wolfHSM Documentation

2025-04-30

Contents

0.1 Introduction . . . 2
0.1.1 Why Choose wolfHSM? . . . 3
0.2 Overview . . . 3
0.2.1 Features . . . 3
0.2.2 Architecture . . . 4
0.2.3 Ports . . . 4
0.3 Getting Started With wolfHSM . . . 5
0.3.1 Basic Client Configuration . . . 5
0.3.2 Basic Server Configuration . . . 6
0.4 Library Design / wolfHSM Internals . . . 9
0.4.1 Table of Contents: . . . 9
0.4.2 Generic Component Architecture . . . 9
0.4.3 Communications . . . 10
0.4.4 Non Volatile Memory . . . 12
0.4.5 Key Management . . . 14
0.4.6 Cryptographic Operations . . . 14
0.4.7 AUTOSAR SHE . . . 15
0.5 wolfHSM Client Library . . . 15
0.5.1 Table of Contents . . . 15
0.5.2 API Return Codes . . . 15
0.5.3 Split Transaction Processing . . . 15
0.5.4 The Client Context . . . 16
0.5.5 NVM Operations . . . 18
0.5.6 Key Management . . . 20
0.5.7 Cryptography . . . 20
0.5.8 AUTOSAR SHE API . . . 22
0.6 wolfHSM Server Library . . . 22
0.6.1 Getting Started . . . 22
0.6.2 Architecture . . . 22
0.6.3 API Reference . . . 22
0.6.4 Key Management . . . 22
0.6.5 Cryptographic . . . 22
0.7 Customizing wolfHSM . . . 23
0.7.1 Library Configuration . . . 23
0.7.2 DMA Callbacks . . . 23
0.7.3 DMA Address Allow List . . . 26
0.7.4 Custom Callbacks . . . 27
0.8 WolfHSM Porting . . . 31
0.8.1 WolfHSM Porting Overview . . . 31
0.8.2 WolfHSM Ports . . . 32
0.8.3 WolfHSM Porting Interface . . . 32

A wolfHSM API reference 33

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 */
```

```c
/* 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.

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

/* 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:

- Generic Component Architecture
- Communications
  - Key Components
    - Client/Server APIs
    - Comms Layer
- Non Volatile Memory
  - NVM Metadata
  - NVM Architecture
  - NVM Back-Ends
- Key Management
- Cryptographic Operations
  - Hardware Cryptography Support

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:

```c
/* 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.

```c
/* 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 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.

0.4.4.3 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.

0.4.6 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

- API Return Codes
- Split Transaction Processing
- The Client Context
  - Initializing the client context
- NVM Operations
- Key Management
- Cryptography
- AUTOSAR SHE API

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).

```
#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.

```
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.

```
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 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.

0.5.7 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`. See supported wolfCrypt algorithms for the full list of algorithms wolfHSM supports for remote HSM execution.

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

```
#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:

```
#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.

0.5.7.1 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.

0.5.8 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 e`whMessageCustomCb_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 */
```

```c
#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.

```c
#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 (Port in progress) 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 (Port in progress) 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

A wolfHSM API reference

