wolfHSM Documentation

2026-01-19

Contents

0.1 Introduction

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

You can find the PDF version of this document here.

0.1.1 Why Choose wolfHSM?

Automotive HSMs (Hardware Security Modules) dramatically improve the security of cryptographic keys and processing. They achieve this by isolating signature verification and cryptographic execution, the very foundations of security, into physically independent processors. These HSMs are not just recommended, but often mandatory for ECUs that demand robust security. In line with this, wolfSSL has seamlessly integrated our popular, rigorously tested, and industry-leading cryptographic library to operate in widely used Automotive HSMs such as Aurix Tricore TC3XX. WolfHSM, with its sole dependency on wolfCrypt, ensures portability across almost any runtime environment. It also facilitates a user-friendly client interface, allowing direct utilization of wolfCrypt APIs.

wolfHSM provides a portable and open-source abstraction to hardware cryptography, non-volatile memory, and isolated secure processing, maximizing security and performance for ECUs. By integrating the wolfCrypt software crypto engine on hardware HSMs like Infineon Aurix Tricore TC3XX, Chinese-mandated government algorithms like SM2, SM3, and SM4 are available. Additionally, Post Quantum Cryptography algos like Kyber, LMS, XMSS, and others are easily made available to automotive users to meet customer requirements. At the same time, when hardware cryptographic processing is available on the HSM, we consume it to enhance performance.

wolfBoot is a mature, portable, secure bootloader solution designed for bare-metal bootloaders and equipped with failsafe NVM controls. It offers comprehensive firmware authentication and update mechanisms, leveraging a minimalistic design and a tiny HAL API, which makes it fully independent from any operating system or bare-metal application. wolfBoot efficiently manages the flash interface and pre-boot environment, accurately measures and authenticates applications, and utilizes low-level hardware cryptography as needed. wolfBoot can use the wolfHSM client to support HSM-assisted application core secure boot. Additionally, wolfBoot can run on the HSM core to ensure the HSM server is intact, offering a secondary layer of protection. This setup ensures a secure boot sequence, aligning well with the booting processes of HSM cores that rely on NVM support.

0.2 Overview

wolfHSM is a software framework that provides a unified API for HSM operations such as cryptographic operations, key management, and non-volatile storage. It is designed to improve portability of code related to HSM applications, easing the challenge of moving between hardware with enhanced security features without being tied to any vendor-specific library calls. It dramatically simplifies client

applications by allowing direct use of wolfCrypt APIs, with the library automatically offloading all sensitive cryptographic operations to the HSM core as remote procedure calls with no additional logic required by the client app.

Although initially targeted to automotive-style HSM-enabled microcontrollers, wolfHSM provides an extensible solution to support future capabilities of platforms while still supporting standardized interfaces and protocols such as PKCS11 and AUTOSAR SHE. It has no external dependencies other than wolfCrypt and is portable to almost any runtime environment.

0.2.1 Features

- Secure non-volatile object storage with user-based permissions
- Cryptographic key management with support for hardware keys
- Hardware cryptographic support for compatible devices
- Fully asynchronous client API
- Flexible callback architecture enables custom use cases without modifying library
- Use wolfCrypt APIs directly on client, with automatic offload to HSM core
- Image manager to support chain of trust
- Integration with AUTOSAR
- Integration with SHE+
- PKCS11 interface available
- TPM 2.0 interface available
- Secure OnBoard Communication (SecOC) module integration available
- Certificate handling
- Symmetric and Asymmetric keys and cryptography
- Supports “crypto agility” by providing every algorithm implemented in wolfCrypt, not just those implemented by your silicon vendor
- FIPS 140-3 available

0.2.2 Architecture

wolfHSM employs a client-server architecture where the server runs in a trusted and secure environment (typically on a secure coprocessor) and the client is a library that can be linked against user applications. This architecture ensures that sensitive cryptographic operations and key management are handled securely within the server, while the client library abstracts away the lower level communication with the server.

- Server: The server component of wolfHSM is a standalone application that runs on the HSM core. It handles cryptographic operations, key management, and non-volatile storage within a secure environment. The server is responsible for processing requests from clients and returning the results.
- Client: The client component of wolfHSM is a library that can be linked against user applications. It provides APIs for sending requests to the server and receiving responses. The client abstracts the complexities of communication and ensures that the application can interact with the HSM securely and efficiently.

0.2.3 Ports

wolfHSM itself is not executable and it does not contain any code to interact with any specific hardware. In order for wolfHSM to run on a specific device, the library must be configured with the necessary hardware drivers and abstraction layers so that the server application can run and communicate with the client. Specifically, getting wolfHSM to run on real hardware requires the implementation of the following:

- Server application startup and hardware initialization

- Server wolfCrypt configuration
- Server non-volatile memory configuration
- Server and client transport configuration
- Server and client connection handling

The code that provides these requirements and wraps the server API into a bootable application is collectively referred to as a wolfHSM "port".

Official ports of wolfHSM are provided for various supported architectures, with each port providing the implementation of the wolfHSM abstractions tailored to the specific device. Each port contains:

- Standalone Reference Server Application: This application is meant to run on the HSM core and handle all secure operations. It comes fully functional out-of-the-box but can also be customized by the end user to support additional use cases
- Client Library: This library can be linked against user applications to facilitate communication with the server

0.2.3.1 Supported Ports

wolfHSM has supported ports for the following devices/environments:

- POSIX runtime
- ST Micro SPC58N*
- Infineon Aurix TC3xx*
- Infineon Aurix TC4xx* (coming soon)
- Infineon Traveo T2G* (coming soon)
- Renesas RH850* (coming soon)
- Renesas RL78* (coming soon)
- NXP S32* (coming soon)

With additional ports on the way.

- These ports, unfortunately, require an NDA with the silicon vendor to obtain any information about the HSM core. Therefore, the wolfHSM ports for these platforms are not public and are only available to qualified customers. If you wish to access a restricted wolfHSM port, please contact us at support@wolfssl.com.

0.3 Getting Started With wolfHSM

The most common use case for wolfHSM is adding HSM-enabled functionality to an existing application that runs on one of the application cores of a multi-core device with an HSM coprocessor.

The first step required to run wolfHSM on a device is to follow the steps in the specific wolfHSM port to get the reference server running on the HSM core. Once the wolfHSM server app is loaded on the device and boots, client applications can link against the wolfHSM client library, configure an instance of the wolfHSM client structure, and interact with the HSM through the wolfHSM client API and through the wolfCrypt API.

Each wolfHSM port contains a client demo app showing how to set up the default communication channel and interact with the server. The server reference implementation can also be customized through server callbacks to extend its functionality, which can be invoked through client requests.

0.3.1 Basic Client Configuration

Configuring a wolfHSM client involves allocating a client context structure and initializing it with a valid client configuration that enables it to communicate with a server.

The client context structure `whClientContext` holds the internal state of the client and its communication with the server. All client APIs take a pointer to the client context.

The client configuration structure holds the communication layer configuration (whCommClientConfig) that will be used to configure and initialize the context for the server communication. The whCommClientConfig structure binds an actual transport implementation (either built-in or custom) to the abstract comms interface for the client to use.

The general steps to configure a client are:

1. Allocate and initialize a transport configuration structure, context, and callback implementation for the desired transport
2. Allocate comm client configuration structure and bind it to the transport configuration from step 1 so it can be used by the client
3. Allocate and initialize a client configuration structure using the comm client configuration in step 2
4. Allocate a client context structure
5. Initialize the client with the client configuration by calling wh_Client_Init()
6. Use the client APIs to interact with the server

Here is a bare-minimum example of configuring a client application to use the built-in shared memory transport to send an echo request to the server.

```
#include <string.h> /* for memcmp() */
#include "wolfhsm/client.h"  /* Client API (includes comm config) */
#include "wolfhsm/wh_transport_mem.h" /* transport implementation */

/* Step 1: Allocate and initialize the shared memory transport configuration */
/* Shared memory transport configuration */
static whTransportMemConfig transportMemCfg = { /* shared memory config */ };
/* Shared memory transport context (state) */
whTransportMemClientContext transportMemClientCtx = {0};
/* Callback structure that binds the abstract comm transport interface to
 * our concrete implementation */
whTransportClientCb transportMemClientCb = {WH_TRANSPORT_MEM_CLIENT_CB};

/* Step 2: Allocate client comm configuration and bind to the transport */
/* Configure the client comms to use the selected transport configuration */
whCommClientConfig commClientCfg = {
             .transport_cb      = transportMemClientCb,
             .transport_context = (void*)transportMemClientCtx,
             .transport_config  = (void*)transportMemCfg,
             .client_id         = 123, /* unique client identifier */
};

/* Step 3: Allocate and initialize the client configuration */
whClientConfig clientCfg= {
   .comm = commClientCfg,
};

/* Step 4: Allocate the client context */
whClientContext clientCtx = {0};

/* Step 5: Initialize the client with the provided configuration */
wh_Client_Init(&clientCtx, &clientCfg);

/* Step 6: Use the client APIs to interact with the server */

/* Buffers to hold sent and received data */
```



```
char recvBuffer[WH_COMM_DATA_LEN] = {0};
char sendBuffer[WH_COMM_DATA_LEN] = {0};

uint16_t sendLen = snprintf(&sendBuffer,
                            sizeof(sendBuffer),
                            "Hello World!\n");
uint16_t recvLen = 0;

/* Send an echo request and block on receiving a response */
wh_Client_Echo(client, sendLen, &sendBuffer, &recvLen, &recvBuffer);

if ((recvLen != sendLen ) ||
    (0 != memcmp(sendBuffer, recvBuffer, sendLen))) {
    /* Error, we weren't echoed back what we sent */
}
```

For more information, refer to Chapter 5: Client Library.

0.3.2 Basic Server Configuration

Note: A wolfHSM port comes with a reference server application that is already configured to run on the HSM core and so manual server configuration is not required.

Configuring a wolfHSM server involves allocating a server context structure and initializing it with a valid client configuration that enables it to perform the requested operations. These operations usually include client communication, cryptographic operations, managing keys, and non-volatile object storage. Depending on the required functionality, not all of these configuration components need to be initialized.

The steps required to configure a server that supports client communication, NVM object storage using the NVM flash configuration, and local crypto (software only) are:

1. Initialize the server comms configuration 1. Allocate and initialize a transport configuration structure, context, and callback implementation for the desired transport 2. Allocate and initialize a comm server configuration structure using the transport configuration from step 1.1
2. Initialize the server NVM context 1. Allocate and initialize a config, context, and callback structure for the low-level flash storage drivers (the implementation of these structures is provided by the port) 2. Allocate and initialize an NVM flash config, context, and callback strucure and bind the port flash configuration from step 2.1 to them 3. Allocate an NVM context structure and initialize it with the configuration from step 2.2 using `wh_Nvm_Init()`
3. Allocate and initialize a crypto context structure for the server
4. Initialize wolfCrypt (before initializing the server)
5. Allocate and initialize a server config structure and bind the comm server configuration, NVM context, and crypto context to it
6. Allocate a server context structure and initialize it with the server configuration using `wh_Server_Init()`
7. Set the server connection state to connected using `wh_Server_SetConnected()` when the underlying transport is ready to be used for client communication (see wolfHSM Examples for more information)
8. Process client requests using `wh_Server_HandleRequestMessage()`

