied; NULL = no callback */
```

```
} whServerCertConfig;

/* Per-server cert subsystem context, embedded by value in whServerContext.
 * Holds the registered verify callback and (optionally) the per-client verify
 * cache. When WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE_GLOBAL is defined the cache
 * is relocated into the shared whNvmContext, so the per-client copy is
 * omitted. */
typedef struct {
    VerifyCallback verifyCb;
#if defined(WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE) && \
    !defined(WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE_GLOBAL)
    whCertVerifyCacheContext cache;
#endif
} whServerCertContext;

/* Forward declaration to avoid pulling in wh_server.h */
struct whServerContext_t;

#ifdef WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE
int wh_Server_CertVerifyCache_Lookup(struct whServerContext_t* server,
                                     const whNvmId*            rootNvmIds,
                                     uint16_t numRoots, const uint8_t* hash);

void wh_Server_CertVerifyCache_Insert(struct whServerContext_t* server,
                                      const whNvmId*            rootNvmIds,
                                      uint16_t numRoots, const uint8_t* hash);

int wh_Server_CertVerifyCache_Clear(struct whServerContext_t* server);

int wh_Server_CertVerifyCache_SetEnabled(struct whServerContext_t* server,
                                         uint8_t                   enable);

int wh_Server_CertVerifyCache_EvictRoot(struct whServerContext_t* server,
                                        whNvmId                   rootNvmId);
#endif /* WOLFHSM_CFG_CERTIFICATE_VERIFY_CACHE */

#endif /* WOLFHSM_CFG_CERTIFICATE_MANAGER && !WOLFHSM_CFG_NO_CRYPTO */

#endif /* !WOLFHSM_WH_SERVER_CERT_CACHE_H_ */
```