wolfHSM Documentation

2026-04-24

Contents

0.1 イントロダクション

このマニュアルは、wolfHSM 組込みハードウェアセキュリティモジュールライブラリの技術ガイドとして作成されています。wolfHSM の構築方法と使用開始方法を説明し、構築オプション、機能、移植性の向上、サポートなどの概要を示します。

このドキュメントの PDF 版は こちら で入手できます。

0.1.1 wolfHSM を選択する理由

車載 HSM(Hardware Security Module) は、暗号鍵と処理のセキュリティを大幅に向上させます。これは、セキュリティの基盤である署名検証と暗号処理を物理的に独立したプロセッサに分離することで実現します。堅牢なセキュリティが求められる ECU では、推奨ではなく必須とされることがよくあります。よって、厳密なテストを行っている暗号化ライブラリ wolfSSL を、Aurix Tricore TC3XX などの広く使用されている自動車用 HSM でシームレスに動作するようにしました。wolfHSM が依存しているのは wolfCrypt のみであり、ほぼすべての環境に移植できます。また、ユーザーフレンドリーなクライアント インターフェイスも提供しており、wolfCrypt API を直接利用できます。

wolfHSM は ECU のセキュリティとパフォーマンスを最大限に高めるべく、ハードウェア暗号化、不揮発性メモリ、各種隔離環境などと組み合わせて使用できるように設計しています。暗号エンジン wolfCrypt を Infineon Aurix Tricore TC3XX などのハードウェア HSM に統合することで、SM2、SM3、SM4 などの中国政府義務アルゴリズムも利用できます。さらに、Kyber、LMS、XMSS などの耐量子暗号アルゴリズムを自動車ユーザーが簡単に利用できるようにし、顧客の要件に対応できるようにします。併せて、HSM でハードウェア暗号処理を利用できる場合は、これも利用してパフォーマンスを向上させます。

wolfBoot は、ベアメタル環境のための安全かつポータブルなブートローダーソリューションです。最小限の設計と小さな HAL API を活用した包括的なファームウェア認証および更新メカニズムを提供し、オペレーティングシステムやベアメタルアプリケーションから完全に独立しています。wolfBoot は、フラッシュインターフェイスとプリブート環境を効率的に管理、アプリケーションを正確に測定および認証します。また、必要に応じてよりレイヤーの低いハードウェア暗号化も利用します。wolfBoot は wolfHSM クライアントを使用し、HSM 支援アプリケーションコアのセキュアブートを使用できます。さらに、wolfBoot を HSM コア上で実行して HSM サーバーが損傷していないことを確認し、2 次的な保護層を提供できます。この設定により、安全なブートシーケンスが保証され NVM サポートに依存する HSM コアのブートプロセスと適切に連携します。

0.2 概要

wolfHSM は、暗号操作・鍵管理・不揮発性ストレージの制御など、HSM 操作用の統合 API を提供するソフトウェアフレームワークです。HSM アプリケーションに関連するコードの移植性を向上させるように設計しており、ベンダー固有のライブラリ呼び出しに縛られることなく、強力なセキュリティ機能を多くのハードウェア上で容易に使用できるようにします。wolfCrypt API を直接使用できるため、クライアントアプリケーションが大幅に簡素化されます。ライブラリは、クライアントアプリで追加のロジックを必要とせずに、すべての機密性の高い暗号化操作をリモートプロシージャコールとして HSM コアに自動的にオフロードします。

当初、主に自動車に搭載される HSM 対応マイクロコントローラを対象としていました。しかし現在では、自動車に限らずあらゆるユースケースにおいて PKCS11 や AUTOSAR SHE などの標準化されたインターフェイスとプロトコルをサポートできるようにしました。プラットフォームに将来追加される機能をサポー

トすることのできる、拡張可能なソリューションです。wolfCrypt 以外の外部依存はなく、ほぼすべての実行環境に移植可能です。

0.2.1 機能

- ユーザーベースの権限による安全な不揮発性オブジェクトストレージ
- ハードウェアキーをサポートする暗号鍵管理
- 互換性のあるデバイスに対するハードウェア暗号化サポート
- 完全に非同期のクライアント API
- 柔軟なコールバックアーキテクチャにより、ライブラリを変更せずにカスタムユースケースを実現
- wolfCrypt API をクライアントで直接使用し、HSM コアに自動的にオフロード
- 信頼チェーンをサポートするイメージマネージャー
- AUTOSAR との統合
- SHE+ との統合
- PKCS11 インターフェイスを使用可能
- TPM 2.0 インターフェイスを使用可能
- Secure OnBoard Communication (SecOC) モジュール統合を使用可能
- 証明書の処理
- 共通鍵暗号・公開鍵暗号の両方に対応
- 半導体ベンダーによって実装されたアルゴリズムだけでなく、wolfCrypt に実装されているすべてのアルゴリズムを提供することで、「暗号アジリティ」をサポート
- FIPS 140-3 認証を取得可能

0.2.2 アーキテクチャ

wolfHSM は、サーバーが信頼できる安全な環境 (通常は安全なコプロセッサ上) で実行され、クライアントがライブラリであるクライアントサーバーモデルを採用しています。このアーキテクチャにより、機密性の高い暗号化操作と鍵管理がサーバー内で安全に処理され、クライアントライブラリがサーバーとのレイヤーの低い通信を抽象化します。

- サーバー: wolfHSM のサーバーコンポーネントは、HSM コアで実行されるスタンドアロンアプリケーションです。安全な環境内で暗号化操作、鍵管理、不揮発性ストレージを処理します。サーバーはクライアントからの要求を処理し、結果を返す役割を担います。
- クライアント: wolfHSM のクライアントコンポーネントは、ユーザーアプリケーションにリンクできるライブラリです。サーバーに要求を送信し、応答を受信するための API を提供します。クライアントは、通信の複雑さを抽象化し、アプリケーションが HSM と安全かつ効率的に対話できるようにします。

0.2.3 ポート

wolfHSM 自体には特定のハードウェアと対話するためのコードは含まれておらず、単体では実行できません。wolfHSM をデバイスで実行するには、サーバ アプリケーションがクライアントと通信して実行できるように、必要なハードウェアドライバーと抽象化レイヤーを使用してライブラリを構成する必要があります。具体的には、次の要素が必要です。

- サーバーアプリケーションの起動とハードウェアの初期化
- サーバーにおける wolfCrypt 構成
- サーバーの不揮発性メモリ構成
- サーバーとクライアントのトランスポート構成
- サーバーとクライアントの接続処理

これらの要件を提供し、サーバー API を起動可能なアプリケーションにラップするコードを、wolfHSM「ポート」と呼んでいます。

wolfHSM の公式ポートは、サポートされているさまざまなアーキテクチャ用に提供しています。それぞれのポートには次のものが含まれます。

- スタンドアロンリファレンスサーバーアプリケーション: このアプリケーションは、HSM コアで実行され、すべての安全な操作を処理することを目的としています。すぐに使用できる状態で完全に機能しますが、エンドユーザーがカスタマイズして追加のユースケースをサポートすることもできます。
- クライアントライブラリ: このライブラリは、ユーザーアプリケーションにリンクして、サーバーとの通信を容易にします。

0.2.3.1 対応状況 本章執筆時点において、次のデバイス/環境のポートをサポートしています。

- POSIX ランタイム
- ST Micro SPC58N*
- Infineon Aurix TC3xx*
- Infineon Aurix TC4xx* (近日対応予定)
- Infineon Traveo T2G* (近日対応予定)
- Renesas RH850* (近日対応予定)
- NXP S32* (近日対応予定)

ここに示した環境のほか、いくつかの環境へのポートを現在準備中です。

- 残念ながら、これらのポートでは、HSM コアに関する情報を取得するためにシリコンベンダーとの NDA が必要です。したがって、これらのプラットフォームの wolfHSM ポートは一般には公開しておりません。ご興味がございましたら、info@wolfssl.jp までお問い合わせください。

0.3 wolfHSM の始め方

wolfHSM の最も一般的な使用例は、HSM コプロセッサを搭載したマルチコアデバイスのアプリケーションコアの 1 つで動作する既存のアプリケーションに、HSM 対応の機能を追加することです。

デバイス上で wolfHSM を実行するために必要な最初のステップは、特定の wolfHSM ポートの手順に従って、HSM コア上でリファレンスサーバーを実行することです。wolfHSM サーバーアプリケーションがデバイスにロードされて起動すると、クライアントアプリケーションは wolfHSM クライアントライブラリとリンクし、wolfHSM クライアント構造体のインスタンスを設定して、wolfHSM クライアント API と wolfCrypt API を通じて HSM と対話できるようになります。

各 wolfHSM ポートには、デフォルトの通信チャネルの設定方法とサーバーとの対話方法を示すクライアントデモアプリが含まれています。また、サーバーのリファレンス実装はサーバーコールバックによってカスタマイズでき、その機能はクライアントリクエストを通じて呼び出すことができます。

0.3.1 基本的なクライアント設定

wolfHSM クライアントを使用するには、クライアントコンテキスト構造体を割り当て、有効なクライアントのパラメータを用いて初期化処理を実行する必要があります。

クライアントコンテキスト構造体 `whClientContext` は、クライアントの内部状態とサーバーとの通信を保持します。すべてのクライアント API は、クライアントコンテキストへのポインタを受け取ります。

クライアント設定構造体には、サーバーとの通信用にコンテキストを設定・初期化するために使用される通信レイヤー設定（`whCommClientConfig`）が含まれています。`whCommClientConfig` 構造体は、実際のトランスポート実装（組み込みまたはカスタム）をクライアントが使用する抽象的な通信インターフェースにバインドします。

クライアントを設定する一般的な手順は以下の通りです。

1. 目的のトランスポートのトランスポート設定構造体、コンテキスト、コールバック実装を割り当て、初期化
2. 通信クライアント設定構造体を割り当て、手順 1 のトランスポート設定にバインド

3. 手順 2 の通信クライアント設定を使用してクライアント設定構造体を割り当て、初期化
4. クライアントコンテキスト構造体を割り当て
5. wh_Client_Init() を呼び出してクライアント設定でクライアントを初期化
6. クライアント API を使用してサーバーと接続

以下に、クライアントアプリケーション用の実装例を示します。ここでは、組み込みの共有メモリトランスポートレイヤを使用して、サーバーにエコーリクエストを送信します。

```
#include <string.h> /* for memcmp() */
#include "wolfhsm/client.h"  /* クライアント API（通信設定を含む） */
#include "wolfhsm/wh_transport_mem.h" /* トランスポート実装 */

/* 手順 1: 共有メモリトランスポート設定の割り当てと初期化 */
/* 共有メモリトランスポート設定 */
static whTransportMemConfig transportMemCfg = { /* 共有メモリ設定 */ };
/* 共有メモリトランスポートコンテキスト（状態） */
whTransportMemClientContext transportMemClientCtx = {0};
/* 抽象的な通信トランスポートインターフェースを
   具体的な実装にバインドするコールバック構造体 */
whTransportClientCb transportMemClientCb = {WH_TRANSPORT_MEM_CLIENT_CB};

/* 手順 2: クライアント通信設定を割り当て、トランスポートにバインド */
/* 選択したトランスポート設定を使用するようにクライアント通信を設定 */
whCommClientConfig commClientCfg = {
            .transport_cb      = transportMemClientCb,
            .transport_context = (void*)transportMemClientCtx,
            .transport_config  = (void*)transportMemCfg,
            .client_id         = 123, /* 一意のクライアント識別子 */
};

/* 手順 3: クライアント設定の割り当てと初期化 */
whClientConfig clientCfg= {
  .comm = commClientCfg,
};

/* 手順 4: クライアントコンテキストの割り当て */
whClientContext clientCtx = {0};

/* 手順 5: 提供された設定でクライアントを初期化 */
wh_Client_Init(&clientCtx, &clientCfg);

/* 手順 6: クライアント API を使用してサーバーと接続 */

/* 送受信データを保持するバッファ */
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;

/* エコーリクエストを送信し、応答を受信するまでブロック */
wh_Client_Echo(client, sendLen, &sendBuffer, &recvLen, &recvBuffer);
```



```
if ((recvLen != sendLen ) ||
    (0 != memcmp(sendBuffer, recvBuffer, sendLen))) {
    /* エラー：送信したものと同じものがエコーバックされなかった */
}
```

詳細については、第 5 章：クライアントライブラリをご覧ください。

0.3.2 基本的なサーバ設定

注：*wolfHSM* ポートには、*HSM* コア上で実行するように設定済みのリファレンスサーバーアプリケーションが付属しているため、手動でのサーバー設定は必要ありません。

wolfHSM サーバーを使用するには、サーバーコンテキスト構造体を割り当て、有効なクライアントのパラメータを用いてを初期化処理を実行する必要があります。通常、クライアントとの通信、暗号化操作、鍵の管理、不揮発性オブジェクトの保存が含まれます。ただし、これらの設定コンポーネントをすべて初期化する必要はありません。必要なもののみ初期化します。

クライアント通信、NVM フラッシュ設定を使用した NVM オブジェクトストレージ、およびローカル暗号（ソフトウェアのみ）をサポートするサーバーを設定するために必要な手順を以下に示します。

1. サーバー通信設定の初期化 1. 目的のトランスポートのトランスポート設定構造体、コンテキスト、コールバック実装を割り当て、初期化 2. 手順 1.1 のトランスポート設定を使用して通信サーバー設定構造体を割り当て、初期化
2. サーバー NVM コンテキストの初期化 1. 低レベルフラッシュストレージドライバーの設定、コンテキスト、コールバック構造体を割り当て、初期化（これらの構造体の実装はポートによって提供されます）2. NVM フラッシュ設定、コンテキスト、コールバック構造体を割り当て、初期化し、手順 2.1 のポートフラッシュ設定をそれらにバインド 3. NVM コンテキスト構造体を割り当て、`wh_Nvm_Init()` を使用して手順 2.2 の設定で初期化
3. サーバー用の暗号コンテキスト構造体を割り当て、初期化
4. wolfCrypt を初期化（必ず、サーバーを初期化する前に行います）
5. サーバー設定構造体を割り当て、初期化し、通信サーバー設定、NVM コンテキスト、暗号コンテキストをバインド
6. サーバーコンテキスト構造体を割り当て、`wh_Server_Init()` を使用してサーバー設定で初期化
7. 基盤となるトランスポートがクライアント通信に使用できる準備が整い次第、`wh_Server_SetConnected()` を使用してサーバー接続状態を接続済みに設定（詳細についてはサンプルプログラムを参照ください）
8. `wh_Server_HandleRequestMessage()` を使用してクライアントリクエストを処理

サーバーは、NVM フラッシュ設定を使用して NVM オブジェクトストレージをサポートするように設定できます。手順 1 の後に、サーバーでNVM を初期化する手順を実行してください。

```
#include <string.h> /* for memcmp() */
#include "wolfhsm/server.h"  /* サーバー API（通信設定を含む） */
#include "wolfhsm/wh_transport_mem.h" /* トランスポート実装 */

/* 手順 1.1: 共有メモリトランスポート設定の割り当てと初期化 */
/* 共有メモリトランスポート設定 */
static whTransportMemConfig transportMemCfg = { /* 共有メモリ設定 */ };

/* 共有メモリトランスポートコンテキスト（状態） */
whTransportMemServerContext transportMemServerCtx = {0};

/* 抽象的な通信トランスポートインターフェースを
   具体的な実装にバインドするコールバック構造体 */
whTransportServerCb transportMemServerCb = {WH_TRANSPORT_MEM_SERVER_CB};

/* 手順 1.2: 通信サーバー設定構造体を割り当て、
   トランスポートにバインド */
```

```
/* 選択したトランスポート設定を使用するようにサーバー通信を設定 */
whCommServerConfig commServerCfg = {
       .transport_cb       = transportMemServerCb,
       .transport_context  = (void*)transportMemServerCtx,
       .transport_config   = (void*)transportMemCfg,
       .server_id          = 456, /* 一意のサーバー識別子 */
};

/* サーバー NVM コンテキストの初期化 */

/* 手順 2.1: ポート固有のフラッシュストレージドライバー用の
   コンテキストと設定の割り当てと初期化 */

/* ポートフラッシュコンテキスト（構造体名はポート固有） */
MyPortFlashContext portFlashCtx = {0}

/* ポートフラッシュ設定 */
MyPortFlashConfig portFlashCfg = { /* ポート固有の設定 */ };

/* ポートフラッシュ用の NVM フラッシュコールバック実装 */
whFlashCb portFlashCb = { /* NVM フラッシュコールバックのポートフラッシュ実装 */ };

/* 手順 2.2: NVM フラッシュ設定構造体を割り当て、初期化し、
   手順 2.1 のポート設定にバインド */
whNvmFlashConfig nvmFlashCfg = {
   .cb         = portFlashCb,
   .context    = portFlashCtx,
   .config     = portFlashCfg,
};
whNvmFlashContext nfc = {0};

/* 手順 2.3: NVM コンテキスト、設定、コールバック構造体を割り当て、
   手順 2.2 の NVM フラッシュ設定で初期化 */
whNvmCb nvmFlashCb = {WH_NVM_FLASH_CB};

whNvmConfig nvmConf = {
   .cb      = nvmFlashCb;
   .context = nfc;
   .config  = nvmFlashCfg,
};
whNvmContext nvmCtx = {0};

wh_Nvm_Init(&nvmCtx, &whNvmConfig);

/* 手順 3: 暗号コンテキスト構造体の割り当てと初期化 */
whServerCryptoContext cryptoCtx = {0};

/* サーバー設定の割り当てと初期化 */
whServerConfig serverCfg = {
       .comm   = commServerCfg,
       .nvm    = nvmCtx,
       .crypto = &cryptoCtx,
       .devId  = INVALID_DEVID, /* あるいは、カスタム暗号コールバック devID を設定 */
};
```

```
/* 手順 4: wolfCrypt の初期化 */
wolfCrypt_Init();

/* 手順 5: サーバー設定構造体を割り当て、初期化し、
   通信サーバー設定と暗号コンテキストをバインド */
whServerContext server = {0};
wh_Server_Init(&server, &serverCfg);

/* トランスポートの準備が整い次第（例：共有メモリバッファがクリアされた後）、
   サーバー接続状態を接続済みに設定 */
wh_Server_SetConnected(&server, WH_COMM_CONNECTED);

/* クライアントリクエストの処理 */
while (1) {
   wh_Server_HandleRequestMessage(&server);
}
```