The server may be configured to support NVM object storage using NVM flash configuration. Include the steps to initialize NVM on the server after step 1.

```
#include <string.h> /* for memcmp() */
#include "wolfhsm/server.h"  /* Server API (includes comm config) */
#include "wolfhsm/wh_transport_mem.h" /* transport implementation */
```

```c
/* Step 1.1: Allocate and initialize the shared memory transport configuration
 ↪  */
/* Shared memory transport configuration */
static whTransportMemConfig transportMemCfg = { /* shared memory config */ };

/* Shared memory transport context (state) */
whTransportMemServerContext transportMemServerCtx = {0};

/* Callback structure that binds the abstract comm transport interface to
 * our concrete implementation */
whTransportServerCb transportMemServerCb = {WH_TRANSPORT_MEM_SERVER_CB};

/* Step 1.2: Allocate a comm server configuration structure and bind to the
 * transport */
/* Configure the server comms to use the selected transport configuration*/
whCommServerConfig commServerCfg = {
        .transport_cb       = transportMemServerCb,
        .transport_context  = (void*)transportMemServerCtx,
        .transport_config   = (void*)transportMemCfg,
        .server_id          = 456, /* unique server identifier */
};

/* Initialize the server NVM context */

/* Step 2.1: Allocate and initialize context and config for port-specific
 * flash storage drivers */

/* Port Flash context (structure names are port-specific)  */
MyPortFlashContext portFlashCtx = {0}

/* Port Flash config */
MyPortFlashConfig portFlashCfg = { /* port specific configuration */ };

/* NVM Flash callback implementation for Port Flash */
whFlashCb portFlashCb = { /* port flash implementation of NVM Flash callbacks */

/* Step 2.2: Allocate and initialize NVM flash config structure and bind to port
 * configuration from step 2.1 */
whNvmFlashConfig nvmFlashCfg = {
    .cb         = portFlashCb,
    .context    = portFlashCtx,
    .config     = portFlashCfg,
};
whNvmFlashContext nfc = {0};

/* Step 2.3: Allocate NVM context, config, and callback structure and
 ↪  initialize
 * with NVM flash configuration from step 2.2 */
whNvmCb nvmFlashCb = {WH_NVM_FLASH_CB};

whNvmConfig nvmConf = {
    .cb      = nvmFlashCb;
    .context = nfc;
```

```
    .config  = nvmFlashCfg,
};
whNvmContext nvmCtx = {0};

wh_Nvm_Init(&nvmCtx, &whNvmConfig);

/* Step 3: Allocate and initialize a crypto context structure */
whServerCryptoContext cryptoCtx {
    .devID = INVALID_DEVID; /* or set to custom crypto callback devID */
};

/* Allocate and initialize the Server configuration*/
whServerConfig serverCfg = {
        .comm   = commServerCfg,
        .nvm    = nvmCtx,
        .crypto = cryptoCtx,
};

/* Step 4: Initialize wolfCrypt*/
wolfCrypt_Init();

/* Step 5: Allocate and initialize server config structure and bind the comm
 * server configuration and crypto context to it*/
whServerContext server = {0};
wh_Server_Init(&server, &serverCfg);

/* Set server connection state to connected when transport is ready (e.g.
 * shared memory buffers cleared) */
wh_Server_SetConnected(&server, WH_COMM_CONNECTED);

/* Process client requests*/
while (1) {
    wh_Server_HandleRequestMessage(&server);
}
```

0.4 Library Design / wolfHSM Internals

wolfHSM is a modular and extensible library designed to provide a secure and efficient hardware security module (HSM) API for embedded systems. The library is built around a set of functional components that can be easily configured and combined to meet the specific requirements of a given application. This chapter provides an overview of the key functional components of wolfHSM, including the component architecture, communications layer, non-volatile memory (NVM), key management, cryptographic operations, and hardware security module (HSM) support.

0.4.1 Table of Contents:

0.4.2 Generic Component Architecture

To support easily porting wolfHSM to different hardware platforms and build environments, each component of wolfHSM is designed to have a common initialization, configuration, and context storage architecture to allow compile-time, link- time, and/or run-time selection of functional components. Hardware specifics are abstracted from the logical operations by associating callback functions with untyped context structures, referenced as a `void*`.

0.4.2.1 Example component initialization The prototypical compile-time static instance configuration and initialization sequence of a wolfHSM component is:

```
#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 for function callbacks for mycomponent. Note the
 ↪  type
is the abstract type provided in wolfhsm/component.h */
whComponentCb my_cb[1] = {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[1] = {0};

/* Initialization of the component using platform-specific callbacks */
const whComponentConfig comp_config[1] = {
        .cb = my_cb,
        .context = my_context,
        .config = my_config
    };
whComponentContext comp_context[1] = {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);
```

0.4.3 Communications

The communication layer of wolfHSM is designed to provide reliable, bidirectional, and packet-based communication between clients and servers. This layer abstracts the underlying transport mechanisms, allowing for flexibility and modularity. A key aspect of wolfHSM communication is split request and response functions for both client and server, enabling synchronous polling of message reception or asynchronous handling based on interrupt/event support.

0.4.3.1 Key Components

- Client/Server APIs: Main interface for communicating between client and server. These are the APIs that are directly used by user applications.
- Comms layer: Defines the format and structure of messages exchanged between clients and servers, and provides an abstract interface to the underlying transport layer implementation, exposing a consistent interface for sending and receiving messages.
- Transport Layer: Concrete implementations of the underlying transport. Defines how data is actually transported between client and server.

0.4.3.2 Client/Server APIs High-level client and server APIs (defined in `wolfhsm/wh_client.h` and `wolfhsm/wh_server.h`) are the primary interface for communication. These functions abstract the low level communications details from the caller, providing a simple split transaction interface for logical operations.

For example, using the client API to send an echo request to the server:

```
/* send the echo request */
wh_Client_EchoRequest(&clientCtx, sendLen, &sendBuffer));

/* optionally do stuff */

/* poll for the server response */
while (WH_ERROR_NOTREADY == wh_Client_EchoResponse(client, &recv_len,
 ↪  recv_buffer));
```

0.4.3.3 Comms Layer The comms layer encapsulates the messaging structure and control logic to send and receive data from lower level transports. The comms layer is directly invoked by the higher level client and server APIs. The comms layer provides comm client and comm server abstractions that hold communication state and provide the abstract interface functions to interact with lower level transports. The comms layer API consists of send and receive functions for requests and responses, where the requests and responses pertain to messages rather than high level operations.

Each client is only allowed a single outstanding request to the server at a time. The server will process a single request at a time to ensure client isolation.

0.4.3.3.1 Messages Messages comprise a header with a variable length payload. The header indicates the sequence id, and type of a request or response. The header also provides additional fields to provide auxiliary flags or session information.

```
/* wolfhsm/wh_comm.h */

typedef struct {
    uint16_t magic;
    uint16_t kind;
    uint16_t seq;
```

```
    uint16_t size;
} whCommHeader;
```

Messages are used to encapsulate the request data necessary for the server to execute the desired function and for the response to provide the results of the function execution back to the client. Message types are grouped based on the component that is performing the function and uniquely identify which of the enumerated functions is being performed. To ensure compatibility (endianness and version), messages include a Magic field which has known values used to indicate what operations are necessary to demarshall data passed within the payload for native processing. Each functional component has a "remote" implementation that converts between native values and the "on-the-wire" message formats. The servers ensures the response format matches the request format.

In addition to passing data contents within messages, certain message types also support passing shared or mapped memory pointers, especially for performance- critical operations where the server component may be able to directly access the data in a DMA fashion. To avoid integer pointer size (IPS) and `size_t` differences, all pointers and sizes should be sent as `uint64_t` when possible.

Messages are encoded in the "on-the-wire" format using the Magic field of the header indicating the specified endianness of structure members as well as the version of the communications header (currently 0x01). Server components that process request messages translate the provided values into native format, perform the task, and then reencode the result into the format of the request. Client response handling is not required to process messages that do not match the request format. Encoded messages assume the same size and layout as the native structure, with the endianness specified by the Magic field.

Here is an example of how the client comm layer sends a request:

```
uint16_t req_magic = wh_COMM_MAGIC_NATIVE;
uint16_t req_type = 123;
uint16_t request_id;
char* req_data = "RequestData";
rc = wh_CommClient_SendRequest(context, req_magic, req_type, &request_id,
                    sizeof(req_data), req_data);
/* Do other work */

uint16_t resp_magic, resp_type, resp_id, resp_size;
char response_data[20];
while((rc = wh_CommClient_RecvResponse(context,&resp_magic, &resp_type,
 ↪  &resp_id,
                    &resp_size, resp_data)) == WH_ERROR_NOTREADY) {
        /* Do other work or yield */
}
```

Note that transport errors passed into the message layer are expected to be fatal and the client/server should Cleanup any context as a result.

0.4.3.4 Transports Transports provide intact packets (byte sequences) of variable size (up to a maximum MTU), to the messaging layer for the library to process as a request or response. Transports implement the abstract interface defined by `whTransportClientCb` and are invoked directly by the commClient/commServer when needing to send and receive data.

Custom transport modules that implement the `whTransportClientCb` interface can be registered with the server and client and then are automatically used via the standard server and client request/response functions.

Examples of a memory buffer transport module and a POSIX TCP socket transport can be found in wolfHSM's supported transports.

0.4.3.4.1 Supported Transports wolfHSM ships with two built-in transports: a memory buffer transport (`wh_transport_mem.c`) and a POSIX TCP socket transport (`port/posix_transport_tcp.c`).

The memory transport is the default transport for most embedded wolfHSM ports, and is part of the core wolfHSM library. It provides a concrete implementation of the transport callbacks using shared memory blocks between client and server. The shared memory transport mechanism works by allocating two blocks of memory, one for incoming requests and one for outgoing responses. The client writes requests to the incoming memory block and reads responses from the outgoing memory block. The server reads requests from the incoming memory block and writes responses to the outgoing memory block. Each block contains control and status flags signaling to the consumer when it is ready for use. This mechanism is designed to be fast and efficient, as it avoids the need for system calls or network communication.

The POSIX TCP transport is part of the wolfHSM POSIX port. It uses TCP sockets as the transport medium for data between client and server. The sockets are IPv4 only and non-blocking.

0.4.4 Non Volatile Memory

Non-Volatile Memory (NVM) in the context of wolfHSM is used to manage persistent objects with metadata and data blocks. The NVM library ensures reliable, atomic operations to ensure transactions are fully committed before returning success. Key operations include adding, listing, reading, and destroying objects, as well as obtaining associated metadata.

High level NVM features include:

- API's to associate metadata (ID, Label, Length, Access, Flags) with variable-sized data within accessible NVM
- Always recoverable using 2 erasable partitions with status flags
- Objects are added by using the next entry and programmed into free space
- Duplicated id's are allowed but only the latest is readable
- Objects are destroyed by copying the entire space to the inactive partition without the listed objects
- Internal epoch counters used to identify the later objects during recovery

0.4.4.1 NVM Metadata In the wolfHSM library, Non-Volatile Memory (NVM) metadata is used to manage and describe objects stored in NVM. This metadata provides essential information about each object, such as its identifier, access permissions, flags, and other attributes. The metadata ensures that objects can be reliably managed, accessed, and manipulated within the NVM.

```
/* User-specified metadata for an NVM object */
typedef struct {
    whNvmId id;             /* Unique identifier */
    whNvmAccess access;     /* Access Permissions */
    whNvmFlags flags;       /* Additional flags */
    whNvmSize len;          /* Length of data in bytes */
    uint8_t label[WOLFHSM_NVM_LABEL_LEN];
} whNvmMetadata;
```

- ID (`whNvmId id`): A unique identifier for the NVM object. This ID is used to reference and access the specific object within the NVM. It allows for operations like reading, writing, and deleting the object.
- Access (`whNvmAccess access`): Defines the access permissions for the object. This field specifies who can access the object and under what conditions. It helps enforce security policies and ensures that only authorized entities can interact with the object.
- Flags (`whNvmFlags flags`): Additional flags that provide extra information or modify the behavior of the object. Flags can be used to mark objects with special attributes or states, such as

whether the object is read-only, temporary, or has other specific properties. Length (whNvmSize len): The length of the data associated with the object, in bytes.
- Label (`uint8_t label[]`): A human-readable label or name for the object.

0.4.4.2 NVM Access and Flags NVM access controls are stored in the metadata for all objects and are returned by `wh_Nvm_GetMetadata()` and related client APIs.

NVM flags provide object and key policy hints that are enforced by the NVM library and keystore. Relevant flags include:

- `WH_NVM_FLAGS_NONMODIFIABLE`: Object cannot be modified and/or destroyed.
- `WH_NVM_FLAGS_NONDESTROYABLE`: Object cannot be destroyed.
- `WH_NVM_FLAGS_NONEXPORTABLE`: Object data cannot be read/exported.
- `WH_NVM_FLAGS_USAGE_*`: Key usage policy flags for encrypt/decrypt/sign/verify/wrap/derive.
- `WH_NVM_FLAGS_USAGE_ANY`: Allow all usage flags.

0.4.4.3 NVM Architecture The wolfHSM server follows the generic component architecture approach to handle Non-Volatile Memory (NVM) operations. The configuration is divided into generic and specific parts, allowing for flexibility and customization.

1. **Generic Configuration (wh_nvm.h):** This header file defines the generic interface for NVM operations. It includes function pointers for NVM operations like `nvm_Read`, `nvm_Write`, `nvm_Erase`, and `nvm_Init`. These function pointers are part of the `whNvmConfig` structure, which is used to bind an actual NVM implementation to the abstract NVM interface.
2. **Specific Configuration (wh_nvm_flash.c, wh_nvm_flash.h):** These files provide a specific implementation of the NVM interface for flash memory. The functions defined here adhere to the function signatures defined in the generic interface, allowing them to be used as the actual implementation for the NVM operations.

The `whServerContext` structure includes a `whNvmConfig` member. This is used to bind the NVM operations to the server context, allowing the server to perform NVM operations using the configured NVM interface.

Steps required to initialize NVM on the server are:

1. Allocate and initialize a `whNvmConfig` structure, providing bindings to a specific NVM back-end (e.g., from `wh_nvm_flash.c`).
2. Allocate and initialize a `whServerConfig` structure, and set its `nvmConfig` member to the `whNvmConfig` structure initialized in step 1.
3. Allocate a `whServerContext` structure.
4. Initialize the server with the `whServerConfig` structure by calling `wh_Server_Init()`.

This allows the server to use the configured NVM operations on the given backing store, which can be easily swapped out by providing a different implementation in the `whNvmConfig` structure.

0.4.4.4 NVM Back-Ends Currently, wolfHSM only supports one NVM back-end provider: the NVM flash module (`wh_nvm_flash.c`). This module provides a concrete implementation of the NMV interface functions (`wh_nvm.h`), mapping the NVM data store to a flash memory device. The low-level flash drivers are device-specific and themselves specified as generic components (`wh_flash.h`) that can be swapped out depending on the target hardware.

0.4.5 Key Management

The wolfHSM library provides comprehensive key management capabilities, including storing, loading, and exporting keys from non-volatile memory, caching of frequently used keys in RAM for fast access, and interacting with hardware-exclusive device keys. Keys are stored in non-volatile memory along

side other NVM objects with corresponding access protections. wolfHSM will automatically load keys into the necessary cryptographic hardware when the key is selected for use with a specific consumer. More information on the key management API can be found in the client library and API documentation sections.

0.4.5.1 Key Revocation Key revocation provides a lightweight mechanism to invalidate a key without destroying its storage. When a key is revoked on the server, its metadata is updated by setting `WH_NVM_FLAGS_NONMODIFIABLE` and clearing all `WH_NVM_FLAGS_USAGE_*` bits. The key remains present in cache/NVM, but is no longer eligible for cryptographic operations that enforce usage flags.

Runtime behavior for revoked keys:

- Cryptographic operations that enforce usage flags return `WH_ERROR_USAGE` when the required usage bit is no longer set.
- The `WH_NVM_FLAGS_NONMODIFIABLE` setting prevents further key metadata changes and blocks NVM modification or destruction checks.
- Revocation does not automatically set `WH_NVM_FLAGS_NONEXPORTABLE`; export behavior remains controlled by those flags.

Persistence behavior:

- Revocation is committed to NVM for keys that already exist in NVM, so the revoked state persists across resets or power cycles.
- If a key exists only in cache and has not been committed, the revoked state is limited to the cache lifetime.

0.4.6 Cryptographic Operations

One of the defining features of wolfHSM is that it enables the client application to use the wolfCrypt API directly, but with the underlying cryptographic operations actually being executed on the HSM core. This is an incredibly powerful feature for a number of reasons:

- client applications are dramatically simpler as they do not need to set up the complicated communication transactions required to pass data back and forth between the HSM
- local and remote HSM implementations can be easily switched between by changing a single parameter to the wolfCrypt call, enabling maximum flexibility of implementation and ease of development. Client application development can be prototyped with local instances of wolfCrypt before the HSM core is even brought on-line
- the wolfHSM API is simple, stable, well documented, and battle tested

The ability to easily redirect wolfCrypt API calls to the wolfHSM server is based on the “crypto callback” (a.k.a cryptocb) of wolfCrypt.

The wolfHSM client is able to redirect wolfCrypt API calls to the wolfHSM server by implementing the remote procedure call logic as a crypto callback. The Crypto callback framework in wolfCrypt enables users to override the default implementation of select cryptographic algorithms and provide their own custom implementations at runtime. The wolfHSM client library registers a crypto callback with wolfCrypt that transforms each wolfCrypt crypto API function into a remote procedure call to the HSM server to be executed in a secure environment. Crypto callbacks are selected for use based on the device ID (`devId`) parameter accepted by most wolfCrypt API calls.

wolfHSM defines the `WOLFHSM_DEV_ID` value to represent the wolfHSM server crypto device, which can be passed to any wolfCrypt function as the `devId` parameter. wolfCrypt APIs that support the `devId` parameter can be passed `WOLFHSM_DEV_ID` and, if supported, the cryptographic operation will be automatically exectued by the wolfHSM server.

0.4.6.1 Hardware Cryptography Support Many HSM devices also have hardware acceleration capabilities for select algorithms available. In these cases, the wolfHSM server may also support offloading the HSM server-side cryptography to device hardware. If supported, the wolfHSM server can be configured to do this automatically with no input required from the user. Any port-specific hardware acceleration capabilities will be documented in the wolfHSM port for that device.

0.4.7 AUTOSAR SHE

TODO

0.5 wolfHSM Client Library

The client library API is the primary mechanism through which users will interact with wolfHSM. Refer to the API documentation for a full list of available functions and their descriptions.

0.5.1 Table of Contents

0.5.2 API Return Codes

All client API functions return a wolfHSM error code indicating success or the type of failure. Some failures are critical errors, while others may simply indicate an action is required from the caller (e.g. `WH_ERROR_NOTREADY` in the case of a non-blocking operation). Many client APIs also propagate a server error code (and in some cases an additional status) to the caller, allowing for the case where the underlying request transaction succeeded but the server was unable to perform the operation. Examples of this include requesting an NVM object from the server that doesn't exist, attempting to add an object when NVM is full, or trying to use a cryptographic algorithm that the server is not configured to support.

Error codes are defined in `wolfhsm/wh_error.h`. Refer to the API documentation for more details.

0.5.3 Split Transaction Processing

Most client APIs are fully asynchronous and decomposed into split transactions, meaning there is a separate function for the operation request and response. The request function sends the request to the server and immediately returns without blocking. The response function polls the underlying transport for a response, processing it if it exists, and immediately returning if it has not yet arrived. This allows for the client to request long running operations from the server without wasting client CPU cycles. The following example shows an example asynchronous request and response invocation using the "echo" message:

```
int rc;

/* send an echo request */
```

```
rc = wh_Client_EchoRequest(&clientCtx, sendLen, &sendBuffer);
if (rc != WH_ERROR_OK) {
    /* handle error */
}

/* do work... */

/* poll for a server response */
while ((rc = wh_Client_EchoResponse(client, &recv_len, recv_buffer)) ==
 ↪  WH_ERROR_NOTREADY) {
    /* do work or yield */
}

if (rc != WH_ERROR_OK) {
    /* handle error */
}
```

0.5.4 The Client Context

The client context structure (`whClientContext`) holds the runtime state of the client and represents the endpoint of the connection with the server. There is a one-to-one relationship between client and server contexts, meaning an application that interacts with multiple servers will need multiple client contexts - one for each server. Each client API function takes a client context as an argument, indicating which server connection the operations will correspond to. If familiar with wolfSSL, the client context structure is analogous to the `WOLFSSL` connection context structure.

0.5.4.1 Initializing the client context Before using any client APIs on a client context, the structure must be configured and initialized using the `whClientConfig` configuration structure and the `wh_Client_Init()` function.

The client configuration structure holds the communication layer configuration (`whCommClientConfig`) that will be used to configure and initialize the context for the server communication. The `whCommClientConfig` structure binds an actual transport implementation (either built-in or custom) to the abstract comms interface for the client to use.

The general steps to configure a client are:

1. Allocate and initialize a transport configuration structure, context, and callback implementation for the desired transport
2. Allocate and comm client configuration structure and bind it to the transport configuration from step 1 so it can be used by the client
3. Allocate and initialize a client configuration structure using the comm client configuration in step 2
4. Allocate a client context structure
5. Initialize the client with the client configuration by calling `wh_Client_Init()`
6. Use the client APIs to interact with the server

Here is a bare-minimum example of configuring a client application to use the built-in shared memory transport:

```
#include <string.h> /* for memcmp() */
#include "wolfhsm/client.h"  /* Client API (includes comm config) */
#include "wolfhsm/wh_transport_mem.h" /* transport implementation */

/* Step 1: Allocate and initialize the shared memory transport configuration */
```

```
/* Shared memory transport configuration */
static whTransportMemConfig transportMemCfg = { /* shared memory config */ };
/* Shared memory transport context (state) */
whTransportMemClientContext transportMemClientCtx = {0};
/* Callback structure that binds the abstract comm transport interface to
 * our concrete implementation */
whTransportClientCb transportMemClientCb = {WH_TRANSPORT_MEM_CLIENT_CB};

/* Step 2: Allocate client comm configuration and bind to the transport */
/* Configure the client comms to use the selected transport configuration */
whCommClientConfig commClientCfg[1] = {{
            .transport_cb      = transportMemClientCb,
            .transport_context = (void*)transportMemClientCtx,
            .transport_config  = (void*)transportMemCfg,
            .client_id         = 123, /* unique client identifier */
}};

/* Step 3: Allocate and initialize the client configuration */
whClientConfig clientCfg= {
   .comm = commClientCfg,
};

/* Step 4: Allocate the client context */
whClientContext clientCtx = {0};

/* Step 5: Initialize the client with the provided configuration */
wh_Client_Init(&clientCtx, &clientCfg);
```

The client context is now initialized and can be used with the client library API functions in order to do work. Here is an example of sending an echo request to the server:

```
/* Step 6: Use the client APIs to interact with the server */

/* Buffers to hold sent and received data */
char recvBuffer[WH_COMM_DATA_LEN] = {0};
char sendBuffer[WH_COMM_DATA_LEN] = {0};

uint16_t sendLen = snprintf(&sendBuffer,
                            sizeof(sendBuffer),
                            "Hello World!\n");
uint16_t recvLen = 0;

/* Send an echo request and block on receiving a response */
wh_Client_Echo(client, sendLen, &sendBuffer, &recvLen, &recvBuffer);

if ((recvLen != sendLen ) ||
    (0 != memcmp(sendBuffer, recvBuffer, sendLen))) {
    /* Error, we weren't echoed back what we sent */
}
```

While there are indeed a large number of nested configurations and structures to set up, designing wolfHSM this way allowed for different transport implementations to be swapped in and out easily without changing the client code. For example, in order to switch from the shared memory transport to a TCP transport, only the transport configuration and callback structures need to be changed, and the rest of the client code remains the same (everything after step 2 in the sequence above).


```c
#include <string.h> /* for memcmp() */
#include "wolfhsm/client.h"  /* Client API (includes comm config) */
#include "port/posix_transport_tcp.h" /* transport implementation */

/* Step 1: Allocate and initialize the posix TCP transport configuration */
/* Client configuration/contexts */
whTransportClientCb posixTransportTcpCb = {PTT_CLIENT_CB};
posixTransportTcpClientContext posixTranportTcpCtx = {0};
posixTransportTcpConfig posixTransportTcpCfg = {
    /* IP and port configuration */
};

/* Step 2: Allocate client comm configuration and bind to the transport */
/* Configure the client comms to use the selected transport configuration */
whCommClientConfig commClientCfg = {{
             .transport_cb      = posixTransportTcpCb,
             .transport_context = (void*)posixTransportTcpCtx,
             .transport_config  = (void*)posixTransportTcpCfg,
             .client_id         = 123, /* unique client identifier */
}};

/* Subsequent steps remain the same... */
```

Note that the echo request in step 6 is just a simple usage example. Once the connection to the server is set up, any of the client APIs are available for use.

0.5.5 NVM Operations

This section provides examples of how to use the client NVM API. Blocking APIs are used for simplicity, though the split transaction APIs can be used in a similar manner.

Client usage of the server NVM storage first requires sending an initialization request to the server. This currently does not trigger any action on the server side but it may in the future and so it is recommended to include in client applications.

```c
int rc;
int serverRc;
uint32_t clientId; /* unused for now */
uint32_t serverId;

rc = wh_Client_NvmInit(&clientCtx, &serverRc, &clientId, &serverId);

/* error check both local and remote error codes */
/* serverId holds unique ID of server */
```

Once initialized, the client can create and add an object using the `NvmAddObject` functions. Note that a metadata entry must be created for every object.

```c
int serverRc;

whNvmId id = 123;
whNvmAccess access = WOLFHSM_NVM_ACCESS_ANY;
whNvmFlags flags = WOLFHSM_NVM_FLAGS_ANY;
uint8_t label[] = “My Label”;

uint8_t myData[] = “This is my data.”
```

```
whClient_NvmAddObject(&clientCtx, id, access, flags, strlen(label), &label,
 ↪  sizeof(myData), &myData, &serverRc);
```

Data corresponding to an existing objects can be updated in place:

```
byte myUpdate[] = “This is my update.”

whClient_NvmAddObject(&clientCtx, &myMeta, sizeof(myUpdate), myUpdate);
```

For objects that should not be copied and sent over the transport, there exist DMA versions of the NvmAddObject functions. These pass the data to the server by reference rather than by value, allowing the server to access the data in memory directly. Note that if your platform requires custom address translation or cache invalidation before the server may access client addresses, you will need to implement a DMA callback.

```
whNvmMetadata myMeta = {
  .id = 123,
  .access = WOLFHSM_NVM_ACCESS_ANY,
  .flags = WOLFHSM_NVM_FLAGS_ANY,
  .label = “My ”Label
};


uint8_t myData[] = “This is my data”.

wh_Client_NvmAddObjectDma(client, &myMeta, sizeof(myData), &myData), &serverRc
   );
```

NVM Object data can be read using the NvmRead functions. There also exist DMA versions of NvmRead functions that can be used identically to their AddbjectDma counterparts.

```
const whNvmId myId = 123; /* ID of the object we want to read */
const whNvmSize offset = 0; /* byte offset into the object data */

whNvmSize outLen; /* will hold length in bytes of the requested data */
int outRc; /* will hold server return code */

byte myBuffer[BIG_SIZE];

whClient_NvmRead(&clientCtx, myId, offset, sizeof(myData), &serverRc, outLen,
 ↪  &myBuffer)
/* or via DMA */
whClient_NvmReadDma(&clientCtx
iint wh_Client_NvmReadDma(&clientCtx, myid, offset, sizeof(myData), &myBuffer,
 ↪  &serverRc);
```

Objects can be deleted/destroyed using the NvmDestroy functions. These functions take a list (array) of object IDs to be deleted. IDs in the list that are not present in NVM do not cause an error.

```
whNvmId idList[] = {123, 456};
whNvmSize count = sizeof(myIds)/ sizeof(myIds[0]);
int serverRc;

wh_Client_NvmDestroyObjectsRequest(&clientCtx, count, &idList);
wh_Client_NvmDestroyObjectsResponse(&clientCtx, &serverRc);
```

The objects in NVM can also be enumerated using the NvmList functions. These functions retrieve

the next matching id in the NVM list starting at `start_id`, and sets `out_count` to the total number of IDs that match `access` and `flags`:

```
int wh_Client_NvmList(whClientContext* c,
        whNvmAccess access, whNvmFlags flags, whNvmId start_id,
        int32_t *out_rc, whNvmId *out_count, whNvmId *out_id);
```

For a full description of all the NVM API functions, please refer to the API documentation.