0.4 ライブラリ構造

wolfHSM は、組込みシステム向けに安全で効率的なハードウェアセキュリティモジュール（HSM）API を提供するために設計した、モジュラーで拡張可能なライブラリです。このライブラリは、特定のアプリケーションの要件を満たすために簡単に設定・組み合わせることが可能な一連の機能コンポーネントを中心に構築されています。この章では、コンポーネントアーキテクチャ、通信レイヤー、不揮発性メモリ（NVM）、鍵管理、暗号化操作、ハードウェアセキュリティモジュール（HSM）サポートなど、wolfHSM の主要な機能コンポーネントの概要を説明します。

0.4.1 目次

- コンポーネントアーキテクチャ
- 通信
  - 主要コンポーネント
    - クライアント/サーバー API
    - 通信レイヤー
- 不揮発性メモリ
  - 不揮発性メモリのメタデータ
  - 不揮発性メモリのアーキテクチャ
  - 不揮発性メモリのバックエンド
- 鍵管理
- 暗号処理
  - ハードウェア暗号サポート

0.4.2 コンポーネントアーキテクチャ

wolfHSM の各コンポーネントは、共通の初期化、設定、コンテキストストレージアーキテクチャを持つように設計しています。これは、異なるハードウェアプラットフォームやビルド環境に簡単に移植できるようにするためです。

ハードウェアの詳細は、void* として参照される型付けされていないコンテキスト構造体にコールバック関数を関連付けることで、論理操作から抽象化しています。

0.4.2.1 コンポーネントの初期化

wolfHSM コンポーネントの、一般的な設定と初期化の例を示します。

```
#include "wolfhsm/component.h"          /* wolfHSM コンポーネントの抽象 API リファレン
 ↪  ス */
#include "port/vendor/mycomponent.h"   /* プラットフォーム固有の設定とコンテキスト構造体
 ↪  の定義、
                                        * およびコールバック関数の宣言 */

/* mycomponent 用の関数コールバックのルックアップテーブルを提供。型は
wolfhsm/component.h で提供される抽象型であることに注意 */
whComponentCb my_cb[1] = {MY_COMPONENT_CB};

/* 固定設定データ。関連データは init() の実行中に構造体から
* コピーされることに注意 */
const myComponentConfig my_config = {
    .my_number = 3,
    .my_string = "This is a string",
}

/* myComponent の動的状態の静的割り当て */
myComponentContext my_context[1] = {0};

/* プラットフォーム固有のコールバックを使用したコンポーネントの初期化 */
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 通信

wolfHSM の通信レイヤーは、クライアントとサーバー間において信頼性の高い双方向のパケットベースの通信を提供するように設計しています。このレイヤーは基盤となるトランスポートメカニズムを抽象化し、柔軟性とモジュール性を実現します。クライアントとサーバーの両方でリクエストとレスポンスの機能を分割していることで、メッセージ受信の同期的なポーリングや、割り込み/イベントサポートに基づく非同期処理を可能にしています。

0.4.3.1 主要コンポーネント

- クライアント/サーバー API：クライアントとサーバー間の通信のためのメインインターフェース。これらはユーザーアプリケーションによって直接使用される API です。
- 通信レイヤー：クライアントとサーバー間で交換されるメッセージのフォーマットと構造を定義し、基盤となるトランスポートレイヤーの実装への抽象インターフェースを提供。
- トランスポートレイヤー：基盤となるトランスポートの具体的な実装。クライアントとサーバー間でデータが実際にどのように転送されるかを定義します。

0.4.3.2 クライアント/サーバー API 高レベルのクライアントとサーバー API（`wolfhsm/wh_client.h` と `wolfhsm/wh_server.h` で定義）は、通信のための主要なインターフェースです。これらの関数は、呼び出し元から低レベルの通信の詳細を抽象化し、論理操作のためのシンプルな分割トランザクションインターフェースを提供します。

以下に、クライアント API を使用してサーバーにエコーリクエストを送信する場合の実装例を示します。

```
/* エコーリクエストを送信 */
wh_Client_EchoRequest(&clientCtx, sendLen, &sendBuffer);

/* 任意で他の処理を実行 */

/* サーバーのレスポンスをポーリング */
while (WH_ERROR_NOTREADY == wh_Client_EchoResponse(client, &recv_len,
 ↪  recv_buffer));
```

0.4.3.3 通信レイヤー 通信レイヤーは、より高レベルのクライアントとサーバーの API によって呼び出され、下位レベルのトランスポートとの間でデータを送受信するためのメッセージング構造と制御ロジックをカプセル化します。下位レベルのトランスポートと対話するための抽象インターフェース関数を提供する通信クライアントと通信サーバーの抽象化を実現しています。通信レイヤー API は、リクエストとレスポンスの送受信機能で構成されており、これらのリクエストとレスポンスは高レベルの操作ではなくメッセージに関連します。

各クライアントは、一度に 1 つのリクエストのみをサーバーに送信できます。サーバーはクライアントの分離を確実にするため、一度に 1 つのリクエストを処理します。

0.4.3.3.1 メッセージ メッセージは、可変長のペイロードを持つヘッダーで構成されます。ヘッダーはシーケンス ID とリクエストまたはレスポンスの種類を示します。併せて、補助フラグやセッション情報を提供するための追加フィールドも提供します。

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

typedef struct {
    uint16_t magic;
    uint16_t kind;
    uint16_t seq;
    uint16_t size;
} whCommHeader;
```

メッセージは、サーバーが望ましい機能を実行するために必要なリクエストデータと、機能の実行結果をクライアントに返すためのレスポンスをカプセル化するために使用されます。メッセージタイプは、機能を実行するコンポーネントに基づいてグループ化され、列挙された機能のどれが実行されているかを一意に識別します。互換性（エンディアンとバージョン）を確保するためにメッセージには Magic フィールドが含まれており、ペイロード内で渡されるデータをネイティブ処理用にデマーシャリングするために必要な操作を示す既知の値が使用されます。各機能コンポーネントには、ネイティブ値と「on-the-wire」メッセージフォーマット間の変換を行うリモート実装があります。サーバーは、レスポンスフォーマットがリクエストフォーマットと一致することを保証します。

メッセージ内でデータコンテンツを渡すことに加えて、特定のメッセージタイプは共有またはマップされたメモリポインタの受け渡しもサポートしています。これは、特にサーバーコンポーネントが DMA 方式でデータに直接アクセスできる可能性のある、パフォーマンスクリティカルな操作のためのものです。整数ポインタサイズ（IPS）と `size_t` の違いを避けるため、可能な場合はすべてのポインタとサイズを `uint64_t` として送信する必要があります。

メッセージは、ヘッダーの Magic フィールドを使用して「on-the-wire」フォーマットでエンコードされ、構造体メンバーの指定されたエンディアンと通信ヘッダーのバージョン（現在は 0x01）が示されます。リクエストメッセージを処理するサーバーコンポーネントは、提供された値をネイティブフォーマットに変換し、タスクを実行し、結果をリクエストのフォーマットに再エンコードします。クライアントでは、リクエストフォーマットと一致しないメッセージを処理する必要はありません。エンコードされたメッセージは、Magic フィールドで指定されたエンディアンで、かつネイティブ構造体と同じサイズとレイアウトを想定しています。

以下に、クライアント通信レイヤーがリクエストを送信する際の実装例を示します。

```
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);
/* 他のタスクを実行 */

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) {
        /* 他のタスクを実行 or yield */
}
```

メッセージレイヤーに渡されるトランスポートエラーは致命的であると想定されます。クライアント/サーバーはコンテキストをクリーンアップする必要があることに注意してください。

0.4.3.4 トランスポート

トランスポートは、ライブラリがリクエストまたはレスポンスとして処理するために、可変サイズ（最大 MTU まで）の完全なパケット（バイト列）をメッセージレイヤーに提供します。whTransportClientCb で定義された抽象インターフェースを実装し、データの送受信が必要な時に commClient/commServer によって直接呼び出されます。

whTransportClientCb インターフェースを実装するカスタムトランスポートモジュールは、サーバーとクライアントに登録できます。その後、標準的なサーバーとクライアントのリクエスト/レスポンス機能を介して自動的に使用されます。

メモリバッファトランスポートモジュールと POSIX TCP ソケットトランスポートの実装例は、wolfHSM でサポートしているトランスポートの中で見ることができます。

0.4.3.4.1 サポートしているトランスポート

wolfHSM は、以下 2 つの組み込みトランスポートを付属しています。

- メモリバッファトランスポート（wh_transport_mem.c）
- POSIX TCP ソケットトランスポート（port/posix_transport_tcp.c）

メモリトランスポートは、ほとんどの組込み wolfHSM ポートのデフォルトトランスポートであり、wolfHSM コアライブラリの一部です。クライアントとサーバー間の共有メモリブロックを使用して、トランスポートコールバックの具体的な実装を提供します。共有メモリトランスポートメカニズムは、2 つのメモリブロックを割り当てることで動作します。1 つは受信リクエスト用、もう 1 つは送信レスポンス用です。クライアントは受信メモリブロックにリクエストを書き込み、送信メモリブロックからレスポンスを読み取ります。サーバーは受信メモリブロックからリクエストを読み取り、送信メモリブロックにレスポンスを書き込みます。各ブロックには、使用準備が整ったときに消費者に通知する制御とステータスフラグが含まれています。このメカニズムは、システムコールやネットワーク通信の必要性を回避するため、高速かつ効率的になるように設計されています。

POSIX TCP トランスポートは、wolfHSM POSIX ポートの一部です。クライアントとサーバー間のデータのトランスポート媒体として TCP ソケットを使用します。ソケットは IPv4 のみで、ノンブロッキングです。

0.4.4 不揮発性メモリ

wolfHSM における不揮発性メモリ（NVM）は、メタデータとデータブロックを持つ永続的なオブジェクトを管理するために使用されます。NVM ライブラリは、成功を返す前にトランザクションが完全にコミット

されることを保証する、信頼性の高い原子的な操作を確保します。主要な操作には、オブジェクトの追加、一覧表示、読み取り、破棄、および関連するメタデータの取得が含まれます。

NVM の主要な機能として、以下が含まれます。

- アクセス可能な NVM 内の可変サイズデータとメタデータ（ID、ラベル、長さ、アクセス、フラグ）を関連付けるための API
- ステータスフラグを持つ 2 つの消去可能なパーティションを使用して、常に復元可能
- オブジェクトは次のエントリを使用して追加され、空き領域にプログラムされる
- 重複した ID は許可されるが、最新のものだけが読み取り可能
- オブジェクトは、リストされたオブジェクトを除いて、非アクティブなパーティションに全領域をコピーすることで破棄される
- 復旧時に後のオブジェクトを識別するため、内部エポックカウンタを使用

0.4.4.1 不揮発性メモリのメタデータ

不揮発性メモリ（NVM）メタデータは、NVM に保存されたオブジェクトを管理および記述するために使用されます。このメタデータは、識別子、アクセス権限、フラグ、その他の属性など、各オブジェクトに関する重要な情報を提供します。メタデータにより、NVM 内のオブジェクトを確実に管理、アクセス、操作することができます。

```
/* NVM オブジェクトのユーザー指定メタデータ */
typedef struct {
whNvmId id;              / 一意の識別子 /
whNvmAccess access;      / アクセス権限 /
whNvmFlags flags;        / 追加フラグ /
whNvmSize len;           / データの長さ（バイト単位） /
uint8_t label[WOLFHSM_NVM_LABEL_LEN];  / ラベル */
} whNvmMetadata;
```

- ID (whNvmId id)：NVM オブジェクトの一意の識別子です。この ID は NVM 内の特定のオブジェクトを参照しアクセスするために使用されます。オブジェクトの読み取り、書き込み、削除などの操作を可能にします。
- アクセス (whNvmAccess access)：オブジェクトのアクセス権限を定義します。このフィールドは、誰がどのような条件でオブジェクトにアクセスできるかを指定します。セキュリティポリシーを実施し、権限のあるエンティティのみがオブジェクトと対話できることを保証します。
- フラグ (whNvmFlags flags)：追加情報を提供したりオブジェクトの動作を変更したりする追加フラグです。フラグは、オブジェクトが読み取り専用か、一時的か、その他の特定のプロパティを持つかなど、特別な属性や状態でオブジェクトをマークするために使用できます。
- 長さ (whNvmSize len)：オブジェクトに関連付けられたデータの長さ（バイト単位）です。
- ラベル (uint8_t label[])：オブジェクトの人間が読める形式のラベルまたは名前です。

0.4.4.2 不揮発性メモリのアーキテクチャ

wolfHSM サーバーは、不揮発性メモリ（NVM）操作を処理するために汎用コンポーネントアーキテクチャアプローチに従います。設定は汎用部分と特定部分に分かれており、柔軟性とカスタマイズ性を提供します。

1. **汎用設定 (wh_nvm.h)：**このヘッダーファイルは、NVM 操作の汎用インターフェースを定義します。nvm_Read、nvm_Write、nvm_Erase、nvm_Init などの NVM 操作用の関数ポインタが含まれています。これらの関数ポインタは whNvmConfig 構造体の一部であり、実際の NVM 実装を抽象 NVM インターフェースにバインドするために使用されます。
2. **特定設定 (wh_nvm_flash.c, wh_nvm_flash.h)：**これらのファイルは、フラッシュメモリ用の NVM インターフェースの具体的な実装を提供します。ここで定義される関数は、汎用インターフェースで定義された関数シグネチャに準拠しており、NVM 操作の実際の実装として使用できます。

whServerContext 構造体には whNvmConfig メンバーが含まれています。これは NVM 操作をサーバーコンテキストにバインドするために使用され、サーバーが設定された NVM インターフェースを使用して NVM 操作を実行できるようにします。

サーバーで NVM を初期化するには、以下の手順が必要です。

1. whNvmConfig 構造体を割り当て、初期化し、特定の NVM バックエンド（例：wh_nvm_flash.c から）へのバインディングを提供
2. whServerConfig 構造体を割り当て、初期化し、その nvmConfig メンバーを手順 1 で初期化した whNvmConfig 構造体に設定
3. whServerContext 構造体を割り当て
4. wh_Server_Init() を呼び出して、whServerConfig 構造体でサーバーを初期化

これにより、サーバーは指定されたバッキングストアで設定された NVM 操作を使用できるようになり、whNvmConfig 構造体で異なる実装を提供することで簡単に入れ替えることができます。

0.4.4.3 不揮発性メモリのバックエンド 現在、wolfHSM がサポートしている NVM バックエンドプロバイダーは、NVM フラッシュモジュール（wh_nvm_flash.c）のみです。このモジュールは、NVM インターフェース関数（wh_nvm.h）の具体的な実装を提供し、NVM データストアをフラッシュメモリデバイスにマッピングします。低レベルのフラッシュドライバーはデバイス固有であり、それ自体が汎用コンポーネント（wh_flash.h）として指定され、ターゲットハードウェアに応じて入れ替えることができます。

0.4.5 鍵管理

wolfHSM ライブラリは、不揮発性メモリからの鍵の保存、読み込み、エクスポート、高速アクセスのために RAM に頻繁に使用される鍵のキャッシング、およびハードウェア専用デバイス鍵との対話など、包括的な鍵管理機能を提供します。鍵は、対応するアクセス保護とともに、他の NVM オブジェクトと並んで不揮発性メモリに保存されます。wolfHSM は、特定のコンシューマーが使用するために鍵が選択されたとき、必要な暗号化ハードウェアに鍵を自動的に読み込みます。鍵管理 API の詳細については、クライアントライブラリとAPI ドキュメントのセクションをご参照ください。

0.4.6 暗号処理

wolfHSM の特徴的な機能の 1 つは、クライアントアプリケーションが wolfCrypt API を直接使用できるようにしながら、基盤となる暗号化操作を実際に HSM コア上で実行することです。これは以下のメリットをもたらします。