0.5.6 NVM Flags

NVM objects include flags in their metadata. Flags (such as `WH_NVM_FLAGS_NONMODIFIABLE`, `WH_NVM_FLAGS_NONDESTROYABLE`, and `WH_NVM_FLAGS_NONEXPORTABLE`) are enforced by the server NVM policy checks. Key usage flags (`WH_NVM_FLAGS_USAGE_*`) are enforced by the keystore during cryptographic operations.

0.5.7 Key Management

Keys meant for use with wolfCrypt can be loaded into the HSM's keystore and optionally saved to NVM with the following APIs:

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

uint16_t keyId = WOLFHSM_KEYID_ERASED;
uint32_t keyLen;
byte key[AES_128_KEY_SIZE] = { /* AES key */ };
byte label[WOLFHSM_NVM_LABEL_LEN] = { /* Key label */ };

whClientContext clientCtx;
whClientCfg clientCfg = { /* config */ };

wh_Client_Init(&clientCtx, &clientCfg);

wh_Client_KeyCache(clientCtx, 0, label, sizeof(label), key, sizeof(key),
 ↪  &keyId);
wh_Client_KeyCommit(clientCtx, keyId);
wh_Client_KeyEvict(clientCtx, keyId);
keyLen = sizeof(key);
wh_Client_KeyExport(clientCtx, keyId, label, sizeof(label), key, &keyLen);
wh_Client_KeyErase(clientCtx, keyId);
```

`wh_Client_KeyCache` will store the key and label in the HSM's ram cache and correlate it with the `keyId` passed in. Using a `keyId` of `WOLFHSM_KEYID_ERASED` will make wolfHSM assign a new, unique `keyId` that will be returned through the keyId parameter. wolfHSM has a limited number of cache slots, configured by `WOLFHSM_NUM_RAMKEYS`, and will return `WH_ERROR_NOSPACE` if all keyslots are full. Keys that are in cache and NVM will be removed from the cache to make room for more keys since they're backed up in NVM. `wh_Client_KeyCommit` will save a cached key to NVM with the key indicated by it's keyId. `wh_Client_KeyEvict` will evict a key from the cache but will leave it in NVM if it's been commited. `wh_Client_KeyExport` will read the key contents out of the HSM back to the client. `wh_Client_KeyErase` will remove the indicated key from cache and erase it from NVM.

0.5.8 Key Revocation

Key revocation updates key metadata to prevent further cryptographic use without destroying storage. Revocation clears all `WH_NVM_FLAGS_USAGE_*` bits and sets `WH_NVM_FLAGS_NONMODIFIABLE`. The revoked state is persisted when the key is already committed to NVM.

Creating a key with NVM usage flags:

```
int rc;
uint16_t keyId = WH_KEYID_ERASED;
byte label[WH_NVM_LABEL_LEN] = "app-signing";
byte key[AES_128_KEY_SIZE] = { /* key bytes */ };
whNvmFlags flags = WH_NVM_FLAGS_USAGE_SIGN | WH_NVM_FLAGS_NONEXPORTABLE;

rc = wh_Client_KeyCache(&clientCtx, flags, label, sizeof(label),
                        key, sizeof(key), &keyId);
if (rc == WH_ERROR_OK) {
    rc = wh_Client_KeyCommit(&clientCtx, keyId);
}
```

Revoking a key:

```
int rc;

rc = wh_Client_KeyRevoke(&clientCtx, keyId);
if (rc != WH_ERROR_OK) {
    /* handle error */
}
```

Attempting to use a revoked key and handling failure:

```
int rc;
Aes aes;
byte iv[AES_BLOCK_SIZE] = {0};
byte plain[AES_BLOCK_SIZE] = {0};
byte cipher[AES_BLOCK_SIZE] = {0};

wc_AesInit(&aes, NULL, WOLFHSM_DEV_ID);
wh_Client_AesSetKeyId(&aes, keyId);
wc_AesSetIV(&aes, iv);

rc = wh_Client_AesCbc(&clientCtx, &aes, AES_ENCRYPTION,
                      plain, sizeof(plain), cipher);
if (rc == WH_ERROR_USAGE) {
    /* key revoked or usage not permitted */
}
wc_AesFree(&aes);
```

Compatibility notes:

- Keys stored with `WH_NVM_FLAGS_NONE` (no usage flags) are treated as not permitted for cryptographic use and will return `WH_ERROR_USAGE`.
- Keys committed to NVM retain revocation state across resets; cached-only keys do not persist after reset or eviction.

0.5.8.1 Key revocation client API

0.5.8.1.1 wh_Client_KeyRevokeRequest Send a key revocation request to the server (non-blocking).

This function prepares and sends a revoke request for the specified key ID. It returns after the request is sent; use `wh_Client_KeyRevokeResponse()` to retrieve the result.

Parameters:

- c: Client context.
- keyId: Key ID to revoke.

Return values:

- WH_ERROR_OK on successful request send.
- A negative error code on failure.

Error codes:

- WH_ERROR_BADARGS if c is NULL or keyId is invalid.
- Propagates comm layer errors on send failure.

0.5.8.1.2 wh_Client_KeyRevokeResponse Receive a key revocation response.

This function polls for the revoke response and returns WH_ERROR_NOTREADY until the server reply is available.

Parameters:

- c: Client context.

Return values:

- WH_ERROR_OK on success.
- WH_ERROR_NOTREADY if the response has not arrived.
- A negative error code on failure.

Error codes:

- WH_ERROR_BADARGS if c is NULL.
- Server error codes such as WH_ERROR_NOTFOUND.

0.5.8.1.3 wh_Client_KeyRevoke Revoke a key using a blocking request/response.

This helper sends a revoke request and waits for the response.

Parameters:

- c: Client context.
- keyId: Key ID to revoke.

Return values:

- WH_ERROR_OK on success.
- A negative error code on failure.

Error codes:

- Any error code returned by wh_Client_KeyRevokeRequest() or wh_Client_KeyRevokeResponse().

0.5.9 Cryptography

When using wolfCrypt in the client application, compatible crypto operations can be executed on the wolfHSM server by passing WOLFHSM_DEV_ID as the devId argument. The wolfHSM client must be initialized before using any wolfHSM remote crypto.

If wolfHSM does not yet support that algorithm, the API call will return CRYPTOCB_UNAVAILABLE.

Here is an example of how a client application would perform an AES CBC encryption operation on the wolfHSM server:

```c
#include "wolfhsm/client.h"
#include "wolfssl/wolfcrypt/aes.h"

whClientContext clientCtx;
whClientCfg clientCfg = { /* config */ };

wh_Client_Init(&clientCtx, &clientCfg);

Aes aes;
byte key[AES_128_KEY_SIZE] = { /* AES key */ };
byte iv[AES_BLOCK_SIZE] = { /* AES IV */ };

byte plainText[AES_BLOCK_SIZE] = { /* plaintext */ };
byte cipherText[AES_BLOCK_SIZE];

wc_AesInit(&aes, NULL, WOLFHSM_DEV_ID);

wc_AesSetKey(&aes, &key, AES_BLOCK_SIZE, &iv, AES_ENCRYPTION);

wc_AesCbcEncrypt(&aes, &cipherText, &plainText, sizeof(plainText));

wc_AesFree(&aes);
```

If it is necessary to use an HSM-owned key instead of a client-owned key (e.g. a HSM hardware key), client API functions such as `wh_Client_SetKeyAes` (or similar for other crypto algorithms) will make wolfHSM use the indicated HSM key for the subsequent cryptographic operation instead of requiring a client-supplied key:

```c
#include "wolfhsm/client.h"
#include "wolfssl/wolfcrypt/aes.h"

whClientContext clientCtx;
whClientCfg clientCfg = { /* config */ };

wh_Client_Init(&clientCtx, &clientCfg);

uint16_t keyId;
Aes aes;
byte key[AES_128_KEY_SIZE] = { /* AES key */ };
byte label[WOLFHSM_NVM_LABEL_LEN] = { /* Key label */ };
byte iv[AES_BLOCK_SIZE] = { /* AES IV */ };

byte plainText[AES_BLOCK_SIZE] = { /* plaintext */ };
byte cipherText[AES_BLOCK_SIZE];

wc_AesInit(&aes, NULL, WOLFHSM_DEV_ID);

/* IV needs to be set seperate from the key */
wc_AesSetIV(&aes, iv);

/* this key can be cached at any time before use, done here for the sake of
 ↪  example */
wh_Client_KeyCache(clientCtx, 0, label, sizeof(label), key, sizeof(key),
 ↪  &keyId);
```

```
wh_Client_SetKeyAes(&aes, keyId);

wc_AesCbcEncrypt(&aes, &cipherText, &plainText, sizeof(plainText));

/* key eviction is optional, the key can be stored in cache or NVM and used
 ↪  with wolfCrypt */
wh_Client_KeyEvict(clientCtx, keyId);

wc_AesFree(&aes);
```

If it is desired to run the crypto locally on the client, all that is necessary is to pass `INVALID_DEVID` to `wc_AesInit()`:

```
wc_AesInit(&aes, NULL, INVALID_DEVID);
```

Outside of the steps mentioned above, the usage of the wolfHSM API should be otherwise unchanged. Please consult the wolfCrypt API reference inside the wolfSSL manual for further usage instructions and the extensive list of supported cryptographic algorithms.

0.5.9.1 CMAC For CMAC operations that need to use cached keys, seperate wolfHSM specific functions must be called to do the CMAC hash and verify operation in one function call. The normal `wc_AesCmacGenerate_ex` and `wc_AesCmacVerify_ex` are acceptable to use if the client can supply a key when the functions are invoked, but in order to use a pre-cached key, `wh_Client_AesCmacGenerate` and `wh_Client_AesCmacVerify` must be used. The non-oneshot functions `wc_InitCmac_ex`, `wc_CmacUpdate` and `wc_CmacFinal` can be used with either a client-side key or a pre-cached key. To use a cached key for these functions, the caller should pass a `NULL` key parameter and use `wh_Client_SetKeyCmac` to set the appropriate keyId.

0.5.10 AUTOSAR SHE API

0.6 wolfHSM Server Library

The wolfHSM server library is a server-side implementation of the wolfCrypt cryptography library. It provides an interface for applications to offload cryptographic operations to a dedicated server, which runs the wolfHSM server software. This allows the application to perform cryptographic operations without having to manage the cryptographic keys or perform the operations locally.

0.6.1 Getting Started

TODO

0.6.2 Architecture

TODO

0.6.3 API Reference

TODO

0.6.4 Key Management

TODO

0.6.5 Cryptographic

wolfHSM uses wolfCrypt for all cryptographic operations, which means wolfHSM can offload any algorithm supported by wolfCypt to run on the wolfHSM server. This includes the Chinese government mandated ShāngMì ciphers (SM2, SM3, SM4), as well as post-quantum algorithms such as Kyber, LMS, XMSS, and more!

0.7 Customizing wolfHSM

wolfHSM provides multiple points of customization via build time options and user-supplied callbacks, enabling it to to be tailored to a wide range of use cases and environments without requiring changes to the core library code. This chapter provides an overview of the customization options available in wolfHSM, including:

- Library Configuration: Compile-time options that can be used to enable or disable specific features in the library.
- DMA Callbacks: Custom callbacks that can be registered with the server to perform operations before and after accessing client memory directly.
- DMA Address Allow List: A mechanism for the server to restrict the client's access to specific memory regions.
- Custom Callbacks: Custom callbacks that can be registered with the server and invoked by the client to perform specific operations that are not covered by the default HSM capabilities.

0.7.1 Library Configuration

The wolfHSM library has a number of build options that can be turned on or off though compile time definitions. The library expects these configuration macros to be defined in a configuration header named `wh_config.h`. This file should be defined by applications using wolfHSM and located in a directory in the compilers include path.

An example `wh_config.h` is distributed with every wolfHSM port providing a known good configuration.

For a full list of wolfHSM configuration settings that can be defined in `wh_config.h`, refer to the API documentation.

0.7.2 DMA Callbacks

The Direct Memory Access (DMA) callback feature in wolfHSM provides hooks on the server side for custom operations before and after accessing client memory directly. This is often required when porting to a new shared memory architecture. The feature is particularly useful for scenarios where the server needs to perform specific actions such as cache invalidation, address translation, or other custom memory manipulations to ensure coherency between client and server memory.

Callbacks can be registered with a server using the `wh_Server_DmaRegisterCb32()` and `wh_Server_DmaRegisterCb64()` functions, which bind the supplied callback to all DMA operations on the server context.

Separate callback functions for handling 32 and 64-bit addresses are required, corresponding to the distinct 32 and 64-bit client DMA API functions. Callback functions are of type `whServerDmaClientMem32Cb` and `whServerDmaClientMem64Cb`, respectively, defined as:

```
typedef int (*whServerDmaClientMem32Cb)(struct whServerContext_t* server,
                                        uint32_t clientAddr, void** serverPtr,
                                        uint32_t len, whServerDmaOper oper,
                                        whServerDmaFlags flags);
typedef int (*whServerDmaClientMem64Cb)(struct whServerContext_t* server,
```

```
                                    uint64_t clientAddr, void** serverPtr,
                                    uint64_t len, whServerDmaOper oper,
                                    whServerDmaFlags flags);
```

The DMA callback functions receive the following arguments:

- `server`: A pointer to the server context.
- `clientAddr`: The client memory address to be accessed.
- `serverPtr`: A pointer to a the server memory address (also a pointer), which the callback will set after applying any necessary transformations/remappings
- `len`: The length of the requested memory operation in bytes
- `oper`: The type of memory operation (injection point in the next section) that is about to be performed on the transformed server address
- `flags`: Additional flags for the memory operation. Right now these are reserved for future use and should be ignored.

The callback should return `WH_ERROR_OK` on success, or an error code if an error occurs. The server will propagate the error code back to the client if the callback fails.

0.7.2.1 Callback Locations The DMA callbacks are at four distinct points around the server's memory access:

- Pre-Read: Callback is invoked before reading data from the client memory. The server should use the callback to perform any necessary pre-read operations, such as address translation or cache invalidation.
- Post-Read: Callback is invoked after reading data from the client memory. The server should use the callback to perform any necessary post-read operations, such as cache synchronization.
- Pre-Write: Callback is invoked before writing data to the client memory. The server should use the callback to perform any necessary pre-write operations, such as address translation or cache invalidation.
- Post-write: Callback is invoked after writing data to the client memory. The server should use the callback to perform any necessary post-write operations, such as cache synchronization.

The point at which the callback is invoked is passed into the callback through the `oper` argument, which can take the following values:

```
typedef enum {
    WH_SERVER_DMA_OPER_PRE_READ,  /* Pre-read operation */
    WH_SERVER_DMA_OPER_POST_READ, /* Post-read operation */
    WH_SERVER_DMA_OPER_PRE_WRITE, /* Pre-write operation */
    WH_SERVER_DMA_OPER_POST_WRITE /* Post-write operation */
} whServerDmaOper;
```

This enables the callback to `switch` on the `oper` value and perform custom logic based on the type of memory operation being performed. An example DMA callback implementation is shown below:

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

/* Example DMA callback for 32-bit client addresses */
int myDmaCallback32(whServerContext* server, uint32_t clientAddr,
                    void** xformedCliAddr, uint32_t len,
                    whServerDmaOper oper, whServerDmaFlags flags)
{
    /* Optionally transform client address to server address space, e.g.
     ↪  memmap() */
    *xformedCliAddr = (void*)clientAddr; /* do transformation */
```

```
    switch (oper) {
        case WH_DMA_OPER_CLIENT_READ_PRE:
            /* Pre-Read Operation here, e.g. cache invalidation */
            break;
        case WH_DMA_OPER_CLIENT_READ_POST:
            /* Post-Read Operation here */
            break;
        case WH_DMA_OPER_CLIENT_WRITE_PRE:
            /* Pre-Write Operation here */
            break;
        case WH_DMA_OPER_CLIENT_WRITE_POST:
            /* Post-Write Operation here, e.g. cache flush */
            break;
        default:
            return WH_ERROR_BADARGS;
    }

    return WH_ERROR_OK;
}
```

0.7.2.2 Callback Registration The callback can be registered with the server context, either at initialization through the server configuration structure, or at any time after initialization using the callback registration functions.

To register the callback at initialization, the callback function should be included in the DMA configuration structure within the server configuration structure. Note that the callback functions are optional, so unused callbacks can be set to NULL.

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

/* Example of initializing a server config structr with a DMA32 callback then
 ↪  initializing the server */
int main(void)
{
    whServerDmaConfig dmaCfg = {0};
    dmaCfg.dma32Cb = myDmaCallback32;


    whServerConfig serverCfg = {
        .dmaCfg = dmaCfg,

        /* other configuration omitted for brevity */
    };

    whServerContext serverCtx;

    wh_Server_Init(&serverCtx, &serverCfg);

    /* server app logic */
}
```

To register the callback after initialization, first initialize the server context with the desired configuration, then call the appropriate registration function.

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

```
int main(void)
{

    whServerConfig serverCfg =  { /* server config */ };

    whServerContext serverCtx;

    wh_Server_Init(&serverCtx, &serverCfg);

    /* register the callback defined above */
    wh_Server_DmaRegisterCb32(&serverCtx, myDmaCallback32);

    /* server app logic */
}
```

0.7.3 DMA Address Allow List

wolfHSM also exposes an "allow list" for client DMA addresses, providing a mechanism for the server to restrict the client's access to a pre-configured list of specific memory regions. This feature is particularly useful in scenarios where the server needs to limit the client's access to certain memory regions to prevent unauthorized access or to ensure that the client only accesses memory that is safe to access. For example, in a multicore system with one client running per-core, it is most likely that clients should not be able to access each others memory regions, nor read out server memory which could contain sensitive information like cryptographic keys.

It is important to note that the software allow list feature is meant to work as a second layer of protection on top of device-specific memory protection mechanisms, and should not be considered a first line of defense in preventing unauthorized memory accesses. It is imperative that the user configure the device-specific memory protection mechanisms required to enforce the isolation of their applications and segment the HSM core and associated memory from the rest of the system.

0.7.3.1 Registering an Allow List Similar to the DMA callbacks, the allow list can be registered with the server context, either at initialization through the server configuration structure, or at any time after initialization using the allow list registration functions.

To register the list at initialization, the list should be populated in the DMA configuration structure inside the server configuration structure.

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

/* Define the allowed memory regions */
const whServerDmaAddrAllowList allowList = {
    .readList = {
        {(void*)0x20001000, 0x100},  /* Allow read from 0x20001000 to
↪   0x200010FF */
        {(void*)0x20002000, 0x200},  /* Allow read from 0x20002000 to
↪   0x200021FF */
    },
    .writeList = {
        {(void*)0x20003000, 0x100},  /* Allow write from 0x20003000 to
↪   0x200030FF */
        {(void*)0x20004000, 0x200},  /* Allow write from 0x20004000 to
↪   0x200041FF */
    },
```

```
};

int main()
{
    whServerConfig config;


    whServerDmaConfig dmaCfg = {0};
    dmaCfg.allowList = &allowList;

    whServerConfig serverCfg = {
        .dmaCfg = dmaCfg,
        /* other configuration omitted for brevity */
    };

    whServerContext server;

    wh_Server_Init(&server, &config);

    /* Server is now configured with the allowlist */
    /* Perform other server operations */

    /* Allow list can also be registered after initialization if the
     * list is not present in the server configuration struct using:
     *
     *    wh_Server_DmaRegisterAllowList(&server, &allowList);
     */
}
```

Once registered, all DMA operations requested of the server by the client will be checked against the allow list. If the client attempts to access a memory region that is not in the allow list, the server will return an error to the client, and the operation will not be performed.

0.7.4 Custom Callbacks

The custom callback feature in wolfHSM allows developers to extend the functionality of the library by registering custom callback functions on the server. These callbacks can then be invoked by clients to perform specific operations that are not covered by the default HSM capabilities such as enabling or disabling peripheral hardware, implementing custom monitoring or authentication routines, staging secure boot for an additional core, etc.

0.7.4.1 Server side The server can register custom callback functions that define specific operations. These functions must be of type `whServerCustomCb`.

```
/* wh_server.h */

/* 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 */
);
```

Custom server callback functions are associated with unique identifiers (IDs), which correspond to indices into the server's custom callback dispatch table. The client will refer to the callback by it's ID

when it requests invocation.

The custom callback has access to data passed from the client by value or by reference (useful in a shared memory system) through the whMessageCustomCb_Request argument passed into the callback function. The callback can act on the input data and produce output data that can be passed back to the client through th ewhMessageCustomCb_Response argument. The custom callback does not need to handle sending or receiving any of the input / output client data, as this is handled externally by wolfHSM. The response structure also contains fields for an error code and return code to propagate back to the client. The error code should be populated by the callback, and the return code will be set the return value from the custom callback.

0.7.4.2 Client Side Clients can send requests to the server to invoke these custom callbacks. The API provides a request and response function similar to the other functions in the client API. The client should declare an instance of a custom request structure, populate it with its custom data, and then send it to the server using wh_Client_CustomCbRequest(). The server response can then be polled using wh_Client_CustomCbResponse(), and the response data will populate the output whMessageCustomCb_Response() on successful receipt.

The client can also check the registration status of a given callback ID using the wh_Client_CustomCheckRegistered family of functions. This function queries the server for whether a given callback ID is registered in its internal callback table. The server responds with a true or false indicating the registration status.

0.7.4.3 Custom Messaging The client is able to pass data in and receive data from the custom callbacks through the custom request and response message data structures. These custom request and response messages are structured to include a unique ID, a type indicator, and a data payload. The ID corresponds to the index in the server's callback table. The type field indicating to the custom callback how the data payload should be interpreted. The data payload is a fixed size data buffer that the client can use in any way it wishes. The response structure contains additional error code values described above.

```
/* request message to the custom server callback */
typedef struct {
    uint32_t               id;   /* indentifier of registered callback  */
    uint32_t               type; /* whMessageCustomCb_Type */
    whMessageCustomCb_Data data;
} whMessageCustomCb_Request;

/* response message from the custom server callback */
typedef struct {
    uint32_t id;   /* indentifier of registered callback  */
    uint32_t type; /* whMessageCustomCb_Type */
    int32_t  rc;   /* Return code from custom callback. Invalid if err != 0 */
    int32_t  err;  /* wolfHSM-specific error. If err != 0, rc is invalid */
    whMessageCustomCb_Data data;
} whMessageCustomCb_Response;
```

0.7.4.4 Defining Custom Data Types Custom data types can be defined using the whMessageCustomCb_Data union, which provides several helpful predefined structures for common data types (e.g., dma32, dma64) and a raw data buffer (buffer) for user-defined schemas. Clients can indicate to the server callback how it should interpret the data in the union through the type field in the request. wolfHSM reserves the first few type indices for internal use, with the remainder of the type values available for custom client types.

0.7.4.5 Custom Callback Example In this example, a custom callback is implemented that is able to process three types of client requests, one using the built-in DMA-style addressing type, and two that use custom user defined types.

First, common messages shared between the client and server should be defined:

```
/* my_custom_cb.h */

#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,
} myUserDefinedTypes;

typedef struct {
    int foo;
    int bar;
} myCustomCbDataA;

typedef struct {
    int noo;
    int baz;
} myCustomCbDataB;
```

On the server side, the callback must be defined and then registered with the server context before processing requests. Note that the callback can be registered at any time, not necessarily before processing the first request.