- クライアントアプリケーションは、HSM 間でデータを双方向に受け渡すために必要な複雑な通信トランザクションを設定する必要がありません。これにより、劇的にシンプルになります。
- wolfCrypt 呼び出しのパラメータを 1 つ変更するだけで、ローカルとリモートの HSM 実装を簡単に切り替えることができます。実装の最大限の柔軟性と開発の容易さを実現します。クライアントアプリケーションの開発は、HSM コアがオンラインになる前でも、wolfCrypt のローカルインスタンスでプロトタイプを作成できます。
- wolfHSM API は、シンプルかつ安定していて、豊富なドキュメントがあり、あらゆる環境でその動作が検証されています。

wolfCrypt API 呼び出しを wolfHSM サーバーに簡単にリダイレクトできる機能は、wolfCrypt の「暗号コールバック」（cryptocb とも呼ばれる）に基づいています。

wolfHSM クライアントは、暗号コールバックとしてリモートプロシージャコール（RPC）ロジックを実装することで、wolfCrypt API 呼び出しを wolfHSM サーバーにリダイレクトできます。wolfCrypt の暗号コールバックフレームワークにより、ユーザーは選択した暗号化アルゴリズムのデフォルト実装を上書きし、実行時に独自のカスタム実装を提供できます。wolfHSM クライアントライブラリは、wolfCrypt に暗号コールバックを登録し、各 wolfCrypt 暗号化 API 関数を安全な環境で実行される HSM サーバーへのリモートプロシージャコールに変換します。暗号コールバックは、ほとんどの wolfCrypt API 呼び出しで受け入れられるデバイス ID（devId）パラメータに基づいて選択されます。

wolfHSM は、wolfHSM サーバー暗号化デバイスを表す WOLFHSM_DEV_ID 値を定義しており、これを任意の wolfCrypt 関数の devId パラメータとして渡すことができます。devId パラメータをサポートする

wolfCrypt API には WOLFHSM_DEV_ID を渡すことができます。これがサポートされている場合、暗号化操作は自動的に wolfHSM サーバーによって実行されます。

0.4.6.1 ハードウェア暗号サポート 多くの HSM デバイスは、いくつかのアルゴリズムをターゲットとしたハードウェアアクセラレーション機能も備えています。wolfHSM サーバーは、HSM サーバーサイドの暗号化処理をデバイスハードウェアにオフロードすることもサポートしています。HSM デバイスが対応している場合、wolfHSM サーバーはユーザーからの特別な入力を必要とせずに、これを自動的に行うように設定できます。デバイス固有のハードウェアアクセラレーション機能については、そのデバイスの wolfHSM ポートのドキュメントをご覧ください。

0.4.7 AUTOSAR SHE

(本章は後日提供予定です。)

0.5 クライアントライブラリ

クライアントライブラリ API は、ユーザーが wolfHSM と対話するための主要な手段です。利用可能な関数の完全なリストとその説明については、API ドキュメントをご参照ください。

0.5.1 API の返り値

すべてのクライアント API 関数は、成功または失敗の種類を示す wolfHSM エラーコードを返します。一部の失敗は重大なエラーですが、他の失敗は単にユーザーからのアクションが必要であることを示すものもあります（例：ノンブロッキング操作の場合の WH_ERROR_NOTREADY）。多くのクライアント API は、サーバーエラーコード（場合によっては追加のステータス）もユーザーに伝達します。これにより、基礎となるリクエストトランザクションは成功したものの、サーバーが操作を実行できなかった場合にも対応できます。例えば、存在しない NVM オブジェクトをサーバーに要求した場合、NVM が満杯時にオブジェクトを追加しようとした場合、またはサーバーが対応するように設定されていない暗号化アルゴリズムを使用しようとした場合などが該当します。

エラーコードは wolfhsm/wh_error.h で定義しています。詳細については、API ドキュメントをご参照ください。

0.5.2 分割トランザクション処理

ほとんどのクライアント API は完全に非同期で、分割トランザクションに分解されています。つまり、操作リクエストとレスポンスについて別々の関数があります。リクエスト関数はサーバーにリクエストを送信し、ブロックせずに即座に戻ります。レスポンス関数は基礎となるトランスポートをポーリングしてレスポンスを確認し、レスポンスが存在する場合は処理を行い、まだ到着していない場合は即座に戻ります。これにより、クライアントの CPU サイクルを無駄にすることなく、サーバーに長時間実行される操作をリクエストできます。以下に、「echo」メッセージを使用した非同期リクエストとレスポンスの呼び出し例を示します。

```
int rc;

/* echo リクエストを送信 */
rc = wh_Client_EchoRequest(&clientCtx, sendLen, &sendBuffer);
if (rc != WH_ERROR_OK) {
    /* エラー処理 */
}

/* 他の処理を実行... */

/* サーバーレスポンスをポーリング */
```

```
while ((rc = wh_Client_EchoResponse(client, &recv_len, recv_buffer)) ==
 ↪  WH_ERROR_NOTREADY) {
    /* 他の処理を実行するか、または制御を譲る */
}

if (rc != WH_ERROR_OK) {
    /* エラー処理 */
}
```

0.5.3 クライアントコンテキスト

クライアントコンテキスト構造体（whClientContext）は、クライアントの実行時状態を保持し、サーバーとの接続のエンドポイントを表します。また、クライアントとサーバーのコンテキストは 1 対 1 の関係にあります。つまり複数のサーバーと対話するアプリケーションでは、サーバーごとに 1 つのクライアントコンテキストが必要になります。各クライアント API 関数は、クライアントコンテキストを引数として受け取り、どのサーバー接続に対応する操作かを示します。wolfSSL に慣れている方であれば、WOLFSSL 接続コンテキスト構造体と同様の使い方をするとご認識いただいてよいかもしれません。

0.5.3.1 クライアントコンテキストの初期化 クライアントコンテキストで任意のクライアント API を使用する前に、whClientConfig 設定構造体と wh_Client_Init() 関数を使用して構造体を設定し、初期化する必要があります。

クライアント設定構造体は、サーバー通信のためのコンテキストを設定・初期化するために使用される通信層設定（whCommClientConfig）を保持します。whCommClientConfig 構造体は、実際のトランスポート実装（組み込みまたはカスタム）を、クライアントが使用する抽象的な通信インターフェースにバインドします。

クライアントを設定する一般的な手順は以下の通りです。

1. 目的のトランスポートのためのトランスポート設定構造体、コンテキスト、およびコールバック実装を割り当てて初期化する
2. 通信クライアント設定構造体を割り当て、クライアントが使用できるようにステップ 1 のトランスポート設定にバインドする
3. ステップ 2 の通信クライアント設定を使用してクライアント設定構造体を割り当てて初期化する
4. クライアントコンテキスト構造体を割り当てる
5. wh_Client_Init() を呼び出してクライアント設定でクライアントを初期化する
6. クライアント API を使用してサーバーと接続する

以下に、組み込みの共有メモリトランスポートを使用するクライアントアプリケーションの設定例を示します。

```
#include <string.h> /* for memcmp() */
#include "wolfhsm/client.h"  /* クライアント API（通信設定を含む） */
#include "wolfhsm/wh_transport_mem.h" /* トランスポート実装 */

/* ステップ 1: 共有メモリトランスポート設定の割り当てと初期化 */
/* 共有メモリトランスポート設定 */
static whTransportMemConfig transportMemCfg = { /* 共有メモリ設定 */ };
/* 共有メモリトランスポートコンテキスト（状態） */
whTransportMemClientContext transportMemClientCtx = {0};
/* 抽象的な通信トランスポートインターフェースを
 * 具体的な実装にバインドするコールバック構造体 */
whTransportClientCb transportMemClientCb = {WH_TRANSPORT_MEM_CLIENT_CB};

/* ステップ 2: クライアント通信設定の割り当てとトランスポートへのバインド */
/* 選択したトランスポート設定を使用するようにクライアント通信を設定 */
```

```
whCommClientConfig commClientCfg[1] = {{
             .transport_cb      = transportMemClientCb,
             .transport_context = (void*)transportMemClientCtx,
             .transport_config  = (void*)transportMemCfg,
             .client_id         = 123, /* 一意のクライアント識別子 */
}};

/* ステップ 3: クライアント設定の割り当てと初期化 */
whClientConfig clientCfg= {
   .comm = commClientCfg,
};

/* ステップ 4: クライアントコンテキストの割り当て */
whClientContext clientCtx = {0};

/* ステップ 5: 提供された設定でクライアントを初期化 */
wh_Client_Init(&clientCtx, &clientCfg);
```

これでクライアントコンテキストが初期化され、作業を実行するためにクライアントライブラリ API 関数で使用できるようになりました。引き続き、サーバーに echo リクエストを送信してみます。

```
/* ステップ 6: クライアント API を使用してサーバーと対話 */

/* 送受信データを保持するバッファ */
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;

/* echo リクエストを送信し、レスポンスを受信するまでブロック */
wh_Client_Echo(client, sendLen, &sendBuffer, &recvLen, &recvBuffer);

if ((recvLen != sendLen ) ||
    (0 != memcmp(sendBuffer, recvBuffer, sendLen))) {
    /* エラー：送信したものと同じものが返ってこなかった */
}
```

設定と構造体の組み合わせが少々複雑だと感じるかもしれません。しかし、これによりクライアントコードを変更することなく、異なるトランスポート実装に簡単に入れ替えられるようにしています。例えば、共有メモリトランスポートから TCP トランスポートに切り替える場合、トランスポート設定とコールバック構造体を変更するだけで済み、クライアントコードの残りの部分（上記の手順 2 以降のすべて）は同じままです。

```
#include <string.h> /* for memcmp() */
#include "wolfhsm/client.h"  /* クライアント API（通信設定を含む） */
#include "port/posix_transport_tcp.h" /* トランスポート実装 */

/* ステップ 1: POSIX TCP トランスポート設定の割り当てと初期化 */
/* クライアント設定/コンテキスト */
whTransportClientCb posixTransportTcpCb = {PTT_CLIENT_CB};
posixTransportTcpClientContext posixTranportTcpCtx = {0};
posixTransportTcpConfig posixTransportTcpCfg = {
    /* IP とポートの設定 */
```

```
};

/* ステップ 2: クライアント通信設定の割り当てとトランスポートへのバインド */
/* 選択したトランスポート設定を使用するようにクライアント通信を設定 */
whCommClientConfig commClientCfg = {{
            .transport_cb      = posixTransportTcpCb,
            .transport_context = (void*)posixTransportTcpCtx,
            .transport_config  = (void*)posixTransportTcpCfg,
            .client_id         = 123, /* 一意のクライアント識別子 */
}};

/* 以降のステップは同じ... */
```

ステップ 6 の echo リクエストはあくまでも単純な使用例です。サーバーとの接続が設定されれば、任意のクライアント API を使用できます。

0.5.4 不揮発性メモリ操作

このセクションでは、クライアント NVM API の使用方法の例を示します。分割トランザクション API も同様に使用できますが、簡単のためにブロッキング API を使用します。

クライアントがサーバーの NVM ストレージを使用するには、まず初期化リクエストをサーバーに送信します。現時点では、これによってサーバー側で何らかのアクションが実行されることはありません。しかし将来的にはここに実装を加える可能性があるため、クライアントアプリケーションに初期化リクエストを含めることを推奨します。

```
int rc;
int serverRc;
uint32_t clientId; /* 現時点では未使用 */
uint32_t serverId;

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

/* ローカルとリモートの両方のエラーコードをチェック */
/* serverId にはサーバーの一意の ID が格納される */
```

初期化が完了すると、クライアントは NvmAddObject 関数を使用してオブジェクトを作成し追加できます。すべてのオブジェクトに対してメタデータエントリを作成する必要があることに注意してください。

```
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);
```

既存のオブジェクトに対応するデータは、その場で更新できます。

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

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

トランスポート経由でコピーして送信すべきでないオブジェクトについては、NvmAddObject 関数の DMA バージョンを使用できます。これらは値ではなく参照によってデータをサーバーに渡すため、サーバーが直接メモリ内のデータにアクセスできます。ただし、サーバーがクライアントのアドレスにアクセスする前に、プラットフォームがカスタムのアドレス変換またはキャッシュ無効化を必要とする場合は、DMA コールバックを実装する必要があることにご注意ください。

```
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 オブジェクトのデータは NvmRead 関数を使用して読み取ることができます。また、AddObjectDma に対応する NvmRead 関数の DMA バージョンも存在し、同様に使用できます。

```
const whNvmId myId = 123; /* 読み取りたいオブジェクトの ID */
const whNvmSize offset = 0; /* オブジェクトデータへのバイトオフセット */

whNvmSize outLen; /* リクエストされたデータのバイト長が格納される */
int outRc; /* サーバーのリターンコードが格納される */

byte myBuffer[BIG_SIZE];

whClient_NvmRead(&clientCtx, myId, offset, sizeof(myData), &serverRc, outLen,
 ↪  &myBuffer)
/* または DMA 経由で */
whClient_NvmReadDma(&clientCtx, myid, offset, sizeof(myData), &myBuffer,
 ↪  &serverRc);
```

オブジェクトは NvmDestroy 関数を使用して削除/破棄できます。これらの関数は、削除するオブジェクト ID のリスト（配列）を引数に取ります。リスト内の ID が NVM に存在しない場合でもエラーは発生しません。

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

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

NVM 内のオブジェクトは、NvmList 関数を使用して列挙できます。この関数は、start_id から始まる NVM リスト内の次にマッチする ID を取得し、access と flags にマッチする ID の総数を out_count に設定します。

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

すべての NVM API 関数の詳細な説明については、API ドキュメントをご参照ください。

0.5.5 鍵管理

wolfCrypt で使用することを意図した鍵は、以下の API を使用して HSM のキーストアにロードし、必要に応じて NVM に保存できます。

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

uint16_t keyId = WOLFHSM_KEYID_ERASED;
uint32_t keyLen;
byte key[AES_128_KEY_SIZE] = { /* AES 鍵 */ };
byte label[WOLFHSM_NVM_LABEL_LEN] = { /* 鍵ラベル */ };

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

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 は、鍵とラベルを HSM の RAM キャッシュに格納し、渡された keyId と関連付けます。WOLFHSM_KEYID_ERASED を keyId として使用すると、wolfHSM は新しい一意の keyId を割り当て、keyId パラメータを通じて返します。wolfHSM のキャッシュスロットの数は WOLFHSM_NUM_RAMKEYS で設定された数に制限されており、すべての鍵スロットが満杯の場合は WH_ERROR_NOSPACE を返します。キャッシュと NVM の両方に存在する鍵は、NVM にバックアップされているため、より多くの鍵のためのスペースを確保するためにキャッシュから削除されます。

wh_Client_KeyCommit は、キャッシュされた鍵を keyId で指定されたキーとして NVM に保存します。

wh_Client_KeyEvict は、鍵をキャッシュから削除しますが、コミットされている場合は NVM には残します。

wh_Client_KeyExport は、鍵の内容を HSM からクライアントに読み出します。

wh_Client_KeyErase は、指定された鍵をキャッシュから削除し、NVM からも消去します。

0.5.6 暗号操作

クライアントアプリケーションで wolfCrypt を使用する場合、devId 引数として WOLFHSM_DEV_ID を渡すことで、互換性のある暗号化操作を wolfHSM サーバーで実行できます。ただし、wolfHSM のリモート暗号化を使用する前に、wolfHSM クライアントを初期化する必要があります。

wolfHSM がそのアルゴリズムをまだサポートしていない場合、API 呼び出しは CRYPTOCB_UNAVAILABLE を返します。

以下に、クライアントアプリケーションが wolfHSM サーバーで AES CBC 暗号化操作を実行する例を示します。

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

whClientContext clientCtx;
whClientCfg clientCfg = { /* 設定 */ };
```

```
wh_Client_Init(&clientCtx, &clientCfg);

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

byte plainText[AES_BLOCK_SIZE] = { /* 平文 */ };
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);
```

クライアント所有の鍵の代わりに HSM 所有の鍵（例：HSM ハードウェアキー）を使用する必要がある場合、wh_Client_SetKeyAes（または他の暗号化アルゴリズム用の同様の関数）などのクライアント API 関数を使用します。これにより、指定された HSM キーを後続の暗号化操作に使用するよう wolfHSM に指示できます。

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

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

wh_Client_Init(&clientCtx, &clientCfg);

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

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

wc_AesInit(&aes, NULL, WOLFHSM_DEV_ID);

/* IV は鍵と別に設定する必要がある */
wc_AesSetIV(&aes, iv);

/* この鍵は使用前の任意のタイミングでキャッシュできます。ここでは例として示しています */
wh_Client_KeyCache(clientCtx, 0, label, sizeof(label), key, sizeof(key),
 ↪  &keyId);

wh_Client_SetKeyAes(&aes, keyId);

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

/* 鍵の削除は任意です。鍵はキャッシュまたは NVM に保存され、wolfCrypt で使用できます */
wh_Client_KeyEvict(clientCtx, keyId);
```

```
wc_AesFree(&aes);
```

暗号化をクライアントのローカルで実行したい場合は、wc_AesInit() に INVALID_DEVID を渡します。

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

より詳しい使用方法の説明やサポートされている暗号化アルゴリズムの詳細なリストをお求めでしたら、wolfSSL マニュアル内の wolfCrypt API リファレンスをご参照ください。

0.5.6.1 CMAC キャッシュされた鍵を使用する CMAC 操作の場合、CMAC ハッシュと検証操作を 1 回の関数呼び出しで実行するための、wolfHSM 固有の別の関数を呼び出す必要があります。関数が呼び出されたときにクライアントが鍵を提供できる場合は、通常の wc_AesCmacGenerate_ex と wc_AesCmacVerify_ex を使用できます。しかし、事前にキャッシュされたキーを使用するためには、wh_Client_AesCmacGenerate と wh_Client_AesCmacVerify を使用する必要があります。ワンショットではない関数の wc_InitCmac_ex、wc_CmacUpdate、wc_CmacFinal は、クライアント側の鍵・事前にキャッシュされた鍵のどちらでも使用できます。これらの関数でキャッシュされたキーを使用するには、呼び出し元は NULL キーパラメータを渡し、wh_Client_SetKeyCmac を使用して適切な keyId を設定する必要があります。

0.5.7 AUTOSAR SHE API

(本章は後日提供予定です。)

0.6 サーバライブラリ

wolfHSM サーバーライブラリは、暗号ライブラリ wolfCrypt のサーバー側実装です。アプリケーションが暗号化操作を専用サーバーにオフロードするためのインターフェイスを提供します。専用サーバーでは wolfHSM サーバーソフトウェアを実行します。これにより、アプリケーションは暗号化キーを管理したり、ローカルで操作を実行したりすることなく、暗号化操作を実行できます。

0.6.1 ことはじめ

(本章は後日提供予定です。)

0.6.2 内部構造

(本章は後日提供予定です。)

0.6.3 API リファレンス

(本章は後日提供予定です。)

0.6.4 鍵管理

(本章は後日提供予定です。)

0.6.5 暗号

wolfHSM はすべての暗号操作に wolfCrypt を使用します。つまり、wolfHSM は wolfCypt でサポートされている任意のアルゴリズムをオフロードし、wolfHSM サーバーで実行できます。これには、中国政府が義務付けた ShāngMì 暗号 (SM2、SM3、SM4) や、Kyber、LMS、XMSS などの耐量子暗号アルゴリズムも含まれます。

0.7 カスタマイズ

wolfHSM は、ビルド時のオプションやユーザー定義のコールバックを通じて複数のカスタマイズポイントを提供しています。これにより、コアライブラリのコードを変更することなく、幅広いユースケースや環境に合わせて調整できます。本章では、wolfHSM で利用可能なカスタマイズオプションの概要を説明します。内容は以下の通りです。

- ライブラリ設定: ライブラリの特定の機能を有効化または無効化するためのコンパイル時オプション
- DMA コールバック: クライアントのメモリに直接アクセスする前後で操作を実行するために、サーバーに登録できるカスタムコールバック
- DMA アドレス許可リスト: クライアントのアクセスを特定のメモリ領域に制限するためのサーバーの仕組み
- カスタムコールバック: デフォルトの HSM 機能では対応していない特定の操作を実行するために、サーバーに登録してクライアントから呼び出すことができるカスタムコールバック

0.7.1 ライブラリ設定

wolfHSM ライブラリには、コンパイル時の定義を通じてオン/オフを切り替えることができる多くのビルドオプションがあります。これらの設定マクロは、wh_config.h という名前の設定ヘッダーファイルで定義されることを想定しています。このファイルは wolfHSM を使用するアプリケーションによって定義され、コンパイラのインクルードパスのディレクトリ内に配置される必要があります。

サンプルの wh_config.h は動作確認済みの設定として、すべての wolfHSM ポートに同梱しています。

wh_config.h で定義可能な wolfHSM の設定項目の完全なリストについては、API ドキュメントをご参照ください。

0.7.2 DMA コールバック

wolfHSM のダイレクトメモリアクセス（DMA）コールバック機能は、クライアントのメモリに直接アクセスする前後でカスタム操作を行うためのフックをサーバー側に提供します。これは新しい共有メモリアーキテクチャへの移植を行う際によく必要とされます。この機能は特に、クライアントとサーバーのメモリ間の一貫性を確保するために、キャッシュの無効化やアドレス変換、その他のカスタムメモリ操作などの特定のアクションをサーバーが実行する必要がある場合に有用です。

コールバックは wh_Server_DmaRegisterCb32() および wh_Server_DmaRegisterCb64() 関数を使用してサーバーに登録できます。これによって提供されたコールバックは、サーバーコンテキストのすべての DMA 操作にバインドされます。

32 ビットと 64 ビットのアドレスを処理するための別々のコールバック関数が必要で、これらはそれぞれ 32 ビットと 64 ビットのクライアント DMA API 関数に対応します。コールバック関数は whServerDmaClientMem32Cb および whServerDmaClientMem64Cb 型で、以下のように定義しています。

```
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);
```

DMA コールバック関数は以下の引数を受け取ります:

- server: サーバーコンテキストへのポインタ
- clientAddr: アクセスされるクライアントのメモリアドレス
- serverPtr: サーバーのメモリアドレス（これもポインタ）へのポインタ。コールバックは必要な変換/再マッピングを適用した後にこれを設定します。

- len: 要求されたメモリ操作のバイト単位の長さ
- oper: 変換されたサーバーアドレスに対して実行されようとしているメモリ操作（注入ポイントについては次のセクションで説明）の種類
- flags: メモリ操作の追加フラグ。将来的に使用することを見越して確保している領域です。

コールバックは成功時に WH_ERROR_OK を、エラーが発生した場合はエラーコードを返す必要があります。コールバックが失敗した場合、サーバーはエラーコードをクライアントに伝達させます。

0.7.2.1 コールバックの位置 DMA コールバックは、サーバーのメモリアクセスの周りの 4 つの異なるポイントで行われます:

- 読み取り前: クライアントメモリからデータを読み取る前にコールバックが呼び出されます。サーバーはコールバックを使用して、アドレス変換やキャッシュの無効化など、必要な読み取り前の操作を実行する必要があります。
- 読み取り後: クライアントメモリからデータを読み取った後にコールバックが呼び出されます。サーバーはコールバックを使用して、キャッシュの同期など、必要な読み取り後の操作を実行する必要があります。
- 書き込み前: クライアントメモリにデータを書き込む前にコールバックが呼び出されます。サーバーはコールバックを使用して、アドレス変換やキャッシュの無効化など、必要な書き込み前の操作を実行する必要があります。
- 書き込み後: クライアントメモリにデータを書き込んだ後にコールバックが呼び出されます。サーバーはコールバックを使用して、キャッシュの同期など、必要な書き込み後の操作を実行する必要があります。

コールバックが呼び出されるポイントは、oper 引数を通じてコールバックに渡されます。具体的には、以下のいずれかの値です。

```
typedef enum {
    WH_SERVER_DMA_OPER_PRE_READ,  /* 読み取り前の操作 */
    WH_SERVER_DMA_OPER_POST_READ, /* 読み取り後の操作 */
    WH_SERVER_DMA_OPER_PRE_WRITE, /* 書き込み前の操作 */
    WH_SERVER_DMA_OPER_POST_WRITE /* 書き込み後の操作 */
} whServerDmaOper;
```

これにより、コールバックは oper の値で switch 文を使用し、実行されるメモリ操作の種類に基づいてカスタムロジックを実行できます。以下に DMA コールバックの実装例を示します。

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

/* 32 ビットクライアントアドレス用の DMA コールバック実装例 */
int myDmaCallback32(whServerContext* server, uint32_t clientAddr,
                    void** xformedCliAddr, uint32_t len,
                    whServerDmaOper oper, whServerDmaFlags flags)
{
   /* 必要に応じてクライアントアドレスをサーバーのアドレス空間に変換、例：memmap() */
   *xformedCliAddr = (void*)clientAddr; /* 変換を実行 */

   switch (oper) {
       case WH_DMA_OPER_CLIENT_READ_PRE:
           /* 読み取り前の操作をここに記述、例：キャッシュの無効化 */
           break;
       case WH_DMA_OPER_CLIENT_READ_POST:
           /* 読み取り後の操作をここに記述 */
           break;
       case WH_DMA_OPER_CLIENT_WRITE_PRE:
           /* 書き込み前の操作をここに記述 */
```

```
            break;
        case WH_DMA_OPER_CLIENT_WRITE_POST:
            /* 書き込み後の操作をここに記述、例：キャッシュのフラッシュ */
            break;
        default:
            return WH_ERROR_BADARGS;
    }

    return WH_ERROR_OK;
}
```

0.7.2.2 コールバックの登録 コールバックは、いつでもサーバーコンテキストに登録できます。初期化時にはサーバー設定構造体を通じて、初期化後にはコールバック登録関数を使用して実施します。

初期化時にコールバックを登録するには、コールバック関数をサーバー設定構造体内の DMA 設定構造体に含める必要があります。なお、コールバック関数の設定は任意であり、使用しないコールバックは NULL と設定できます。

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

/* DMA32 コールバックを持つサーバー設定構造体の初期化とサーバーの初期化の例 */
int main(void)
{
    whServerDmaConfig dmaCfg = {0};
    dmaCfg.dma32Cb = myDmaCallback32;


    whServerConfig serverCfg = {
        .dmaCfg = dmaCfg,

        /* 簡潔にするため、その他の設定は省略 */
    };

    whServerContext serverCtx;

    wh_Server_Init(&serverCtx, &serverCfg);

    /* サーバーアプリケーションのロジック */
}
```

初期化後にコールバックを登録する際は、以下のように関数を呼び出します。

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

int main(void)
{
    whServerConfig serverCfg =  { /* サーバーの設定 */ };

    whServerContext serverCtx;

    wh_Server_Init(&serverCtx, &serverCfg);

    /* 上で定義したコールバックを登録 */
    wh_Server_DmaRegisterCb32(&serverCtx, myDmaCallback32);
```

```
    /* サーバーアプリケーションのロジック */
}
```

0.7.3 DMA アドレス許可リスト

wolfHSM は、クライアントの DMA アドレスに対する「許可リスト」も公開しています。これにより、クライアントによるアクセスを事前に設定した特定のメモリ領域に制限できます。この機能は、サーバーが不正なアクセスを拒否するため、あるいはクライアントが安全にアクセスできるメモリにのみアクセスできるようにするために有用です。例えば、複数のコアでそれぞれクライアントが実行されるマルチコアシステムでは、クライアントは他のクライアントのメモリ領域にアクセスできないようにすべきであり、暗号化キーなどの機密情報を含む可能性のあるサーバーメモリを読み出すこともできないようにすべきです。

ただしこの許可リスト機能は、デバイス固有のメモリ保護メカニズムの上に第二層の保護として機能することを意図しています。よって、不正なメモリアクセスを防ぐ第一の防衛線とは考えるべきではありません。ユーザーはアプリケーションを厳密に分離し、HSM コアと関連メモリをシステムの残りの部分から分離するために必要な、デバイス固有のメモリ保護メカニズムを設定することが不可欠です。

0.7.3.1 許可リストへの登録 DMA コールバックと同様に、いつでもサーバーコンテキストに登録できます。

初期化時にリストを登録するには、サーバー設定構造体内の DMA 設定構造体にリストを設定する必要があります。

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

/* 許可されるメモリ領域を定義 */
const whServerDmaAddrAllowList allowList = {
    .readList = {
        {(void*)0x20001000, 0x100},  /* 0x20001000 から 0x200010FF までの読み取りを
↪  許可 */
        {(void*)0x20002000, 0x200},  /* 0x20002000 から 0x200021FF までの読み取りを
↪  許可 */
    },
    .writeList = {
        {(void*)0x20003000, 0x100},  /* 0x20003000 から 0x200030FF までの書き込みを
↪  許可 */
        {(void*)0x20004000, 0x200},  /* 0x20004000 から 0x200041FF までの書き込みを
↪  許可 */
    },
};

int main()
{
    whServerConfig config;


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

    whServerConfig serverCfg = {
        .dmaCfg = dmaCfg,
        /* 簡潔にするため、その他の設定は省略 */
    };
```

```
    whServerContext server;

    wh_Server_Init(&server, &config);

    /* これでサーバーは許可リストで設定されました */
    /* その他のサーバー操作を実行 */

    /* 許可リストは、サーバー設定構造体にリストが存在しない場合、
     * 初期化後に以下を使用して登録することもできます。
     *
     *     wh_Server_DmaRegisterAllowList(&server, &allowList);
     */
}
```

登録後、クライアントからサーバーに要求されるすべての DMA 操作は、許可リストと照合してチェックされます。クライアントが許可リストにないメモリ領域にアクセスしようとすると、サーバーはクライアントにエラーを返し、その操作は実行されません。

0.7.4 カスタムコールバック

wolfHSM のカスタムコールバック機能により、開発者はサーバー上にカスタムコールバック関数を登録することで、ライブラリの機能を拡張できます。これらのコールバックは、HSM が対応していない特定の操作を実行するために、クライアントから呼び出すことができます。例えば、周辺ハードウェアの有効化や無効化、カスタムの監視や認証ルーチンの実装、追加コアのセキュアブートのステージングなどです。

0.7.4.1 サーバー側 サーバーは、特定の操作を定義するカスタムコールバック関数を登録できます。これらの関数は `whServerCustomCb` 型である必要があります。

```
/* wh_server.h */

/* サーバーのカスタムコールバックの型定義 */
typedef int (*whServerCustomCb)(
    whServerContext* server,    /* 送信元のサーバーコンテキストを指すポインタ */
    const whMessageCustomCb_Request* req, /* クライアントからコールバックへのリクエスト
    ↪  */
    whMessageCustomCb_Response*      resp /* コールバックからクライアントへのレスポンス
↪  */
);
```

サーバーのカスタムコールバック関数は、サーバーのカスタムコールバックディスパッチテーブルのインデックスに対応する一意の識別子（ID）に関連付けられます。クライアントは呼び出しを要求する際に、この ID でコールバックを参照します。

カスタムコールバックは、コールバック関数に渡される `whMessageCustomCb_Request` 引数を通じて、値またはポインタ参照（共有メモリシステムで有用）によってクライアントから渡されたデータにアクセスできます。コールバックは入力データに基づいて処理を行い、`whMessageCustomCb_Response` 引数を通じてクライアントに返される出力データを生成できます。入力/出力クライアントデータの送受信は wolfHSM によって外部で処理されるため、カスタムコールバックでこれらを処理する必要はありません。レスポンス構造体には、クライアントに伝播させるためのエラーコードとリターンコードのフィールドも含まれています。エラーコードはコールバックによって設定され、リターンコードはカスタムコールバックからの戻り値に設定されます。

0.7.4.2 クライアント側 クライアントは、これらのカスタムコールバックを呼び出すためのリクエストをサーバーに送信できます。API は、クライアント API の他の関数と同様のリクエスト関数とレスポンス関数を提供します。クライアントはカスタムリクエスト構造体のインスタンスを宣言し、カスタムデータを設

定し、wh_Client_CustomCbRequest() を使用してサーバーに送信する必要があります。サーバーのレスポンスは wh_Client_CustomCbResponse() を使用してポーリングすることができ、正常に受信された場合、レスポンスデータは出力の whMessageCustomCb_Response() に格納されます。

クライアントは wh_Client_CustomCheckRegistered() 系の関数を使用して、特定のコールバック ID の登録状態を確認することもできます。この関数は、指定されたコールバック ID がサーバーの内部コールバックテーブルに登録されているかどうかをサーバーに問い合わせます。サーバーは登録状態を示す true または false で応答します。

0.7.4.3 カスタムメッセージ クライアントは、カスタムリクエストおよびレスポンスメッセージ構造体を通じて、カスタムコールバックにデータを渡したり、データを受け取ったりできます。これらのカスタムリクエストおよびレスポンスメッセージは、一意の ID、タイプ指示子、およびデータペイロードを含むように構成されています。ID はサーバーのコールバックテーブルのインデックスに対応します。タイプフィールドは、データペイロードをどのように解釈すべきかをカスタムコールバックに示します。データペイロードは固定サイズのデータバッファで、クライアントは任意の方法で使用できます。レスポンス構造体には、上述のエラーコード値が追加されています。

```
/* カスタムサーバーコールバックへのリクエストメッセージ */
typedef struct {
   uint32_t               id;   /* 登録されたコールバックの識別子 */
   uint32_t               type; /* whMessageCustomCb_Type */
   whMessageCustomCb_Data data;
} whMessageCustomCb_Request;

/* カスタムサーバーコールバックからのレスポンスメッセージ */
typedef struct {
   uint32_t id;   /* 登録されたコールバックの識別子 */
   uint32_t type; /* whMessageCustomCb_Type */
   int32_t  rc;   /* カスタムコールバックからの戻り値。err が 0 でない場合は無効 */
   int32_t  err;  /* wolfHSM 固有のエラー。err が 0 でない場合、rc は無効 */
   whMessageCustomCb_Data data;
} whMessageCustomCb_Response;
```

0.7.4.4 カスタムデータ側の定義 カスタムデータ型は whMessageCustomCb_Data 共用体を使用して定義することができます。この共用体には一般的なデータ型（例：dma32、dma64）用の便利な定義済み構造体と、ユーザー定義のスキーマ用の生データバッファ（buffer）が用意されています。クライアントはリクエストの type フィールドを通じて、サーバーコールバックがユニオン内のデータをどのように解釈すべきかを指示できます。wolfHSM は最初の数個のタイプインデックスを内部使用のために予約し、残りのタイプ値はカスタムクライアントタイプで使用可能です。

0.7.4.5 カスタムコールバックの例 ここでは、組み込みの DMA スタイルのアドレス指定タイプを使用する 1 つのリクエストと、カスタムユーザー定義タイプを使用する 2 つのリクエストの、合計 3 種類のクライアントリクエストを処理できるカスタムコールバックを実装します。

まず、クライアントとサーバー間で共有される共通のメッセージを定義します。