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

int doWorkOnClientAddr(uint8_t* addr, uint32_t size) {
    /* do work */
}

int doWorkWithTypeA(myCustomTypeA* typeA) {
    /* do work */
}

int doWorkWithTypeB(myCustomTypeB* typeB) {
    /* do work */
}

static int customServerCb(whServerContext*                server,
                          const whMessageCustomCb_Request* req,
                          whMessageCustomCb_Response*      resp)
{
    int rc;

    resp->err = WH_ERROR_OK;

    /* detect and handle DMA request */
    if (req->type == WH_MESSAGE_CUSTOM_CB_TYPE_DMA32) {
```

```
        uint8_t* clientPtr =
         ↪  (uint8_t*)((uintptr_t)req->data.dma32.client_addr);
        size_t clientSz = req->data.dma32.client_sz;

        if (clientPtr == NULL) {
            resp->err = WH_ERROR_BADARGS;
        }
        else {
            rc = doWorkOnClientAddr(clientPtr, clientSz);
        }
    }
    else if (req->type == MY_TYPE_A) {
        myCustomCbDataA *data = (myCustomCbDataA*)((uintptr_t)req->data.data);
        rc = doWorkWithTypeA(data);
        /* optionally set error code of your choice */
        if (/* error condition */) {
            resp->err = WH_ERROR_ABORTED;
        }
    }
    else if (req->type == MY_TYPE_B) {
        myCustomCbDataB *data = (myCustomCbDataB)((uintptr_t)req->data.data);
        rc = doWorkWithTypeB(data);
        /* optionally set error code of your choice */
        if (/* error condition */) {
            resp->err = WH_ERROR_ABORTED;
        }
    }

    return rc;
}


int main(void) {

    whServerContext serverCtx;

    whServerConfig serverCfg = {
        /* your server configuration */
    };

    wh_Server_Init(&serverCtx, &serverCfg);

    wh_Server_RegisterCustomCb(&serverCtx, MY_CUSTOM_CB_ID, customServerCb));

    /* process server requests (simplified) */
    while (1) {
        wh_Server_HandleRequestMessage(&serverCtx);
    }

}
```

Now the client is able to check the registration of the custom callback, as well as invoke it remotely:

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

```
whClientContext clientCtx;
whClientConfig  clientCfg = {
    /* your client configuration */
};

whClient_Init(&clientCtx, &clientCfg);

bool isRegistered = wh_Client_CustomCheckRegistered(&client, MY_CUSTOM_CB_ID);

if (isRegistered) {
    uint8_t buffer[LARGE_SIZE] = {/* data*/};
    myCustomCbDataA typeA = {/* data */};
    myCustomCbDataB typeB = {/* data */};

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

    /* send custom request with built-in DMA type */
    req.id = MY_CUSTOM_CB_ID;
    req.type                   = WH_MESSAGE_CUSTOM_CB_TYPE_DMA32;
    req.data.dma32.client_addr = (uint32_t)((uintptr_t)&data);
    req.data.dma32.client_sz   = sizeof(data);
    wh_Client_CustomCbRequest(clientCtx, &req);
    wh_Client_CustomCbResponse(clientCtx, &resp);
    /* do stuff with response */

    /* send custom request with a user defined type */
    memset(req, 0, sizeof(req));
    req.id = MY_CUSTOM_CB_ID;
    req.type = MY_TYPE_A;
    memcpy(&req.data.data, typeA, sizeof(typeA));
    wh_Client_CustomCbRequest(clientCtx, &req);
    wh_Client_CustomCbResponse(clientCtx, &resp);
    /* do stuff with response */

    /* send custom request with a different user defined type */
    memset(req, 0, sizeof(req));
    req.id = MY_CUSTOM_CB_ID;
    req.type = MY_TYPE_B;
    memcpy(&req.data.data, typeA, sizeof(typeB));
    wh_Client_CustomCbRequest(clientCtx, &req);
    wh_Client_CustomCbResponse(clientCtx, &resp);
    /* do stuff with response */
}
```

0.8 WolfHSM Porting

This porting section aims to provide you with wolfHSM porting-related material and information. We will cover the following:

- WolfHSM Porting Overview
- WolfHSM Ports
- WolfHSM Porting Interface

0.8.1 WolfHSM Porting Overview

wolfHSM itself is not executable and it does not contain any code to interact with any specific hardware. In order for wolfHSM to run on a specific device, the library must be configured with the necessary hardware drivers and abstraction layers so that the server application can run and communicate with the client. Specifically, getting wolfHSM to run on real hardware requires the implementation of the following:

- Server application startup and hardware initialization
- Server wolfCrypt configuration
- Server non-volatile memory configuration
- Server and client transport configuration
- Server and client connection handling

The code that provides these requirements and wraps the server API into a bootable application is collectively referred to as a wolfHSM “port”.

Official ports of wolfHSM are provided for various supported architectures, with each port providing the implementation of the wolfHSM abstractions tailored to the specific device. Each port contains:

- Standalone Reference Server Application: This application is meant to run on the HSM core and handle all secure operations. It comes fully functional out-of-the-box but can also be customized by the end user to support additional use cases
- Client Library: This library can be linked against user applications to facilitate communication with the server

0.8.2 WolfHSM Ports

0.8.2.1 Infineon Aurix TC3XX The distribution of this port is restricted by the vendor. Please contact support@wolfssl.com for access.

Infineon Aurix TC3xx

- Up to 6x 300MHz TriCore application cores
- 1x 100MHz ARM Cortex M3 HSM core
- Crypto offload: TRNG, AES128, ECDSA, ED25519, SHA

0.8.2.2 ST SPC58NN The distribution of this port is restricted by the vendor. Please contact support@wolfssl.com for access.

** ST SPC58NN**

- 3x 200MHz e200z4256 PowerPC application cores
- 1x 100MHz e200z0 PowerPC HSM core with NVM
- Crypto offload: TRNG, AES128

0.8.2.3 POSIX The POSIX port provides multiple and fully functional implementations of different wolfHSM abstractions that can be used to better understand the exact functionality expected for different hardware abstractions.

The POSIX port provides: - Memory buffer transport - TCP transport - Unix domain transport - RAM-based and file-based NVM flash simulators

0.8.2.4 Skeleton The Skeleton port source code provides a non-functioning layout to be used as a starting point for future hardware/platform ports. Each function provides the basic description and expected flow with error cases explained so that ports can be used interchangeably with consistent results.

The Skeleton port provides stub implementations of: - Transport callbacks - NVM Flash callbacks - Crypto callbacks

0.8.3 WolfHSM Porting Interface

Ports must implement hardware-specific interfaces: - NVM flash interface

Crypto Hardware - TRNG, Keys, symmetric/asymmetric crypto

Platform Interface - Boot sequence, application core reset, memory limitations - Port and configuration are selected at compile time

.1 wolfHSM API reference

.1.1 Key Revocation

.1.1.1 wh_Client_KeyRevokeRequest Send a key revocation request to the server (non-blocking).

This function prepares and sends a revoke request for the specified key ID. It returns after the request is sent; use `wh_Client_KeyRevokeResponse()` to retrieve the result.

Parameters:

- `c`: Client context.
- `keyId`: Key ID to revoke.

Return values:

- `WH_ERROR_OK` on successful request send.
- A negative error code on failure.

Error codes:

- `WH_ERROR_BADARGS` if `c` is NULL or `keyId` is invalid.
- Propagates comm layer errors on send failure.

.1.1.2 wh_Client_KeyRevokeResponse Receive a key revocation response.

This function polls for the revoke response and returns `WH_ERROR_NOTREADY` until the server reply is available.

Parameters:

- `c`: Client context.

Return values:

- `WH_ERROR_OK` on success.
- `WH_ERROR_NOTREADY` if the response has not arrived.
- A negative error code on failure.

Error codes:

- `WH_ERROR_BADARGS` if `c` is NULL.
- Server error codes such as `WH_ERROR_NOTFOUND`.

.1.1.3 wh_Client_KeyRevoke Revoke a key using a blocking request/response.

This helper sends a revoke request and waits for the response.

Parameters:

- `c`: Client context.

- keyId: Key ID to revoke.

Return values:

- WH_ERROR_OK on success.
- A negative error code on failure.

Error codes:

- Any error code returned by wh_Client_KeyRevokeRequest() or wh_Client_KeyRevokeResponse().

.1.1.4 wh_Server_KeystoreRevokeKey Revoke a key by updating its metadata.

This server-side function marks a key as non-modifiable and clears all usage flags. If the key exists in NVM, the metadata update is committed so the revoke state persists.

Parameters:

- server: Server context.
- keyId: Key ID to revoke.

Return values:

- WH_ERROR_OK on success.
- A negative error code on failure.

Error codes:

- WH_ERROR_BADARGS if parameters are invalid.
- WH_ERROR_NOTFOUND if the key is missing.
- Propagates NVM/storage errors (for example WH_ERROR_NOSPACE).

.2 wolfhsm/wh_client.h

.2.1 Types

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

.2.2 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_CommInitRequest**(whClientContext * c)Sends a communication initialization request to the server.

	Name
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.
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_CommCloseRequest**(whClientContext * c)Sends a communication close request to the server.
int	**wh_Client_EnableCancel**(whClientContext * c)Enables request cancellation.
int	**wh_Client_DisableCancel**(whClientContext * c)Disables request cancellation.
int	**wh_Client_CancelRequest**(whClientContext * c)Cancels the previous request, currently only supports CMAC. Async Request.

	Name
int	**wh_Client_CancelResponse**(whClientContext * c)Handles the response for a cancellation the previous request, currently only supports CMAC. Async response handler.
int	**wh_Client_Cancel**(whClientContext * c)Cancels the previous request, currently only supports CMAC.
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.
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.

	Name
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.
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.
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.

	Name
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_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.
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.

	Name
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_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_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.

	Name
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.
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.

	Name
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).
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.

	Name
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.
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.

	Name
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).
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_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.

	Name
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.
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.

	Name
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_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.

	Name
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.
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_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.

	Name
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_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.

.2.3 Attributes

	Name
const int[WH_NUM_DEVIDS]	**WH_DEV_IDS_ARRAY**

.2.4 Types Documentation

.2.4.1 enum wc_CipherType

Enumerator	Value	Description
WC_CIPHER_NONE	0	

.2.5 Functions Documentation

.2.5.1 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.

.2.5.2 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

.2.5.3 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.

.2.5.4 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.

.2.5.5 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.

.2.5.6 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.

.2.5.7 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.

.2.5.8 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.

.2.5.9 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.

.2.5.10 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.

.2.5.11 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.

.2.5.12 function wh_Client_EnableCancel

```
int wh_Client_EnableCancel(
    whClientContext * c
)
```

Enables request cancellation.

Parameters:

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

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

This function allows subsequent requests to be canceled, the responses that are normally handled by automatically by wolfCrypt must be handled with a wolfHSM specific function call.

.2.5.13 function wh_Client_DisableCancel

```
int wh_Client_DisableCancel(
    whClientContext * c
)
```

Disables request cancellation.

Parameters:

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

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

This function disables request cancellation, making wolfCrypt automatically handle responses again.

.2.5.14 function wh_Client_CancelRequest

```
int wh_Client_CancelRequest(
    whClientContext * c
)
```

Cancels the previous request, currently only supports CMAC. Async Request.

Parameters:

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

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

This function sends a cancellation request to the server to cancel the previous request made. Does not wait for the response which must be handled separately

.2.5.15 function wh_Client_CancelResponse

```
int wh_Client_CancelResponse(
    whClientContext * c
)
```

Handles the response for a cancellation the previous request, currently only supports CMAC. Async response handler.

Parameters:

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

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

This function handles the response for a request cancellation previously sent to the server. Blocks to wait for the response.

.2.5.16 function wh_Client_Cancel

```
int wh_Client_Cancel(
    whClientContext * c
)
```

Cancels the previous request, currently only supports CMAC.

Parameters:

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

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

This function sends a cancellation request to the server and waits for the response to cancel the previous request made.

.2.5.17 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.

.2.5.18 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.

.2.5.19 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.

.2.5.20 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.

.2.5.21 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.

.2.5.22 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.

.2.5.23 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.

.2.5.24 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.

.2.5.25 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.

.2.5.26 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.

.2.5.27 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.

.2.5.28 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.

.2.5.29 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.

.2.5.30 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.

.2.5.31 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.

.2.5.32 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.

.2.5.33 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.

.2.5.34 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.

.2.5.35 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.

.2.5.36 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.

.2.5.37 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.

.2.5.38 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.

.2.5.39 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.

.2.5.40 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.

.2.5.41 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.

.2.5.42 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.

.2.5.43 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.

.2.5.44 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.

.2.5.45 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.

.2.5.46 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.

.2.5.47 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.

.2.5.48 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.

.2.5.49 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.

.2.5.50 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.

.2.5.51 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.

.2.5.52 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.

.2.5.53 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.

.2.5.54 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.

.2.5.55 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.

.2.5.56 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

.2.5.57 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

.2.5.58 function wh_Client_CounterInitRequest

```
int wh_Client_CounterInitRequest(
    whClientContext * c,
    whNvmId counterId,
    uint32_t counter
)
```

.2.5.59 function wh_Client_CounterInitResponse