```
/* 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;
```

サーバー側では、リクエストを処理する前に、コールバックを定義し、サーバーコンテキストに登録する必要があります。なお、コールバックの登録は必ずしも最初のリクエストの処理前である必要はありません。いつでも行うことができます。

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

int doWorkOnClientAddr(uint8_t* addr, uint32_t size) {
    /* タスクを実行 */
}

int doWorkWithTypeA(myCustomTypeA* typeA) {
    /* タスクを実行 */
}

int doWorkWithTypeB(myCustomTypeB* typeB) {
    /* タスクを実行 */
}

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

    resp->err = WH_ERROR_OK;

    /* DMA リクエストを検出して処理 */
    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);
```

```
        /* 必要に応じてエラーコードを設定 */
        if (/* エラー条件 */) {
            resp->err = WH_ERROR_ABORTED;
        }
    }
    else if (req->type == MY_TYPE_B) {
        myCustomCbDataB *data = (myCustomCbDataB)((uintptr_t)req->data.data);
        rc = doWorkWithTypeB(data);
        /* 必要に応じてエラーコードを設定 */
        if (/* エラー条件 */) {
            resp->err = WH_ERROR_ABORTED;
        }
    }

    return rc;
}


int main(void) {

    whServerContext serverCtx;

    whServerConfig serverCfg = {
        /* サーバーの設定 */
    };

    wh_Server_Init(&serverCtx, &serverCfg);

    wh_Server_RegisterCustomCb(&serverCtx, MY_CUSTOM_CB_ID, customServerCb));

    /* サーバーリクエストを処理（簡略化） */
    while (1) {
        wh_Server_HandleRequestMessage(&serverCtx);
    }

}
```

これで、クライアントはカスタムコールバックの登録を確認し、リモートで呼び出すことができるようになりました。

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

whClientContext clientCtx;
whClientConfig  clientCfg = {
    /* クライアントの設定 */
};

whClient_Init(&clientCtx, &clientCfg);

bool isRegistered = wh_Client_CustomCheckRegistered(&client, MY_CUSTOM_CB_ID);

if (isRegistered) {
    uint8_t buffer[LARGE_SIZE] = {/* データ */};
    myCustomCbDataA typeA = {/* データ */};
```

```
    myCustomCbDataB typeB = {/* データ */};

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

    /* 組み込み DMA タイプでカスタムリクエストを送信 */
    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);
    /* レスポンスを使用した処理を実行 */

    /* ユーザー定義タイプでカスタムリクエストを送信 */
    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);
    /* レスポンスを使用した処理を実行 */

    /* 別のユーザー定義タイプでカスタムリクエストを送信 */
    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);
    /* レスポンスを使用した処理を実行 */
}
```

0.8 ポーティング

このセクションでは、wolfHSM のポーティングに関連する資料と情報を提供することを目的としています。次の内容についてご説明いたします。

- 概要
- 対応状況
- ポーティングインタフェース

0.8.1 概要

wolfHSM 自体には特定のハードウェアと対話するためのコードは含まれておらず、単体では実行できません。wolfHSM をデバイスで実行するには、サーバ アプリケーションがクライアントと通信して実行できるように、必要なハードウェアドライバーと抽象化レイヤーを使用してライブラリを構成する必要があります。具体的には、次の要素が必要です。

- サーバーアプリケーションの起動とハードウェアの初期化
- サーバーにおける wolfCrypt 構成
- サーバーの不揮発性メモリ構成
- サーバーとクライアントのトランスポート構成
- サーバーとクライアントの接続処理

これらの要件を提供し、サーバー API を起動可能なアプリケーションにラップするコードを、wolfHSM「ポート」と呼んでいます。

wolfHSM の公式ポートは、サポートされているさまざまなアーキテクチャ用に提供しています。それぞれのポートには次のものが含まれます。

- スタンドアロンリファレンスサーバーアプリケーション: このアプリケーションは、HSM コアで実行され、すべての安全な操作を処理することを目的としています。すぐに使用できる状態で完全に機能しますが、エンドユーザーがカスタマイズして追加のユースケースをサポートすることもできます。
- クライアントライブラリ: このライブラリは、ユーザーアプリケーションにリンクして、サーバーとの通信を容易にします。

0.8.2 対応状況

0.8.2.1 Infineon Aurix TC3XX このポートの配布はベンダーによって制限されています。ご入用の方は、info@wolfssl.jp までお問い合わせください。

Infineon Aurix TC3xx

- 最大 6 つのの 300MHz TriCore アプリケーションコア
- 1 つの 100MHz ARM Cortex M3 HSM コア
- 暗号化オフロード: TRNG、AES128、ECDSA、ED25519、SHA

0.8.2.2 ST SPC58NN このポートの配布はベンダーによって制限されています。ご入用の方は、info@wolfssl.jp までお問い合わせください。

ST SPC58NN

- 3 つの 200MHz e200z4256 PowerPC アプリケーションコア
- 1 つの 100MHz e200z0 PowerPC HSM コア (NVM 付き)
- 暗号化オフロード: TRNG、AES128

0.8.2.3 POSIX POSIX ポートは、さまざまな wolfHSM 抽象化の完全機能実装を提供します。これを使用すると、さまざまなハードウェア抽象化に期待される正確な機能をよりよく理解できます。

POSIX ポートは以下を提供します。

- メモリバッファトランスポート
- TCP トランスポート
- Unix ドメイントランスポート
- RAM ベースおよびファイルベースの NVM フラッシュシミュレーター

0.8.2.4 Skeleton スケルトンポートのソースコードは、将来のハードウェア/プラットフォーム ポートの開始点として使用できる非機能レイアウトを提供します。各関数には、基本的な説明と予想されるフローがエラーケースとともに説明されているため、さまざまな環境において一貫した結果を得ることができます。

スケルトンポートは、次のスタブ実装を提供します。

- トランスポートコールバック
- NVM フラッシュコールバック
- 暗号処理コールバック

0.8.3 ポーティングインタフェース

ポートはハードウェア固有のインターフェースを実装する必要があります。

- NVM フラッシュ インターフェース

暗号化ハードウェア

- TRNG、鍵、公開鍵暗号・共通鍵暗号

プラットフォームインターフェース

- ブートシーケンス、アプリケーションコアリセット、メモリ制限
- ポートと構成はコンパイル時に指定します。

.1 wolfHSM API リファレンス

.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_IsRequestPending**(const whClientContext * c)Reports whether a request has been sent whose matching response has not yet been consumed.
int	**wh_Client_CommInitRequest**(whClientContext * c)Sends a communication initialization request to the server.
int	**wh_Client_CommInitResponse**(whClientContext * c, uint32_t * out_clientid, uint32_t * out_serverid)Receives a communication initialization response from the server.
int	**wh_Client_CommInit**(whClientContext * c, uint32_t * out_clientid, uint32_t * out_serverid)Initializes communication with the server with a blocking call.
int	**wh_Client_CommInfoRequest**(whClientContext * c)Sends a communications information request to the server.

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

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

	Name
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.
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.

	Name
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.
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.

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

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

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

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

	Name
int	**wh_Client_CustomCbCheckRegisteredResponse**(whClientCont * c, uint16_t * outId, int * responseError)Receives a response from the server after checking if a custom callback is registered.
int	**wh_Client_CustomCbCheckRegistered**(whClientContext * c, uint16_t id, int * responseError)Sends a request to the server and receives a response to check if a custom callback is registered.
int	**wh_Client_AuthLoginRequest**(whClientContext * c, whAuthMethod method, const char * username, const void * auth_data, uint16_t auth_data_len)Sends an authentication request to the server.
int	**wh_Client_AuthLoginResponse**(whClientContext * c, int32_t * out_rc, whUserId * out_user_id)Receives an authentication response from the server.
int	**wh_Client_AuthLogin**(whClientContext * c, whAuthMethod method, const char * username, const void * auth_data, uint16_t auth_data_len, int32_t * out_rc, whUserId * out_user_id)Authenticates a user with the server (blocking convenience wrapper).
int	**wh_Client_AuthLogoutRequest**(whClientContext * c, whUserId user_id)Sends a logout request to the server.
int	**wh_Client_AuthLogoutResponse**(whClientContext * c, int32_t * out_rc)Receives a logout response from the server.
int	**wh_Client_AuthLogout**(whClientContext * c, whUserId user_id, int32_t * out_rc)Logs out a user from the server (blocking convenience wrapper).
int	**wh_Client_AuthUserAddResponse**(whClientContext * c, int32_t * out_rc, whUserId * out_user_id)Receives a user add response from the server.
int	**wh_Client_AuthUserAddRequest**(whClientContext * c, const char * username, whAuthPermissions permissions, whAuthMethod method, const void * credentials, uint16_t credentials_len)Sends a user add request to the server.
int	**wh_Client_AuthUserAdd**(whClientContext * c, const char * username, whAuthPermissions permissions, whAuthMethod method, const void * credentials, uint16_t credentials_len, int32_t * out_rc, whUserId * out_user_id)Adds a new user to the server (blocking convenience wrapper).

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

	Name
int	**wh_Client_CertInitRequest**(whClientContext * c)Sends a request to initialize the certificate manager on the server.
int	**wh_Client_CertInitResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after initializing the certificate manager.
int	**wh_Client_CertInit**(whClientContext * c, int32_t * out_rc)Sends a request and receives a response to initialize the certificate manager.
int	**wh_Client_CertAddTrustedRequest**(whClientContext * c, whNvmId id, whNvmAccess access, whNvmFlags flags, uint8_t * label, whNvmSize label_len, const uint8_t * cert, uint32_t cert_len)Sends a request to add a trusted certificate to NVM storage.
int	**wh_Client_CertAddTrustedResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after adding a trusted certificate.
int	**wh_Client_CertAddTrusted**(whClientContext * c, whNvmId id, whNvmAccess access, whNvmFlags flags, uint8_t * label, whNvmSize label_len, const uint8_t * cert, uint32_t cert_len, int32_t * out_rc)Sends a request and receives a response to add a trusted certificate.
int	**wh_Client_CertEraseTrustedRequest**(whClientContext * c, whNvmId id)Sends a request to erase a trusted certificate from NVM storage.
int	**wh_Client_CertEraseTrustedResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after erasing a trusted certificate.
int	**wh_Client_CertEraseTrusted**(whClientContext * c, whNvmId id, int32_t * out_rc)Sends a request and receives a response to erase a trusted certificate.
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.

	Name
int	**wh_Client_CertVerifyRequest**(whClientContext * c, const uint8_t * cert, uint32_t cert_len, whNvmId trustedRootNvmId)Sends a request to verify a certificate against trusted certificates.
int	**wh_Client_CertVerifyResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after verifying a certificate.
int	**wh_Client_CertVerify**(whClientContext * c, const uint8_t * cert, uint32_t cert_len, whNvmId trustedRootNvmId, int32_t * out_rc)Sends a request and receives a response to verify a certificate.
int	**wh_Client_CertVerifyAndCacheLeafPubKeyRequest**(whClientC * c, const uint8_t * cert, uint32_t cert_len, whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId keyId)Sends a request to verify a certificate and cache the leaf public key.
int	**wh_Client_CertVerifyAndCacheLeafPubKeyResponse**(whClient * c, whKeyId * out_keyId, int32_t * out_rc)Receives a response from the server after verifying a certificate and caching the leaf public key.
int	**wh_Client_CertVerifyAndCacheLeafPubKey**(whClientContext * c, const uint8_t * cert, uint32_t cert_len, whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId * inout_keyId, int32_t * out_rc)Sends a request and receives a response to verify a certificate, while also instructing the server to cache the public key of the leaf certificate.
int	**wh_Client_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.

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

	Name
int	**wh_Client_CertVerifyAcert**(whClientContext * c, const void * cert, uint32_t cert_len, whNvmId trustedRootNvmId, int32_t * out_rc)Sends a request and receives a response to verify an attribute certificate.
int	**wh_Client_CertVerifyAcertDmaRequest**(whClientContext * c, const void * cert, uint32_t cert_len, whNvmId trustedRootNvmId)Prepares and sends a DMA request to verify an attribute certificate.
int	**wh_Client_CertVerifyAcertDmaResponse**(whClientContext * c, int32_t * out_rc)Receives a response from the server after verifying an attribute certificate using DMA.
int	**wh_Client_DmaRegisterAllowList**(struct whClientContext_t * client, const whDmaAddrAllowList * allowlist)Registers a DMA address allowlist for client-side validation.
int	**wh_Client_DmaRegisterCb**(struct whClientContext_t * client, whClientDmaClientMemCb cb)Registers a custom client DMA callback.
int	**wh_Client_DmaProcessClientAddress**(struct whClientContext_t * client, uintptr_t clientAddr, void ** serverPtr, size_t len, whDmaOper oper, whDmaFlags flags)Processes a client address for DMA operations, using the native pointer size of the system.
int	**wh_Client_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_IsRequestPending

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

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

Parameters:

- **c** The client context.

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

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

.2.5.6 function wh_Client_CommInitRequest

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

Sends a communication initialization request to the server.

Parameters:

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

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

Comm component functions

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

.2.5.7 function wh_Client_CommInitResponse

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

Receives a communication initialization response from the server.

Parameters:

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

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

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

.2.5.8 function wh_Client_CommInit

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

Initializes communication with the server with a blocking call.

Parameters:

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

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

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

.2.5.9 function wh_Client_CommInfoRequest

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

Sends a communications information request to the server.

Parameters:

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

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

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

.2.5.10 function wh_Client_CommInfoResponse

```
int wh_Client_CommInfoResponse(
    whClientContext * c,
    uint8_t * out_version,
    uint8_t * out_build,
    uint32_t * out_cfg_comm_data_len,
    uint32_t * out_cfg_nvm_object_count,
    uint32_t * out_cfg_keycache_count,
    uint32_t * out_cfg_keycache_bufsize,
```

```
    uint32_t * out_cfg_keycache_bigcount,
    uint32_t * out_cfg_keycache_bigbufsize,
    uint32_t * out_cfg_customcb_count,
    uint32_t * out_cfg_dmaaddr_count,
    uint32_t * out_debug_state,
    uint32_t * out_boot_state,
    uint32_t * out_lifecycle_state,
    uint32_t * out_nvm_state
)
```

Receives a communication information response from the server.

Parameters:

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

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

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

.2.5.11 function wh_Client_CommInfo

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

Retrieves server configuration and state with a blocking call.

Parameters:

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

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

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

.2.5.12 function wh_Client_SetCryptoAffinity

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

Sets the crypto affinity on the client context.

Parameters:

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

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

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

.2.5.13 function wh_Client_GetCryptoAffinity

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

Gets the current crypto affinity from the client context.

Parameters:

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

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

.2.5.14 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.

.2.5.15 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.

.2.5.16 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.

.2.5.17 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.

.2.5.18 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.

.2.5.19 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.

.2.5.20 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.

.2.5.21 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.

.2.5.22 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.

.2.5.23 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.

.2.5.24 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.

.2.5.25 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.

.2.5.26 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.

.2.5.27 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.

.2.5.28 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.

.2.5.29 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.

.2.5.30 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.

.2.5.31 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.

.2.5.32 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.

.2.5.33 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.

.2.5.34 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.

.2.5.35 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.

.2.5.36 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.

.2.5.37 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.

.2.5.38 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.

.2.5.39 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.

.2.5.40 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.

.2.5.41 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.

.2.5.42 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.

.2.5.43 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.

.2.5.44 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.

.2.5.45 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.

.2.5.46 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.

.2.5.47 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.

.2.5.48 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.

.2.5.49 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.

.2.5.50 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.

.2.5.51 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.

.2.5.52 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.

.2.5.53 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.

.2.5.54 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

.2.5.55 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

.2.5.56 function wh_Client_CounterInitRequest

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

.2.5.57 function wh_Client_CounterInitResponse

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

.2.5.58 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.

.2.5.59 function wh_Client_CounterResetRequest

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

.2.5.60 function wh_Client_CounterResetResponse

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

.2.5.61 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.

.2.5.62 function wh_Client_CounterIncrementRequest

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

.2.5.63 function wh_Client_CounterIncrementResponse

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

.2.5.64 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.

.2.5.65 function wh_Client_CounterReadRequest

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

.2.5.66 function wh_Client_CounterReadResponse

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

.2.5.67 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.

.2.5.68 function wh_Client_CounterDestroyRequest

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

.2.5.69 function wh_Client_CounterDestroyResponse

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

.2.5.70 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.

.2.5.71 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.

.2.5.72 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.

.2.5.73 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.

.2.5.74 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.

.2.5.75 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.

.2.5.76 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.

.2.5.77 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.

.2.5.78 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.

.2.5.79 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.

.2.5.80 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.

.2.5.81 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.

.2.5.82 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.

.2.5.83 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.

.2.5.84 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.

.2.5.85 function wh_Client_NvmList

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

Sends a request to the server and receives a response to list non-volatile memory (NVM) objects.

Parameters:

- **c** Pointer to the client context.
- **access** The access permissions for the NVM objects to list.
- **flags** Flags associated with the NVM objects to list.
- **start_id** The starting ID of the NVM objects to list.
- **out_rc** Pointer to store the return code from the server.
- **out_count** Pointer to store the count of NVM objects that match the criteria.
- **out_id** Pointer to store the ID of the first matching NVM object.

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

Note: Enumerating all objects requires calling this function in a loop with successive start_id values. In thread-safe builds, concurrent NVM modifications between calls may result in inconsistent results.

This function handles the complete process of sending a request to the server to list NVM objects and receiving the response. It sends the request and repeatedly attempts to receive a valid response. The count is the number of objects that match the flags/access pattern starting at start_id. The out_id is the first matching object ID. This function blocks until the entire operation is complete or an error occurs.

.2.5.86 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.

.2.5.87 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.

.2.5.88 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.

.2.5.89 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.

.2.5.90 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.

.2.5.91 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.

.2.5.92 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.

.2.5.93 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.

.2.5.94 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.

.2.5.95 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.

.2.5.96 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.

.2.5.97 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.

.2.5.98 function wh_Client_NvmReadDmaRequest

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

.2.5.99 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.

.2.5.100 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.

.2.5.101 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.

.2.5.102 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.

.2.5.103 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.

.2.5.104 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.

.2.5.105 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.

.2.5.106 function wh_Client_AuthLoginRequest