```
int wh_Client_CounterInitResponse(
    whClientContext * c,
    uint32_t * counter
)
```

.2.5.60 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.

.2.5.61 function wh_Client_CounterResetRequest

```
int wh_Client_CounterResetRequest(
    whClientContext * c,
    whNvmId counterId
)
```

.2.5.62 function wh_Client_CounterResetResponse

```
int wh_Client_CounterResetResponse(
    whClientContext * c,
    uint32_t * counter
)
```

.2.5.63 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.

.2.5.64 function wh_Client_CounterIncrementRequest

```
int wh_Client_CounterIncrementRequest(
    whClientContext * c,
    whNvmId counterId
)
```

.2.5.65 function wh_Client_CounterIncrementResponse

```
int wh_Client_CounterIncrementResponse(
    whClientContext * c,
    uint32_t * counter
)
```

.2.5.66 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.

.2.5.67 function wh_Client_CounterReadRequest

```
int wh_Client_CounterReadRequest(
    whClientContext * c,
```

```
    whNvmId counterId
)
```

.2.5.68 function wh_Client_CounterReadResponse

```
int wh_Client_CounterReadResponse(
    whClientContext * c,
    uint32_t * counter
)
```

.2.5.69 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.

.2.5.70 function wh_Client_CounterDestroyRequest

```
int wh_Client_CounterDestroyRequest(
    whClientContext * c,
    whNvmId counterId
)
```

.2.5.71 function wh_Client_CounterDestroyResponse

```
int wh_Client_CounterDestroyResponse(
    whClientContext * c
)
```

.2.5.72 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.

.2.5.73 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.

.2.5.74 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.

.2.5.75 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.

.2.5.76 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.

.2.5.77 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.

.2.5.78 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.

.2.5.79 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.

.2.5.80 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.

.2.5.81 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.

.2.5.82 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.

.2.5.83 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.

.2.5.84 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.

.2.5.85 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.

.2.5.86 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.

.2.5.87 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.

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.

.2.5.88 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.

.2.5.89 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.

.2.5.90 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.

.2.5.91 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.

.2.5.92 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.

.2.5.93 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.

.2.5.94 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.

.2.5.95 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.

.2.5.96 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.

.2.5.97 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.

.2.5.98 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.

.2.5.99 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.

.2.5.100 function wh_Client_NvmReadDmaRequest

```
int wh_Client_NvmReadDmaRequest(
    whClientContext * c,
    whNvmId id,
    whNvmSize offset,
    whNvmSize data_len,
    uint8_t * data
)
```

.2.5.101 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.

.2.5.102 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.

.2.5.103 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.

.2.5.104 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.

.2.5.105 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.

.2.5.106 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.

.2.5.107 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.

.2.5.108 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.

.2.5.109 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.

.2.5.110 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.

.2.5.111 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.

.2.5.112 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.

.2.5.113 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.

.2.5.114 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.

.2.5.115 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.

.2.5.116 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.

.2.5.117 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.

.2.5.118 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.

.2.5.119 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.

.2.5.120 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.

.2.5.121 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** 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 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.

.2.5.122 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** 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 verify a certificate and receiving the response. It blocks until the entire operation is complete or an error occurs.

.2.5.123 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.

.2.5.124 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** 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 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.

.2.5.125 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** 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 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.

.2.5.126 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.

.2.5.127 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.

.2.5.128 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.

.2.5.129 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.

.2.5.130 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.

.2.5.131 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.

.2.5.132 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.

.2.5.133 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** 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 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.

.2.5.134 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** 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 verify a certificate using DMA and receiving the response. It blocks until the entire operation is complete or an error occurs.

.2.5.135 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.

.2.5.136 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** 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 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.

.2.5.137 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** 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 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.

.2.5.138 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.

.2.5.139 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** 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 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.

.2.5.140 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** 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 verify an attribute certificate and receiving the response. It blocks until the entire operation is complete or an error occurs.

.2.5.141 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.

.2.5.142 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** 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 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.

.2.5.143 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.

.2.5.144 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.

.2.5.145 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.

.2.5.146 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** 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 DMA request to verify an attribute certificate and receiving the response. It blocks until the entire operation is complete or an error occurs.

.2.6 Attributes Documentation

.2.6.1 variable WH_DEV_IDS_ARRAY

```
const int[WH_NUM_DEVIDS] WH_DEV_IDS_ARRAY;
```

.2.7 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, the device id WH_DEV_ID should be
 * passed into any of the wolfCrypt init functions.
 *
 * 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"


/* 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"

/* Device Id to be registered and passed to wolfCrypt functions */
enum WH_CLIENT_DEVID_ENUM {
    WH_DEV_ID = 0x5748534D, /* "WHSM" */
#ifdef WOLFHSM_CFG_DMA
```

```
    WH_DEV_ID_DMA = 0x57444D41, /* "WDMA" */
    WH_NUM_DEVIDS = 2
#else
    WH_NUM_DEVIDS = 1
#endif
};
extern const int WH_DEV_IDS_ARRAY[WH_NUM_DEVIDS];
#else
/*  for compile purpose */
#define WH_DEV_ID -2 /* invalid ID */
/* cipher types */
enum wc_CipherType {
    WC_CIPHER_NONE = 0,
};
#endif

#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 */
} whClientDmaConfig;

typedef struct {
    whClientDmaClientMemCb    cb;
    const whDmaAddrAllowList* dmaAddrAllowList; /* allowed addresses */
    void* heap; /* heap hint for using static memory (or other allocator) */
} whClientDmaContext;
#endif /* WOLFHSM_CFG_DMA */

typedef int (*whClientCancelCb)(uint16_t cancelSeq);

/* Client context */
struct whClientContext_t {
    uint16_t     last_req_id;
    uint16_t     last_req_kind;
#ifdef WOLFHSM_CFG_CANCEL_API
    uint8_t          cancelable;
    whClientCancelCb cancelCb;
#endif
#ifdef WOLFHSM_CFG_DMA
    whClientDmaContext dma;
#endif /* WOLFHSM_CFG_DMA */
    whCommClient comm[1];
};

struct whClientConfig_t {
    whCommClientConfig* comm;
#ifdef WOLFHSM_CFG_CANCEL_API
```

```
    whClientCancelCb cancelCb;
#endif
#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_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_CommCloseRequest(whClientContext* c);

#ifdef WOLFHSM_CFG_CANCEL_API
int wh_Client_EnableCancel(whClientContext* c);

int wh_Client_DisableCancel(whClientContext* c);

int wh_Client_CancelRequest(whClientContext* c);
int wh_Client_CancelResponse(whClientContext* c);
int wh_Client_Cancel(whClientContext* c);
#endif /* WOLFHSM_CFG_CANCEL_API */

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_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);
#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_DataUnwrap(whClientContext* ctx, enum wc_CipherType cipherType,
                         uint16_t serverKeyId, void* wrappedDataIn,
                         uint32_t wrappedDataInSz, void* dataOut,
                         uint32_t* dataInOutSz);

/* 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);

/* 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);
```

```
#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);


#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_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_WRAPPED_META(_clientId, _id) \
    WH_MAKE_KEYID(WH_KEYTYPE_WRAPPED, (_clientId), (_id))

#endif /* !WOLFHSM_WH_CLIENT_H_ */
```

.3 wolfhsm/wh_client_crypto.h

.3.1 Functions

	Name
int	**wh_Client_RngGenerate**(whClientContext * ctx, uint8_t * out, uint32_t size)Generate random bytes.
int	**wh_Client_RngGenerateDma**(whClientContext * ctx, uint8_t * out, uint32_t size)Generate random bytes using DMA.
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_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.
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)
int	**wh_Client_EccExportKey**(whClientContext * ctx, whKeyId keyId, ecc_key * key, uint16_t label_len, uint8_t * label)

	Name
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 * out_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_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.
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.

	Name
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.
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_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_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.

	Name
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)
int	**wh_Client_AesEcb**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)
int	**wh_Client_AesCbc**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)
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)Performs an AES-GCM operation using DMA.

	Name
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_CmacCancelableResponse**(whClientContext * c, Cmac * cmac, uint8_t * out, uint16_t * outSz)Handle cancelable CMAC response.
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_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_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_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_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_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_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_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.

	Name
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_MlDsaSetKeyId**(MlDsaKey * key, whKeyId keyId)Associates a ML-DSA key with a specific key ID.
int	**wh_Client_MlDsaGetKeyId**(MlDsaKey * key, whKeyId * outId)Gets the wolfHSM keyId being used by the wolfCrypt struct.
int	**wh_Client_MlDsaImportKey**(whClientContext * ctx, 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, MlDsaKey * key, uint16_t label_len, uint8_t * label)Export a ML-DSA key from the server.
int	**wh_Client_MlDsaMakeExportKey**(whClientContext * ctx, int level, int size, 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.
int	**wh_Client_MlDsaSign**(whClientContext * ctx, const byte * in, word32 in_len, byte * out, word32 * out_len, MlDsaKey * key)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, MlDsaKey * key)Verify a ML-DSA signature.
int	**wh_Client_MlDsaCheckPrivKey**(whClientContext * ctx, MlDsaKey * key, const byte * pubKey, word32 pubKeySz)Check a ML-DSA private key.
int	**wh_Client_MlDsaImportKeyDma**(whClientContext * ctx, 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, MlDsaKey * key, uint16_t label_len, uint8_t * label)Export a ML-DSA key using DMA.
int	**wh_Client_MlDsaMakeExportKeyDma**(whClientContext * ctx, int level, MlDsaKey * key)Generate a new ML-DSA key pair and export it using DMA.

	Name
int	**wh_Client_MlDsaSignDma**(whClientContext * ctx, const byte * in, word32 in_len, byte * out, word32 * out_len, MlDsaKey * key)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, MlDsaKey * key)Verify a ML-DSA signature with DMA.
int	**wh_Client_MlDsaCheckPrivKeyDma**(whClientContext * ctx, MlDsaKey * key, const byte * pubKey, word32 pubKeySz)Check a ML-DSA private key against public key with DMA.

.3.2 Functions Documentation

.3.2.1 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 the where the bytes are to be placed. May 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

.3.2.2 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.

.3.2.3 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

.3.2.4 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.

.3.2.5 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.

.3.2.6 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.

.3.2.7 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.

.3.2.8 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

.3.2.9 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.

.3.2.10 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

.3.2.11 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.

.3.2.12 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
)
```

.3.2.13 function wh_Client_EccExportKey

```
int wh_Client_EccExportKey(
    whClientContext * ctx,
    whKeyId keyId,
    ecc_key * key,
    uint16_t label_len,
    uint8_t * label
)
```

.3.2.14 function wh_Client_EccMakeExportKey

```
int wh_Client_EccMakeExportKey(
    whClientContext * ctx,
    int size,
    int curveId,
    ecc_key * key
)
```

.3.2.15 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
)
```

.3.2.16 function wh_Client_EccSharedSecret

```
int wh_Client_EccSharedSecret(
    whClientContext * ctx,
    ecc_key * priv_key,
    ecc_key * pub_key,
    uint8_t * out,
    uint16_t * out_size
)
```

.3.2.17 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
)
```

.3.2.18 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
)
```

.3.2.19 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.

.3.2.20 function wh_Client_Ed25519GetKeyId

```
int wh_Client_Ed25519GetKeyId(
    ed25519_key * key,
    whKeyId * outId
)
```

Retrieves the key ID from an Ed25519 key device context.

.3.2.21 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.

.3.2.22 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.

.3.2.23 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.

.3.2.24 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.

.3.2.25 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.

.3.2.26 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.

.3.2.27 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.

.3.2.28 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.

.3.2.29 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.

.3.2.30 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.

.3.2.31 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.

.3.2.32 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.

.3.2.33 function wh_Client_RsaMakeExportKey

```
int wh_Client_RsaMakeExportKey(
    whClientContext * ctx,
    uint32_t size,
    uint32_t e,
    RsaKey * rsa
)
```

.3.2.34 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
)
```

.3.2.35 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
)
```

.3.2.36 function wh_Client_RsaGetSize

```
int wh_Client_RsaGetSize(
    whClientContext * ctx,
    const RsaKey * key,
    int * out_size
)
```

.3.2.37 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.

.3.2.38 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

.3.2.39 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.

.3.2.40 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.

.3.2.41 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

.3.2.42 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.

.3.2.43 function wh_Client_AesCtr

```
int wh_Client_AesCtr(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
```

```
    uint32_t len,
    uint8_t * out
)
```

.3.2.44 function wh_Client_AesEcb

```
int wh_Client_AesEcb(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    uint8_t * out
)
```

.3.2.45 function wh_Client_AesCbc

```
int wh_Client_AesCbc(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    uint8_t * out
)
```

.3.2.46 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
)
```

.3.2.47 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.

.3.2.48 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.

.3.2.49 function wh_Client_CmacCancelableResponse

```
int wh_Client_CmacCancelableResponse(
    whClientContext * c,
    Cmac * cmac,
    uint8_t * out,
    uint16_t * outSz
)
```

Handle cancelable CMAC response.

Parameters:

- **c** Pointer to the client context structure.
- **cmac** Pointer to the CMAC key structure.
- **out** Buffer to store the CMAC result, only required after wc_CmacFinal.
- **outSz** Pointer to the size of the out buffer in bytes, will be set to the size returned by the server on return.

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

This function handles a CMAC operation response from the server when cancellation has been enabled, since wolfCrypt won't automatically block and wait for the response. Note that DMA-based CMAC operations are NOT cancellable and if a cancel is requested, the cancellation will be aborted.

.3.2.50 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

.3.2.51 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.

.3.2.52 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.

.3.2.53 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.

.3.2.54 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.

.3.2.55 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.

.3.2.56 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.

.3.2.57 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.

.3.2.58 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.

.3.2.59 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.

.3.2.60 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.

.3.2.61 function wh_Client_MlDsaSetKeyId

```
int wh_Client_MlDsaSetKeyId(
    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

.3.2.62 function wh_Client_MlDsaGetKeyId

```
int wh_Client_MlDsaGetKeyId(
    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.

.3.2.63 function wh_Client_MlDsaImportKey

```
int wh_Client_MlDsaImportKey(
    whClientContext * ctx,
    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.

.3.2.64 function wh_Client_MlDsaExportKey

```
int wh_Client_MlDsaExportKey(
    whClientContext * ctx,
    whKeyId keyId,
    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.

.3.2.65 function wh_Client_MlDsaMakeExportKey

```
int wh_Client_MlDsaMakeExportKey(
    whClientContext * ctx,
    int level,
    int size,
    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.

.3.2.66 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.

.3.2.67 function wh_Client_MlDsaSign

```
int wh_Client_MlDsaSign(
    whClientContext * ctx,
    const byte * in,
    word32 in_len,
    byte * out,
    word32 * out_len,
    MlDsaKey * key
)
```

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.

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.

.3.2.68 function wh_Client_MlDsaVerify

```
int wh_Client_MlDsaVerify(
    whClientContext * ctx,
    const byte * sig,
    word32 sig_len,
    const byte * msg,
    word32 msg_len,
    int * res,
    MlDsaKey * key
)
```

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.

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

This function verifies a ML-DSA signature using the HSM.

.3.2.69 function wh_Client_MlDsaCheckPrivKey

```
int wh_Client_MlDsaCheckPrivKey(
    whClientContext * ctx,
    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.

.3.2.70 function wh_Client_MlDsaImportKeyDma

```
int wh_Client_MlDsaImportKeyDma(
    whClientContext * ctx,
    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.

.3.2.71 function wh_Client_MlDsaExportKeyDma

```
int wh_Client_MlDsaExportKeyDma(
    whClientContext * ctx,
    whKeyId keyId,
    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.

.3.2.72 function wh_Client_MlDsaMakeExportKeyDma

```
int wh_Client_MlDsaMakeExportKeyDma(
    whClientContext * ctx,
    int level,
    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.

.3.2.73 function wh_Client_MlDsaSignDma

```
int wh_Client_MlDsaSignDma(
    whClientContext * ctx,
    const byte * in,
    word32 in_len,
    byte * out,
    word32 * out_len,
    MlDsaKey * key
)
```

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.

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

This function signs a message using ML-DSA with DMA.

.3.2.74 function wh_Client_MlDsaVerifyDma

```
int wh_Client_MlDsaVerifyDma(
    whClientContext * ctx,
    const byte * sig,
    word32 sig_len,
    const byte * msg,
    word32 msg_len,
    int * res,
    MlDsaKey * key
)
```

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.

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

This function verifies a ML-DSA signature with DMA.

.3.2.75 function wh_Client_MlDsaCheckPrivKeyDma

```
int wh_Client_MlDsaCheckPrivKeyDma(
    whClientContext * ctx,
    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.

.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_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>

/* 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/dilithium.h"
#include "wolfssl/wolfcrypt/hmac.h"

int wh_Client_RngGenerate(whClientContext* ctx, uint8_t* out, uint32_t size);

#ifdef WOLFHSM_CFG_DMA
int wh_Client_RngGenerateDma(whClientContext* ctx, uint8_t* out, uint32_t
 ↪  size);
#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_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);

#endif /* HAVE_CURVE25519 */

#ifdef HAVE_ECC
int wh_Client_EccSetKeyId(ecc_key* key, whKeyId keyId);

int wh_Client_EccGetKeyId(ecc_key* key, whKeyId* outId);

/* TODO: Send key to server */
int wh_Client_EccImportKey(whClientContext* ctx, ecc_key* key,
        whKeyId *inout_keyId, whNvmFlags flags,
        uint16_t label_len, uint8_t* label);
/* TODO: Recv key from server */
int wh_Client_EccExportKey(whClientContext* ctx, whKeyId keyId,
        ecc_key* key,
        uint16_t label_len, uint8_t* label);

/* TODO: Server creates and exports a key, without caching */
int wh_Client_EccMakeExportKey(whClientContext* ctx,
        int size, int curveId, ecc_key* key);
/* TODO: Server creates and imports the key to cache. */
int wh_Client_EccMakeCacheKey(whClientContext* ctx,
        int size, int curveId,
        whKeyId *inout_key_id, whNvmFlags flags,
        uint16_t label_len, uint8_t* label);

/* TODO: Perform shared secret computation (ECDH) */
int wh_Client_EccSharedSecret(whClientContext* ctx,
                              ecc_key* priv_key, ecc_key* pub_key,
                              uint8_t* out, uint16_t *out_size);

/* TODO: Server generates signature of input hash */
```

```
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);

/* TODO: Server verifies the signature of the provided hash */
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);

#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_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);

/* 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);


#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);
#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);
#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);
#endif /* HAVE_AES_CBC */

#ifdef HAVE_AESGCM
/* TODO: Add documentation */
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);

#ifdef WOLFHSM_CFG_DMA
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);
#endif /* WOLFHSM_CFG_DMA */
#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_CmacCancelableResponse(whClientContext* c, Cmac* cmac,
    uint8_t* out, uint16_t* outSz);

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);
#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_Sha256Dma(whClientContext* ctx, wc_Sha256* sha, const uint8_t*
 ↪  in,
                        uint32_t inLen, uint8_t* out);

#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_Sha224Dma(whClientContext* ctx, wc_Sha224* sha, const uint8_t*
 ↪  in,
                        uint32_t inLen, uint8_t* out);
#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_Sha384Dma(whClientContext* ctx, wc_Sha384* sha, const uint8_t*
 ↪  in,
```

```
                        uint32_t inLen, uint8_t* out);

#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_Sha512Dma(whClientContext* ctx, wc_Sha512* sha, const uint8_t*
 ↪  in,
                        uint32_t inLen, uint8_t* out);
#endif /* WOLFSSL_SHA512 */

#ifdef HAVE_DILITHIUM

int wh_Client_MlDsaSetKeyId(MlDsaKey* key, whKeyId keyId);

int wh_Client_MlDsaGetKeyId(MlDsaKey* key, whKeyId* outId);

int wh_Client_MlDsaImportKey(whClientContext* ctx, MlDsaKey* key,
                             whKeyId* inout_keyId, whNvmFlags flags,
                             uint16_t label_len, uint8_t* label);

int wh_Client_MlDsaExportKey(whClientContext* ctx, whKeyId keyId, MlDsaKey*
 ↪  key,
                             uint16_t label_len, uint8_t* label);

int wh_Client_MlDsaMakeExportKey(whClientContext* ctx, int level, int size,
                                 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, MlDsaKey* key);

int wh_Client_MlDsaVerify(whClientContext* ctx, const byte* sig, word32
 ↪  sig_len,
                          const byte* msg, word32 msg_len, int* res,
                          MlDsaKey* key);

int wh_Client_MlDsaCheckPrivKey(whClientContext* ctx, MlDsaKey* key,
                                const byte* pubKey, word32 pubKeySz);


#ifdef WOLFHSM_CFG_DMA
int wh_Client_MlDsaImportKeyDma(whClientContext* ctx, MlDsaKey* key,
                                whKeyId* inout_keyId, whNvmFlags flags,
                                uint16_t label_len, uint8_t* label);

int wh_Client_MlDsaExportKeyDma(whClientContext* ctx, whKeyId keyId,
                                MlDsaKey* key, uint16_t label_len,
                                uint8_t* label);

int wh_Client_MlDsaMakeExportKeyDma(whClientContext* ctx, int level,
                                    MlDsaKey* key);
```

```
int wh_Client_MlDsaSignDma(whClientContext* ctx, const byte* in, word32 in_len,
                           byte* out, word32* out_len, MlDsaKey* key);

int wh_Client_MlDsaVerifyDma(whClientContext* ctx, const byte* sig,
                             word32 sig_len, const byte* msg, word32 msg_len,
                             int* res, MlDsaKey* key);

int wh_Client_MlDsaCheckPrivKeyDma(whClientContext* ctx, MlDsaKey* key,
                                   const byte* pubKey, word32 pubKeySz);

#endif /* WOLFHSM_CFG_DMA */

#endif /* HAVE_DILITHIUM */

#endif /* !WOLFHSM_CFG_NO_CRYPTO */
#endif /* !WOLFHSM_WH_CLIENT_CRYPTO_H_ */
```

.4 wolfhsm/wh_server.h

.4.1 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_GetCanceledSequence**(whServerContext * server, uint16_t * outSeq)Gets the canceled sequence number of the server.
int	**wh_Server_SetCanceledSequence**(whServerContext * server, uint16_t cancelSeq)Sets the canceled sequence number 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.

	Name
int	**wh_Server_RegisterCustomCb**(whServerContext * server, uint16_t action, whServerCustomCb handler)Registers a custom callback handler for a specific action.
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.

.4.2 Functions Documentation

.4.2.1 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. Note that the NVM and Crypto components of the config structure MUST be initialized before calling this function.

.4.2.2 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.

.4.2.3 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.

.4.2.4 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.

.4.2.5 function wh_Server_GetCanceledSequence

```
int wh_Server_GetCanceledSequence(
    whServerContext * server,
    uint16_t * outSeq
)
```

Gets the canceled sequence number of the server.

Parameters:

- **server** Pointer to the server context.
- **outSeq** Pointer to store the canceled sequence number.

Return: int Returns 0 on success, or WH_ERROR_BADARGS if the arguments are invalid

The canceled sequence number is the comms layer sequence number of the last canceled request. This number is set by the server port in response to an out-of-band signal from the client when the client wishes to cancel a request.

.4.2.6 function wh_Server_SetCanceledSequence

```
int wh_Server_SetCanceledSequence(
    whServerContext * server,
    uint16_t cancelSeq
)
```

Sets the canceled sequence number of the server.

The canceled sequence number is the comms layer sequence number of the last canceled request. This function should be used by the server port to set the canceled sequence number in response to an out-of-band signal from the client.

.4.2.7 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.

.4.2.8 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.

.4.2.9 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.

.4.2.10 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.

.4.2.11 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.

.4.2.12 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.

.4.2.13 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

.4.2.14 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.

.4.2.15 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.

.4.3 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_nvm.h"
#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 {
    int devId;
#ifndef WC_NO_RNG
    WC_RNG rng[1];
#endif
    union {
#if 0
#ifndef NO_AES
        Aes aes[1];
#endif
#ifndef NO_RSA
        RsaKey rsa[1];
#endif
#ifdef HAVE_CURVE25519
        curve25519_key curve25519Private[1];
#endif
#endif /* 0 */
#ifdef WOLFSSL_CMAC
        Cmac cmac[1];
#endif
    } algoCtx;
} 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 */
```

```c
typedef struct whServerConfig_t {
    whCommServerConfig* comm_config;
    whNvmContext*       nvm;

#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 */
} whServerConfig;


/* Context structure to maintain the state of an HSM server */
struct whServerContext_t {
    whNvmContext* nvm;
    whCommServer  comm[1];
#ifndef WOLFHSM_CFG_NO_CRYPTO
    whServerCryptoContext* crypto;
    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_CANCEL_API
    uint16_t           cancelSeq;
#endif
#ifdef WOLFHSM_CFG_LOGGING
    whLogContext log;
#endif /* WOLFHSM_CFG_LOGGING */
};


/* Initialize the comms and crypto cache components.
 * Note: NVM and Crypto components must be initialized prior to Server Init
 */

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);

#ifdef WOLFHSM_CFG_CANCEL_API
int wh_Server_GetCanceledSequence(whServerContext* server, uint16_t* outSeq);


int wh_Server_SetCanceledSequence(whServerContext* server, uint16_t cancelSeq);
#endif /* WOLFHSM_CFG_CANCEL_API */

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 */

#endif /* !WOLFHSM_WH_SERVER_H_ */
```