```
int wh_Client_AuthLoginRequest(
    whClientContext * c,
    whAuthMethod method,
    const char * username,
    const void * auth_data,
    uint16_t auth_data_len
)
```

Sends an authentication request to the server.

Parameters:

- **c** Pointer to the client context.
- **method** The authentication method to use (e.g., WH_AUTH_METHOD_PIN).
- **username** The user name to login (null-terminated C string).
- **auth_data** Pointer to the authentication data.
- **auth_data_len** Length of the authentication data.

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

This function prepares and sends an authentication request message to the server. The request includes the authentication method and authentication data (e.g., PIN). This function does not block; it returns immediately after sending the request.

.2.5.107 function wh_Client_AuthLoginResponse

```
int wh_Client_AuthLoginResponse(
    whClientContext * c,
    int32_t * out_rc,
    whUserId * out_user_id
)
```

Receives an authentication response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the authenticated user ID.

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

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

.2.5.108 function wh_Client_AuthLogin

```
int wh_Client_AuthLogin(
    whClientContext * c,
    whAuthMethod method,
    const char * username,
```

```
    const void * auth_data,
    uint16_t auth_data_len,
    int32_t * out_rc,
    whUserId * out_user_id
)
```

Authenticates a user with the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **method** The authentication method to use (e.g., WH_AUTH_METHOD_PIN).
- **username** The user name to login (null-terminated C string).
- **auth_data** Pointer to the authentication data.
- **auth_data_len** Length of the authentication data.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the authenticated user ID.

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

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

.2.5.109 function wh_Client_AuthLogoutRequest

```
int wh_Client_AuthLogoutRequest(
    whClientContext * c,
    whUserId user_id
)
```

Sends a logout request to the server.

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to logout.

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

This function prepares and sends a logout request message to the server. This function does not block; it returns immediately after sending the request.

.2.5.110 function wh_Client_AuthLogoutResponse

```
int wh_Client_AuthLogoutResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a logout response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

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

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

.2.5.111 function wh_Client_AuthLogout

```
int wh_Client_AuthLogout(
    whClientContext * c,
    whUserId user_id,
    int32_t * out_rc
)
```

Logs out a user from the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to logout.
- **out_rc** Pointer to store the return code from the server.

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

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

.2.5.112 function wh_Client_AuthUserAddResponse

```
int wh_Client_AuthUserAddResponse(
    whClientContext * c,
    int32_t * out_rc,
    whUserId * out_user_id
)
```

Receives a user add response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the new user ID.

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

This function attempts to process a user add response message from the server. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

.2.5.113 function wh_Client_AuthUserAddRequest

```
int wh_Client_AuthUserAddRequest(
    whClientContext * c,
    const char * username,
    whAuthPermissions permissions,
    whAuthMethod method,
    const void * credentials,
    uint16_t credentials_len
)
```

Sends a user add request to the server.

Parameters:

- **c** Pointer to the client context.
- **username** The username for the new user.

- **permissions** The permissions for the new user.
- **method** The authentication method for the new user.
- **credentials** Pointer to the credentials data.
- **credentials_len** Length of the credentials data.

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

This function prepares and sends a user add request message to the server. This function does not block; it returns immediately after sending the request.

.2.5.114 function wh_Client_AuthUserAdd

```
int wh_Client_AuthUserAdd(
    whClientContext * c,
    const char * username,
    whAuthPermissions permissions,
    whAuthMethod method,
    const void * credentials,
    uint16_t credentials_len,
    int32_t * out_rc,
    whUserId * out_user_id
)
```

Adds a new user to the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **username** The username for the new user.
- **permissions** The permissions for the new user.
- **method** The authentication method for the new user.
- **credentials** Pointer to the credentials data.
- **credentials_len** Length of the credentials data.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the new user ID.

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

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

.2.5.115 function wh_Client_AuthUserGetRequest

```
int wh_Client_AuthUserGetRequest(
    whClientContext * c,
    const char * username
)
```

Sends a user get request to the server.

Parameters:

- **c** Pointer to the client context.
- **username** The username to look up.

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

This function prepares and sends a user get request message to the server. This function does not block; it returns immediately after sending the request.

.2.5.116 function wh_Client_AuthUserGetResponse

```
int wh_Client_AuthUserGetResponse(
    whClientContext * c,
    int32_t * out_rc,
    whUserId * out_user_id,
    whAuthPermissions * out_permissions
)
```

Receives a user get response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the user ID.
- **out_permissions** Pointer to store the user permissions.

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

This function attempts to process a user get response message from the server. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

.2.5.117 function wh_Client_AuthUserGet

```
int wh_Client_AuthUserGet(
    whClientContext * c,
    const char * username,
    int32_t * out_rc,
    whUserId * out_user_id,
    whAuthPermissions * out_permissions
)
```

Gets user information from the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **username** The username to look up.
- **out_rc** Pointer to store the return code from the server.
- **out_user_id** Pointer to store the user ID.
- **out_permissions** Pointer to store the user permissions.

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

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

.2.5.118 function wh_Client_AuthUserDeleteRequest

```
int wh_Client_AuthUserDeleteRequest(
    whClientContext * c,
    whUserId user_id
)
```

Sends a user delete request to the server.

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to delete.

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

This function prepares and sends a user delete request message to the server. This function does not block; it returns immediately after sending the request.

.2.5.119 function wh_Client_AuthUserDeleteResponse

```
int wh_Client_AuthUserDeleteResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a user delete response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

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

This function attempts to process a user delete response message from the server. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

.2.5.120 function wh_Client_AuthUserDelete

```
int wh_Client_AuthUserDelete(
    whClientContext * c,
    whUserId user_id,
    int32_t * out_rc
)
```

Deletes a user from the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to delete.
- **out_rc** Pointer to store the return code from the server.

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

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

.2.5.121 function wh_Client_AuthUserSetPermissionsRequest

```
int wh_Client_AuthUserSetPermissionsRequest(
    whClientContext * c,
    whUserId user_id,
    whAuthPermissions permissions
)
```

Sends a user set permissions request to the server.

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to set permissions for.
- **permissions** The new permissions to set.

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

This function prepares and sends a user set permissions request message to the server. This function does not block; it returns immediately after sending the request.

.2.5.122 function wh_Client_AuthUserSetPermissionsResponse

```
int wh_Client_AuthUserSetPermissionsResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a user set permissions response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

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

This function attempts to process a user set permissions response message from the server. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

.2.5.123 function wh_Client_AuthUserSetPermissions

```
int wh_Client_AuthUserSetPermissions(
    whClientContext * c,
    whUserId user_id,
    whAuthPermissions permissions,
    int32_t * out_rc
)
```

Sets user permissions on the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to set permissions for.
- **permissions** The new permissions to set.
- **out_rc** Pointer to store the return code from the server.

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

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

.2.5.124 function wh_Client_AuthUserSetCredentialsRequest

```
int wh_Client_AuthUserSetCredentialsRequest(
    whClientContext * c,
    whUserId user_id,
    whAuthMethod method,
    const void * current_credentials,
```

```
    uint16_t current_credentials_len,
    const void * new_credentials,
    uint16_t new_credentials_len
)
```

Sends a user set credentials request to the server.

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to set credentials for.
- **method** The authentication method.
- **current_credentials** Pointer to the current credentials data.
- **current_credentials_len** Length of the current credentials data.
- **new_credentials** Pointer to the new credentials data.
- **new_credentials_len** Length of the new credentials data.

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

This function prepares and sends a user set credentials request message to the server. This function does not block; it returns immediately after sending the request.

.2.5.125 function wh_Client_AuthUserSetCredentialsResponse

```
int wh_Client_AuthUserSetCredentialsResponse(
    whClientContext * c,
    int32_t * out_rc
)
```

Receives a user set credentials response from the server.

Parameters:

- **c** Pointer to the client context.
- **out_rc** Pointer to store the return code from the server.

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

This function attempts to process a user set credentials response message from the server. This function does not block; it returns WH_ERROR_NOTREADY if a response has not been received.

.2.5.126 function wh_Client_AuthUserSetCredentials

```
int wh_Client_AuthUserSetCredentials(
    whClientContext * c,
    whUserId user_id,
    whAuthMethod method,
    const void * current_credentials,
    uint16_t current_credentials_len,
    const void * new_credentials,
    uint16_t new_credentials_len,
    int32_t * out_rc
)
```

Sets user credentials on the server (blocking convenience wrapper).

Parameters:

- **c** Pointer to the client context.
- **user_id** The user ID to set credentials for.

- **method** The authentication method.
- **current_credentials** Pointer to the current credentials data.
- **current_credentials_len** Length of the current credentials data.
- **new_credentials** Pointer to the new credentials data.
- **new_credentials_len** Length of the new credentials data.
- **out_rc** Pointer to store the return code from the server.

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

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

.2.5.127 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.

.2.5.128 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.

.2.5.129 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.

.2.5.130 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.

.2.5.131 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.

.2.5.132 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.

.2.5.133 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.

.2.5.134 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.

.2.5.135 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.

.2.5.136 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.

.2.5.137 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.

.2.5.138 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.

.2.5.139 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.

.2.5.140 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.

.2.5.141 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.

.2.5.142 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.

.2.5.143 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.

.2.5.144 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.

.2.5.145 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.

.2.5.146 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.

.2.5.147 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.

.2.5.148 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.

.2.5.149 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.

.2.5.150 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.

.2.5.151 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.

.2.5.152 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.

.2.5.153 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.

.2.5.154 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.

.2.5.155 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.

.2.5.156 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.

.2.5.157 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.

.2.5.158 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.

.2.5.159 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.

.2.5.160 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.

.2.5.161 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.

.2.5.162 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.

.2.5.163 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.

.2.5.164 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.

.2.5.165 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"
#include "wolfhsm/wh_auth.h"


/* Forward declaration of the client structure so its elements can reference
 * itself  (e.g. server argument to custom callback) */
typedef struct whClientContext_t whClientContext;

#ifndef WOLFHSM_CFG_NO_CRYPTO

/* WolfCrypt types and defines */
#include "wolfssl/wolfcrypt/types.h"

/* 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;

/* Per-operation async DMA context: stores translated input DMA address
 * that must survive across the Request/Response boundary for POST cleanup.
 * State is now passed inline (not via DMA), so only input tracking is needed.
 * ioAddr:      translated DMA address for input POST
 * clientAddr:  original client address for POST
 * ioSz:        DMA'd size for POST */
typedef struct {
    uintptr_t ioAddr;
    uintptr_t clientAddr;
    uint64_t  ioSz;
} whClientDmaAsyncSha;

/* Async DMA context union. Only one DMA request can be in flight at a time
 * per client context, so a single union suffices. Each Response function
 * knows which member to access based on its own operation type. */
typedef union {
    whClientDmaAsyncSha sha;
} whClientDmaAsyncCtx;

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

```
#endif /* WOLFHSM_CFG_DMA */

/* Client context */
struct whClientContext_t {
    uint16_t     last_req_id;
    uint16_t     last_req_kind;
    uint32_t     cryptoAffinity;
#ifdef WOLFHSM_CFG_DMA
    whClientDmaContext dma;
#endif /* WOLFHSM_CFG_DMA */
    whCommClient comm[1];
};

struct whClientConfig_t {
    whCommClientConfig* comm;
#ifdef WOLFHSM_CFG_DMA
    whClientDmaConfig* dmaConfig;
#endif /* WOLFHSM_CFG_DMA */
};
typedef struct whClientConfig_t whClientConfig;


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

int wh_Client_Cleanup(whClientContext* c);

int wh_Client_SendRequest(whClientContext* c, uint16_t group, uint16_t action,
                          uint16_t data_size, const void* data);
int wh_Client_RecvResponse(whClientContext* c, uint16_t* out_group,
                           uint16_t* out_action, uint16_t* out_size,
                           void* data);

int wh_Client_IsRequestPending(const whClientContext* c);

int wh_Client_CommInitRequest(whClientContext* c);

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

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


int wh_Client_CommInfoRequest(whClientContext* c);

int wh_Client_CommInfoResponse(whClientContext* c,
        uint8_t* out_version,
        uint8_t* out_build,
        uint32_t *out_cfg_comm_data_len,
        uint32_t *out_cfg_nvm_object_count,
        uint32_t *out_cfg_keycache_count,
        uint32_t *out_cfg_keycache_bufsize,
        uint32_t *out_cfg_keycache_bigcount,
        uint32_t *out_cfg_keycache_bigbufsize,
```

```
        uint32_t *out_cfg_customcb_count,
        uint32_t *out_cfg_dmaaddr_count,
        uint32_t *out_debug_state,
        uint32_t *out_boot_state,
        uint32_t *out_lifecycle_state,
        uint32_t *out_nvm_state);

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

int wh_Client_SetCryptoAffinity(whClientContext* c, uint32_t affinity);

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

int wh_Client_CommCloseRequest(whClientContext* c);

int wh_Client_CommCloseResponse(whClientContext* c);

int wh_Client_CommClose(whClientContext* c);

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

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

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

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

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

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

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

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

int wh_Client_KeyEvictResponse(whClientContext* c);

int wh_Client_KeyEvict(whClientContext* c, uint16_t keyId);

int wh_Client_KeyExportRequest(whClientContext* c, uint16_t keyId);

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

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

int wh_Client_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);

/* Auth Manager functions */

int wh_Client_AuthLoginRequest(whClientContext* c, whAuthMethod method,
                               const char* username, const void* auth_data,
                               uint16_t auth_data_len);

int wh_Client_AuthLoginResponse(whClientContext* c, int32_t* out_rc,
                                whUserId* out_user_id);

int wh_Client_AuthLogin(whClientContext* c, whAuthMethod method,
                        const char* username, const void* auth_data,
                        uint16_t auth_data_len, int32_t* out_rc,
                        whUserId* out_user_id);

int wh_Client_AuthLogoutRequest(whClientContext* c, whUserId user_id);

int wh_Client_AuthLogoutResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_AuthLogout(whClientContext* c, whUserId user_id, int32_t*
 ↪  out_rc);

int wh_Client_AuthUserAddResponse(whClientContext* c, int32_t* out_rc,
                                  whUserId* out_user_id);

int wh_Client_AuthUserAddRequest(whClientContext* c, const char* username,
                                 whAuthPermissions permissions,
                                 whAuthMethod method, const void* credentials,
                                 uint16_t credentials_len);

int wh_Client_AuthUserAdd(whClientContext* c, const char* username,
                          whAuthPermissions permissions, whAuthMethod method,
                          const void* credentials, uint16_t credentials_len,
                          int32_t* out_rc, whUserId* out_user_id);

int wh_Client_AuthUserGetRequest(whClientContext* c, const char* username);

int wh_Client_AuthUserGetResponse(whClientContext* c, int32_t* out_rc,
                                  whUserId*          out_user_id,
                                  whAuthPermissions* out_permissions);

int wh_Client_AuthUserGet(whClientContext* c, const char* username,
                          int32_t* out_rc, whUserId* out_user_id,
                          whAuthPermissions* out_permissions);

int wh_Client_AuthUserDeleteRequest(whClientContext* c, whUserId user_id);

int wh_Client_AuthUserDeleteResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_AuthUserDelete(whClientContext* c, whUserId user_id,
                             int32_t* out_rc);
```

```
int wh_Client_AuthUserSetPermissionsRequest(whClientContext*  c,
                                            whUserId          user_id,
                                            whAuthPermissions permissions);

int wh_Client_AuthUserSetPermissionsResponse(whClientContext* c,
                                             int32_t*         out_rc);

int wh_Client_AuthUserSetPermissions(whClientContext* c, whUserId user_id,
                                     whAuthPermissions permissions,
                                     int32_t*          out_rc);

int wh_Client_AuthUserSetCredentialsRequest(
    whClientContext* c, whUserId user_id, whAuthMethod method,
    const void* current_credentials, uint16_t current_credentials_len,
    const void* new_credentials, uint16_t new_credentials_len);

int wh_Client_AuthUserSetCredentialsResponse(whClientContext* c,
                                             int32_t*         out_rc);

int wh_Client_AuthUserSetCredentials(
    whClientContext* c, whUserId user_id, whAuthMethod method,
    const void* current_credentials, uint16_t current_credentials_len,
    const void* new_credentials, uint16_t new_credentials_len, int32_t*
     ↪  out_rc);
/* Certificate functions */

int wh_Client_CertInitRequest(whClientContext* c);

int wh_Client_CertInitResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertInit(whClientContext* c, int32_t* out_rc);

int wh_Client_CertAddTrustedRequest(whClientContext* c, whNvmId id,
                                    whNvmAccess access, whNvmFlags flags,
                                    uint8_t* label, whNvmSize label_len,
                                    const uint8_t* cert, uint32_t cert_len);

int wh_Client_CertAddTrustedResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertAddTrusted(whClientContext* c, whNvmId id, whNvmAccess
 ↪  access,
                             whNvmFlags flags, uint8_t* label,
                             whNvmSize label_len, const uint8_t* cert,
                             uint32_t cert_len, int32_t* out_rc);

int wh_Client_CertEraseTrustedRequest(whClientContext* c, whNvmId id);

int wh_Client_CertEraseTrustedResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertEraseTrusted(whClientContext* c, whNvmId id, int32_t*
 ↪  out_rc);

int wh_Client_CertReadTrustedRequest(whClientContext* c, whNvmId id,
                                     uint32_t cert_len);
```

```
int wh_Client_CertReadTrustedResponse(whClientContext* c, uint8_t* cert,
                                      uint32_t* cert_len, int32_t* out_rc);

int wh_Client_CertReadTrusted(whClientContext* c, whNvmId id, uint8_t* cert,
                              uint32_t* cert_len, int32_t* out_rc);

int wh_Client_CertVerifyRequest(whClientContext* c, const uint8_t* cert,
                                uint32_t cert_len, whNvmId trustedRootNvmId);

int wh_Client_CertVerifyResponse(whClientContext* c, int32_t* out_rc);

int wh_Client_CertVerify(whClientContext* c, const uint8_t* cert,
                         uint32_t cert_len, whNvmId trustedRootNvmId,
                         int32_t* out_rc);

int wh_Client_CertVerifyAndCacheLeafPubKeyRequest(
    whClientContext* c, const uint8_t* cert, uint32_t cert_len,
    whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId keyId);

int wh_Client_CertVerifyAndCacheLeafPubKeyResponse(whClientContext* c,
                                                   whKeyId*         out_keyId,
                                                   int32_t*         out_rc);

int wh_Client_CertVerifyAndCacheLeafPubKey(
    whClientContext* c, const uint8_t* cert, uint32_t cert_len,
    whNvmId trustedRootNvmId, whNvmFlags cachedKeyFlags, whKeyId* inout_keyId,
    int32_t* out_rc);


#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.

	Name
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)
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.

	Name
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.
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.

	Name
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.
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.

	Name
int	**wh_Client_CmacKdfMakeExportKey**(whClientContext * ctx, whKeyId saltKeyId, const uint8_t * salt, uint32_t saltSz, whKeyId zKeyId, const uint8_t * z, uint32_t zSz, const uint8_t * fixedInfo, uint32_t fixedInfoSz, uint8_t * out, uint32_t outSz)Generate CMAC two-step KDF output and export to the client.
int	**wh_Client_AesSetKeyId**(Aes * key, whNvmId keyId)Associates an AES key with a specific key ID.
int	**wh_Client_AesGetKeyId**(Aes * key, whNvmId * outId)Gets the wolfHSM keyId being used by the wolfCrypt struct.
int	**wh_Client_AesCtr**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-CTR operation.
int	**wh_Client_AesCtrDma**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-CTR operation using DMA.
int	**wh_Client_AesEcb**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-ECB operation.
int	**wh_Client_AesEcbDma**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-ECB operation using DMA.
int	**wh_Client_AesCbc**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-CBC operation.
int	**wh_Client_AesCbcDma**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, uint8_t * out)Performs an AES-CBC operation using DMA.
int	**wh_Client_AesCbcRequest**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len)Send an AES-CBC encrypt/decrypt request to the server (non-blocking)
int	**wh_Client_AesCbcResponse**(whClientContext * ctx, Aes * aes, uint8_t * out, uint32_t * out_size)Receive the server's AES-CBC response (non-blocking)
int	**wh_Client_AesGcm**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, const uint8_t * iv, uint32_t iv_len, const uint8_t * authin, uint32_t authin_len, const uint8_t * dec_tag, uint8_t * enc_tag, uint32_t tag_len, uint8_t * out)Performs an AES-GCM operation.

	Name
int	**wh_Client_AesGcmDma**(whClientContext * ctx, Aes * aes, int enc, const uint8_t * in, uint32_t len, const uint8_t * iv, uint32_t iv_len, const uint8_t * authin, uint32_t authin_len, const uint8_t * dec_tag, uint8_t * enc_tag, uint32_t tag_len, uint8_t * out)Performs an AES-GCM operation using DMA.
int	**wh_Client_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_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_Sha256UpdateRequest**(whClientContext * ctx, wc_Sha256 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a non-DMA SHA-256 Update.
int	**wh_Client_Sha256UpdateResponse**(whClientContext * ctx, wc_Sha256 * sha)Async response half of a non-DMA SHA-256 Update.
int	**wh_Client_Sha256FinalRequest**(whClientContext * ctx, wc_Sha256 * sha)Async request half of a non-DMA SHA-256 Final.
int	**wh_Client_Sha256FinalResponse**(whClientContext * ctx, wc_Sha256 * sha, uint8_t * out)Async response half of a non-DMA SHA-256 Final.
int	**wh_Client_Sha256Dma**(whClientContext * ctx, wc_Sha256 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-256 hash operation on the input data using DMA.
int	**wh_Client_Sha256DmaUpdateRequest**(whClientContext * ctx, wc_Sha256 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a DMA SHA-256 Update.

	Name
int	**wh_Client_Sha256DmaUpdateResponse**(whClientContext * ctx, wc_Sha256 * sha)Async response half of a DMA SHA-256 Update.
int	**wh_Client_Sha256DmaFinalRequest**(whClientContext * ctx, wc_Sha256 * sha)Async request half of a DMA SHA-256 Final.
int	**wh_Client_Sha256DmaFinalResponse**(whClientContext * ctx, wc_Sha256 * sha, uint8_t * out)Async response half of a DMA SHA-256 Final.
int	**wh_Client_Sha224**(whClientContext * ctx, wc_Sha224 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-224 hash operation on the input data.
int	**wh_Client_Sha224UpdateRequest**(whClientContext * ctx, wc_Sha224 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a non-DMA SHA-224 Update.
int	**wh_Client_Sha224UpdateResponse**(whClientContext * ctx, wc_Sha224 * sha)Async response half of a non-DMA SHA-224 Update.
int	**wh_Client_Sha224FinalRequest**(whClientContext * ctx, wc_Sha224 * sha)Async request half of a non-DMA SHA-224 Final.
int	**wh_Client_Sha224FinalResponse**(whClientContext * ctx, wc_Sha224 * sha, uint8_t * out)Async response half of a non-DMA SHA-224 Final.
int	**wh_Client_Sha224Dma**(whClientContext * ctx, wc_Sha224 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-224 hash operation on the input data using DMA.
int	**wh_Client_Sha224DmaUpdateRequest**(whClientContext * ctx, wc_Sha224 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)
int	**wh_Client_Sha224DmaUpdateResponse**(whClientContext * ctx, wc_Sha224 * sha)
int	**wh_Client_Sha224DmaFinalRequest**(whClientContext * ctx, wc_Sha224 * sha)
int	**wh_Client_Sha224DmaFinalResponse**(whClientContext * ctx, wc_Sha224 * sha, uint8_t * out)
int	**wh_Client_Sha384**(whClientContext * ctx, wc_Sha384 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-384 hash operation on the input data.
int	**wh_Client_Sha384UpdateRequest**(whClientContext * ctx, wc_Sha384 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a non-DMA SHA-384 Update.
int	**wh_Client_Sha384UpdateResponse**(whClientContext * ctx, wc_Sha384 * sha)Async response half of a non-DMA SHA-384 Update.

	Name
int	**wh_Client_Sha384FinalRequest**(whClientContext * ctx, wc_Sha384 * sha)Async request half of a non-DMA SHA-384 Final.
int	**wh_Client_Sha384FinalResponse**(whClientContext * ctx, wc_Sha384 * sha, uint8_t * out)Async response half of a non-DMA SHA-384 Final.
int	**wh_Client_Sha384Dma**(whClientContext * ctx, wc_Sha384 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-384 hash operation on the input data using DMA.
int	**wh_Client_Sha384DmaUpdateRequest**(whClientContext * ctx, wc_Sha384 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)
int	**wh_Client_Sha384DmaUpdateResponse**(whClientContext * ctx, wc_Sha384 * sha)
int	**wh_Client_Sha384DmaFinalRequest**(whClientContext * ctx, wc_Sha384 * sha)
int	**wh_Client_Sha384DmaFinalResponse**(whClientContext * ctx, wc_Sha384 * sha, uint8_t * out)
int	**wh_Client_Sha512**(whClientContext * ctx, wc_Sha512 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-512 hash operation on the input data.
int	**wh_Client_Sha512UpdateRequest**(whClientContext * ctx, wc_Sha512 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)Async request half of a non-DMA SHA-512 Update.
int	**wh_Client_Sha512UpdateResponse**(whClientContext * ctx, wc_Sha512 * sha)Async response half of a non-DMA SHA-512 Update.
int	**wh_Client_Sha512FinalRequest**(whClientContext * ctx, wc_Sha512 * sha)Async request half of a non-DMA SHA-512 Final.
int	**wh_Client_Sha512FinalResponse**(whClientContext * ctx, wc_Sha512 * sha, uint8_t * out)Async response half of a non-DMA SHA-512 Final.
int	**wh_Client_Sha512Dma**(whClientContext * ctx, wc_Sha512 * sha, const uint8_t * in, uint32_t inLen, uint8_t * out)Performs a SHA-512 hash operation on the input data using DMA.
int	**wh_Client_Sha512DmaUpdateRequest**(whClientContext * ctx, wc_Sha512 * sha, const uint8_t * in, uint32_t inLen, bool * requestSent)
int	**wh_Client_Sha512DmaUpdateResponse**(whClientContext * ctx, wc_Sha512 * sha)
int	**wh_Client_Sha512DmaFinalRequest**(whClientContext * ctx, wc_Sha512 * sha)
int	**wh_Client_Sha512DmaFinalResponse**(whClientContext * ctx, wc_Sha512 * sha, uint8_t * out)
int	**wh_Client_MlDsaSetKeyId**(MlDsaKey * key, whKeyId keyId)Associates a ML-DSA key with a specific key ID.

	Name
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, const byte * context, byte contextLen, word32 preHashType)Sign a message using a ML-DSA private key.
int	**wh_Client_MlDsaVerify**(whClientContext * ctx, const byte * sig, word32 sig_len, const byte * msg, word32 msg_len, int * res, MlDsaKey * key, const byte * context, byte contextLen, word32 preHashType)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, const byte * context, byte contextLen, word32 preHashType)Sign a message using ML-DSA with DMA.
int	**wh_Client_MlDsaVerifyDma**(whClientContext * ctx, const byte * sig, word32 sig_len, const byte * msg, word32 msg_len, int * res, MlDsaKey * key, const byte * context, byte contextLen, word32 preHashType)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
)
```

Performs an AES-CTR operation.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

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

This function performs an AES-CTR encrypt or decrypt operation on the input data and stores the result in the output buffer.

.3.2.44 function wh_Client_AesCtrDma

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

Performs an AES-CTR operation using DMA.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

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

This function performs an AES-CTR encrypt or decrypt operation on the input data and stores the result in the output buffer using direct memory access when communicating with the wolfHSM server.

.3.2.45 function wh_Client_AesEcb

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

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

Performs an AES-ECB operation.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

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

This function performs an AES-ECB encrypt or decrypt operation on the input data and stores the result in the output buffer.

.3.2.46 function wh_Client_AesEcbDma

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

Performs an AES-ECB operation using DMA.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

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

This function performs an AES-ECB encrypt or decrypt operation on the input data and stores the result in the output buffer using direct memory access when communicating with the wolfHSM server.

.3.2.47 function wh_Client_AesCbc

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

Performs an AES-CBC operation.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

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

This function performs an AES-CBC encrypt or decrypt operation on the input data and stores the result in the output buffer.

.3.2.48 function wh_Client_AesCbcDma

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

Performs an AES-CBC operation using DMA.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.
- **out** Pointer to the output data.

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

This function performs an AES-CBC encrypt or decrypt operation on the input data and stores the result in the output buffer using direct memory access when communicating with the wolfHSM server.

.3.2.49 function wh_Client_AesCbcRequest

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

Send an AES-CBC encrypt/decrypt request to the server (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure with key and IV state
- **enc** 1 for encrypt, 0 for decrypt
- **in** Pointer to the input data (must be block-aligned)
- **len** Length of the input data in bytes (must be a multiple of AES_BLOCK_SIZE)

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

Sends a single AES-CBC request to the server. The key material is read from the Aes struct (set via wc_AesSetKey or wh_Client_AesSetKeyId). For decryption, the IV is updated with the last input ciphertext block before sending to support in-place operation. Use wh_Client_AesCbcResponse to retrieve the result.

.3.2.50 function wh_Client_AesCbcResponse

```
int wh_Client_AesCbcResponse(
    whClientContext * ctx,
    Aes * aes,
    uint8_t * out,
    uint32_t * out_size
)
```

Receive the server's AES-CBC response (non-blocking)

Parameters:

- **ctx** Pointer to the client context
- **aes** Pointer to the AES structure (IV updated on encrypt)
- **out** Pointer to where the output data is placed. Must not be NULL.
- **out_size** Set to the number of bytes produced. May be NULL.

Return: int Returns 0 on success, WH_ERROR_NOTREADY if the response is not yet available, or a negative error code on failure.

Retrieves the result of a prior wh_Client_AesCbcRequest call. For encryption, the IV in the Aes struct is updated with the last ciphertext block for CBC chaining. Returns WH_ERROR_NOTREADY if the response is not yet available.

.3.2.51 function wh_Client_AesGcm

```
int wh_Client_AesGcm(
    whClientContext * ctx,
    Aes * aes,
    int enc,
    const uint8_t * in,
    uint32_t len,
    const uint8_t * iv,
    uint32_t iv_len,
    const uint8_t * authin,
    uint32_t authin_len,
    const uint8_t * dec_tag,
    uint8_t * enc_tag,
    uint32_t tag_len,
    uint8_t * out
)
```

Performs an AES-GCM operation.

Parameters:

- **ctx** Pointer to the wolfHSM client context.
- **aes** Pointer to the AES structure.
- **enc** 1 for encrypt, 0 for decrypt.
- **in** Pointer to the input data.
- **len** Length of the input and output data in bytes.

- **iv** Pointer to the IV data.
- **iv_len** Length of the IV data in bytes.
- **authin** Pointer to the authentication data.
- **authin_len** Length of the authentication data in bytes.
- **dec_tag** Pointer to the decryption tag data.
- **enc_tag** Pointer to the encryption tag data.
- **tag_len** Length of the tag data in bytes.
- **out** Pointer to the output data.

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

This function performs an AES-GCM encrypt or decrypt operation on the input data and stores the result in the output buffer.

.3.2.52 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.

.3.2.53 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.

.3.2.54 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

.3.2.55 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.

.3.2.56 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.

.3.2.57 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.

.3.2.58 function wh_Client_Sha256UpdateRequest

```
int wh_Client_Sha256UpdateRequest(
    whClientContext * ctx,
    wc_Sha256 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a non-DMA SHA-256 Update.

Parameters:

- **ctx** Client context.
- **sha** SHA-256 context (buffer/buffLen updated on success).
- **in** Input data (may be NULL only if inLen == 0).
- **inLen** Input length. Must not exceed the per-call capacity (max inline + remaining buffer slack); use the blocking wrapper for arbitrary lengths.
- **requestSent** Set to true if a server request was sent and a matching Response call is required; false if the input was fully absorbed into sha->buffer and no round-trip was issued.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if inLen exceeds the per-call capacity (sha is left unchanged in that case).

Serializes and sends an Update request carrying as many full blocks as fit in the comm buffer (up to WH_MESSAGE_CRYPTO_SHA256_MAX_INLINE_UPDATE_SZ bytes), absorbing any leading bytes already buffered in sha->buffer. Any tail (<64 bytes) remaining after this call is stored in sha->buffer for the next call. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext (enforced by the comm layer's pending-request tracking). If *requestSent is true, the caller MUST call wh_Client_Sha256UpdateResponse before issuing any other async Request on the same ctx, including a Request using a different wc_Sha256 instance or a different algorithm.

.3.2.59 function wh_Client_Sha256UpdateResponse

```
int wh_Client_Sha256UpdateResponse(
    whClientContext * ctx,
    wc_Sha256 * sha
)
```

Async response half of a non-DMA SHA-256 Update.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, updates sha->digest/hiLen/loLen from the reply. MUST only be called if the matching Request returned requestSent == true.

.3.2.60 function wh_Client_Sha256FinalRequest

```
int wh_Client_Sha256FinalRequest(
    whClientContext * ctx,
```

```
    wc_Sha256 * sha
)
```

Async request half of a non-DMA SHA-256 Final.

Sends the current sha->buffer (0..63 bytes) as the last block.

.3.2.61 function wh_Client_Sha256FinalResponse

```
int wh_Client_Sha256FinalResponse(
    whClientContext * ctx,
    wc_Sha256 * sha,
    uint8_t * out
)
```

Async response half of a non-DMA SHA-256 Final.

Single-shot RecvResponse. Copies final digest into out, then resets sha state via wc_InitSha256_ex (preserving devId).

.3.2.62 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.

.3.2.63 function wh_Client_Sha256DmaUpdateRequest

```
int wh_Client_Sha256DmaUpdateRequest(
    whClientContext * ctx,
    wc_Sha256 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a DMA SHA-256 Update.

Buffers partial blocks on the client. Sends whole blocks via DMA to the server, with any assembled first block (from the partial buffer) as inline trailing data. Sets *requestSent to indicate whether a message was sent (false when all input was absorbed into the partial-block buffer).

.3.2.64 function wh_Client_Sha256DmaUpdateResponse

```
int wh_Client_Sha256DmaUpdateResponse(
    whClientContext * ctx,
    wc_Sha256 * sha
)
```

Async response half of a DMA SHA-256 Update.

Receives the server response and restores the updated SHA state from the inline response. Runs POST DMA cleanup for the input buffer.

.3.2.65 function wh_Client_Sha256DmaFinalRequest

```
int wh_Client_Sha256DmaFinalRequest(
    whClientContext * ctx,
    wc_Sha256 * sha
)
```

Async request half of a DMA SHA-256 Final.

Sends the partial-block tail as inline data with the resume state. No DMA addresses are used (the final hash is returned inline in the response).

.3.2.66 function wh_Client_Sha256DmaFinalResponse

```
int wh_Client_Sha256DmaFinalResponse(
    whClientContext * ctx,
    wc_Sha256 * sha,
    uint8_t * out
)
```

Async response half of a DMA SHA-256 Final.

Receives the final hash from the inline response and copies it to out.

.3.2.67 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.

.3.2.68 function wh_Client_Sha224UpdateRequest

```
int wh_Client_Sha224UpdateRequest(
    whClientContext * ctx,
    wc_Sha224 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a non-DMA SHA-224 Update.

Parameters:

- **ctx** Client context.
- **sha** SHA-224 context (buffer/buffLen updated on success).
- **in** Input data (may be NULL only if inLen == 0).
- **inLen** Input length. Must not exceed the per-call capacity (max inline + remaining buffer slack); use the blocking wrapper for arbitrary lengths.
- **requestSent** Set to true if a server request was sent and a matching Response call is required; false if the input was fully absorbed into sha->buffer and no round-trip was issued.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if inLen exceeds the per-call capacity (sha is left unchanged in that case).

Serializes and sends an Update request carrying as many full blocks as fit in the comm buffer (up to WH_MESSAGE_CRYPTO_SHA224_MAX_INLINE_UPDATE_SZ bytes), absorbing any leading bytes already buffered in sha->buffer. Any tail (<64 bytes) remaining after this call is stored in sha->buffer for the next call. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext (enforced by the comm layer's pending-request tracking). If *requestSent is true, the caller MUST call wh_Client_Sha224UpdateResponse before issuing any other async Request on the same ctx, including a Request using a different wc_Sha224 instance or a different algorithm.

.3.2.69 function wh_Client_Sha224UpdateResponse

```
int wh_Client_Sha224UpdateResponse(
    whClientContext * ctx,
    wc_Sha224 * sha
)
```

Async response half of a non-DMA SHA-224 Update.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, updates sha->digest/hiLen/loLen from the reply. MUST only be called if the matching Request returned requestSent == true.

.3.2.70 function wh_Client_Sha224FinalRequest

```
int wh_Client_Sha224FinalRequest(
    whClientContext * ctx,
    wc_Sha224 * sha
)
```

Async request half of a non-DMA SHA-224 Final.

Sends the current sha->buffer (0..63 bytes) as the last block.

.3.2.71 function wh_Client_Sha224FinalResponse

```
int wh_Client_Sha224FinalResponse(
    whClientContext * ctx,
    wc_Sha224 * sha,
    uint8_t * out
)
```

Async response half of a non-DMA SHA-224 Final.

Single-shot RecvResponse. Copies final digest into out, then resets sha state via wc_InitSha224_ex (preserving devId).

.3.2.72 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.

.3.2.73 function wh_Client_Sha224DmaUpdateRequest

```
int wh_Client_Sha224DmaUpdateRequest(
    whClientContext * ctx,
    wc_Sha224 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

.3.2.74 function wh_Client_Sha224DmaUpdateResponse

```
int wh_Client_Sha224DmaUpdateResponse(
    whClientContext * ctx,
    wc_Sha224 * sha
)
```

.3.2.75 function wh_Client_Sha224DmaFinalRequest

```
int wh_Client_Sha224DmaFinalRequest(
    whClientContext * ctx,
    wc_Sha224 * sha
)
```

.3.2.76 function wh_Client_Sha224DmaFinalResponse

```
int wh_Client_Sha224DmaFinalResponse(
    whClientContext * ctx,
    wc_Sha224 * sha,
    uint8_t * out
)
```

.3.2.77 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.

.3.2.78 function wh_Client_Sha384UpdateRequest

```
int wh_Client_Sha384UpdateRequest(
    whClientContext * ctx,
    wc_Sha384 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a non-DMA SHA-384 Update.

Parameters:

- **ctx** Client context.
- **sha** SHA-384 context (buffer/buffLen updated on success).
- **in** Input data (may be NULL only if inLen == 0).
- **inLen** Input length. Must not exceed the per-call capacity (max inline + remaining buffer slack); use the blocking wrapper for arbitrary lengths.

- **requestSent** Set to true if a server request was sent and a matching Response call is required; false if the input was fully absorbed into sha->buffer and no round-trip was issued.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if inLen exceeds the per-call capacity (sha is left unchanged in that case).

Serializes and sends an Update request carrying as many full blocks as fit in the comm buffer (up to WH_MESSAGE_CRYPTO_SHA384_MAX_INLINE_UPDATE_SZ bytes), absorbing any leading bytes already buffered in sha->buffer. Any tail (<128 bytes) remaining after this call is stored in sha->buffer for the next call. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext (enforced by the comm layer's pending-request tracking). If *requestSent is true, the caller MUST call wh_Client_Sha384UpdateResponse before issuing any other async Request on the same ctx, including a Request using a different wc_Sha384 instance or a different algorithm.

.3.2.79 function wh_Client_Sha384UpdateResponse

```
int wh_Client_Sha384UpdateResponse(
    whClientContext * ctx,
    wc_Sha384 * sha
)
```

Async response half of a non-DMA SHA-384 Update.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, updates sha->digest/hiLen/loLen from the reply. MUST only be called if the matching Request returned requestSent == true.

.3.2.80 function wh_Client_Sha384FinalRequest

```
int wh_Client_Sha384FinalRequest(
    whClientContext * ctx,
    wc_Sha384 * sha
)
```

Async request half of a non-DMA SHA-384 Final.

Sends the current sha->buffer (0..127 bytes) as the last block.

.3.2.81 function wh_Client_Sha384FinalResponse

```
int wh_Client_Sha384FinalResponse(
    whClientContext * ctx,
    wc_Sha384 * sha,
    uint8_t * out
)
```

Async response half of a non-DMA SHA-384 Final.

Single-shot RecvResponse. Copies final digest into out, then resets sha state via wc_InitSha384_ex (preserving devId).

.3.2.82 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.

.3.2.83 function wh_Client_Sha384DmaUpdateRequest

```
int wh_Client_Sha384DmaUpdateRequest(
    whClientContext * ctx,
    wc_Sha384 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

.3.2.84 function wh_Client_Sha384DmaUpdateResponse

```
int wh_Client_Sha384DmaUpdateResponse(
    whClientContext * ctx,
    wc_Sha384 * sha
)
```

.3.2.85 function wh_Client_Sha384DmaFinalRequest

```
int wh_Client_Sha384DmaFinalRequest(
    whClientContext * ctx,
    wc_Sha384 * sha
)
```

.3.2.86 function wh_Client_Sha384DmaFinalResponse

```
int wh_Client_Sha384DmaFinalResponse(
    whClientContext * ctx,
    wc_Sha384 * sha,
    uint8_t * out
)
```

.3.2.87 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.

.3.2.88 function wh_Client_Sha512UpdateRequest

```
int wh_Client_Sha512UpdateRequest(
    whClientContext * ctx,
    wc_Sha512 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

Async request half of a non-DMA SHA-512 Update.

Parameters:

- **ctx** Client context.
- **sha** SHA-512 context (buffer/buffLen updated on success).
- **in** Input data (may be NULL only if inLen == 0).
- **inLen** Input length. Must not exceed the per-call capacity (max inline + remaining buffer slack); use the blocking wrapper for arbitrary lengths.
- **requestSent** Set to true if a server request was sent and a matching Response call is required; false if the input was fully absorbed into sha->buffer and no round-trip was issued.

Return: WH_ERROR_OK on success, WH_ERROR_BADARGS if inLen exceeds the per-call capacity (sha is left unchanged in that case).

Serializes and sends an Update request carrying as many full blocks as fit in the comm buffer (up to WH_MESSAGE_CRYPTO_SHA512_MAX_INLINE_UPDATE_SZ bytes), absorbing any leading bytes already buffered in sha->buffer. Any tail (<128 bytes) remaining after this call is stored in sha->buffer for the next call. Does NOT wait for a reply.

Contract: at most one outstanding async request may be in flight per whClientContext (enforced by the comm layer's pending-request tracking). If *requestSent is true, the caller MUST call wh_Client_Sha512UpdateResponse before issuing any other async Request on the same ctx, including a Request using a different wc_Sha512 instance or a different algorithm.

.3.2.89 function wh_Client_Sha512UpdateResponse

```
int wh_Client_Sha512UpdateResponse(
    whClientContext * ctx,
    wc_Sha512 * sha
)
```

Async response half of a non-DMA SHA-512 Update.

Single-shot RecvResponse; returns WH_ERROR_NOTREADY if the server has not yet replied. On success, updates sha->digest/hiLen/loLen from the reply. MUST only be called if the matching Request returned requestSent == true.

.3.2.90 function wh_Client_Sha512FinalRequest

```
int wh_Client_Sha512FinalRequest(
    whClientContext * ctx,
    wc_Sha512 * sha
)
```

Async request half of a non-DMA SHA-512 Final.

Sends the current sha->buffer (0..127 bytes) as the last block.

.3.2.91 function wh_Client_Sha512FinalResponse

```
int wh_Client_Sha512FinalResponse(
    whClientContext * ctx,
    wc_Sha512 * sha,
    uint8_t * out
)
```

Async response half of a non-DMA SHA-512 Final.

Single-shot RecvResponse. Copies final digest into out, then resets sha state via wc_InitSha512_ex (preserving devId and hashType).

.3.2.92 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.

.3.2.93 function wh_Client_Sha512DmaUpdateRequest

```
int wh_Client_Sha512DmaUpdateRequest(
    whClientContext * ctx,
```

```
    wc_Sha512 * sha,
    const uint8_t * in,
    uint32_t inLen,
    bool * requestSent
)
```

.3.2.94 function wh_Client_Sha512DmaUpdateResponse

```
int wh_Client_Sha512DmaUpdateResponse(
    whClientContext * ctx,
    wc_Sha512 * sha
)
```

.3.2.95 function wh_Client_Sha512DmaFinalRequest

```
int wh_Client_Sha512DmaFinalRequest(
    whClientContext * ctx,
    wc_Sha512 * sha
)
```

.3.2.96 function wh_Client_Sha512DmaFinalResponse

```
int wh_Client_Sha512DmaFinalResponse(
    whClientContext * ctx,
    wc_Sha512 * sha,
    uint8_t * out
)
```

.3.2.97 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

.3.2.98 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.

.3.2.99 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.

.3.2.100 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.

.3.2.101 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.

.3.2.102 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.

.3.2.103 function wh_Client_MlDsaSign

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

Sign a message using a ML-DSA private key.

Parameters:

- **ctx** Pointer to the client context structure.
- **in** Pointer to the message to sign.
- **in_len** Length of the message in bytes.
- **out** Buffer to store the signature.
- **out_len** Pointer to size of output buffer, updated with actual size.
- **key** Pointer to the ML-DSA key structure.
- **context** Optional FIPS 204 context string for domain separation, or NULL for no context.
- **contextLen** Length of the context string (max 255).
- **preHashType** Hash type for HashML-DSA (e.g. WC_HASH_TYPE_SHA256), or WC_HASH_TYPE_NONE for pure ML-DSA.

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

This function signs a message using a ML-DSA private key stored in the HSM.

.3.2.104 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,
    const byte * context,
    byte contextLen,
    word32 preHashType
)
```

Verify a ML-DSA signature.

Parameters:

- **ctx** Pointer to the client context structure.
- **sig** Pointer to the signature to verify.
- **sig_len** Length of the signature in bytes.
- **msg** Pointer to the original message.
- **msg_len** Length of the message in bytes.
- **res** Pointer to store verification result (1=success, 0=failure).
- **key** Pointer to the ML-DSA key structure.
- **context** Optional FIPS 204 context string for domain separation, or NULL for no context.
- **contextLen** Length of the context string (max 255).
- **preHashType** Hash type for HashML-DSA (e.g. WC_HASH_TYPE_SHA256), or WC_HASH_TYPE_NONE for pure ML-DSA.

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

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

.3.2.105 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.

.3.2.106 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.

.3.2.107 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.

.3.2.108 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.

.3.2.109 function wh_Client_MlDsaSignDma

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

Sign a message using ML-DSA with DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **in** Pointer to the message to sign.
- **in_len** Length of the message in bytes.
- **out** Pointer to store the signature.
- **out_len** On input, size of out buffer. On output, length of signature.
- **key** Pointer to the ML-DSA key structure.
- **context** Optional FIPS 204 context string for domain separation, or NULL for no context.
- **contextLen** Length of the context string (max 255).
- **preHashType** Hash type for HashML-DSA (e.g. WC_HASH_TYPE_SHA256), or WC_HASH_TYPE_NONE for pure ML-DSA.

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

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

.3.2.110 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,
    const byte * context,
    byte contextLen,
    word32 preHashType
)
```

Verify a ML-DSA signature with DMA.

Parameters:

- **ctx** Pointer to the client context structure.
- **sig** Pointer to the signature to verify.
- **sig_len** Length of the signature in bytes.
- **msg** Pointer to the message that was signed.
- **msg_len** Length of the message in bytes.
- **res** Result of verification (1 = success, 0 = failure).
- **key** Pointer to the ML-DSA key structure.
- **context** Optional FIPS 204 context string for domain separation, or NULL for no context.
- **contextLen** Length of the context string (max 255).
- **preHashType** Hash type for HashML-DSA (e.g. WC_HASH_TYPE_SHA256), or WC_HASH_TYPE_NONE for pure ML-DSA.

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

This function verifies a ML-DSA signature with DMA.

.3.2.111 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>
#include <stdbool.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"
```

```c
#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);

int wh_Client_AesCtrDma(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);

int wh_Client_AesEcbDma(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);

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

int wh_Client_AesCbcRequest(whClientContext* ctx, Aes* aes, int enc,
                            const uint8_t* in, uint32_t len);

int wh_Client_AesCbcResponse(whClientContext* ctx, Aes* aes, uint8_t* out,
                             uint32_t* out_size);
#endif /* HAVE_AES_CBC */

#ifdef HAVE_AESGCM
int wh_Client_AesGcm(whClientContext* ctx,
        Aes* aes, int enc,
        const uint8_t* in, uint32_t len,
        const uint8_t* iv, uint32_t iv_len,
        const uint8_t* authin, uint32_t authin_len,
        const uint8_t* dec_tag, uint8_t* enc_tag, uint32_t tag_len,
        uint8_t* out);

int wh_Client_AesGcmDma(whClientContext* ctx, Aes* aes, int enc,
                        const uint8_t* in, uint32_t len, const uint8_t* iv,
                        uint32_t iv_len, const uint8_t* authin,
                        uint32_t authin_len, const uint8_t* dec_tag,
                        uint8_t* enc_tag, uint32_t tag_len, uint8_t* out);
#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_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_Sha256UpdateRequest(whClientContext* ctx, wc_Sha256* sha,
                                  const uint8_t* in, uint32_t inLen,
                                  bool* requestSent);

int wh_Client_Sha256UpdateResponse(whClientContext* ctx, wc_Sha256* sha);

int wh_Client_Sha256FinalRequest(whClientContext* ctx, wc_Sha256* sha);

int wh_Client_Sha256FinalResponse(whClientContext* ctx, wc_Sha256* sha,
                                  uint8_t* out);

int wh_Client_Sha256Dma(whClientContext* ctx, wc_Sha256* sha, const uint8_t*
 ↪  in,
                        uint32_t inLen, uint8_t* out);

#ifdef WOLFHSM_CFG_DMA
int wh_Client_Sha256DmaUpdateRequest(whClientContext* ctx, wc_Sha256* sha,
                                     const uint8_t* in, uint32_t inLen,
                                     bool* requestSent);

int wh_Client_Sha256DmaUpdateResponse(whClientContext* ctx, wc_Sha256* sha);

int wh_Client_Sha256DmaFinalRequest(whClientContext* ctx, wc_Sha256* sha);

int wh_Client_Sha256DmaFinalResponse(whClientContext* ctx, wc_Sha256* sha,
                                     uint8_t* out);
#endif /* WOLFHSM_CFG_DMA */

#endif /* !NO_SHA256 */

#if defined(WOLFSSL_SHA224)
int wh_Client_Sha224(whClientContext* ctx, wc_Sha224* sha, const uint8_t* in,
                     uint32_t inLen, uint8_t* out);

int wh_Client_Sha224UpdateRequest(whClientContext* ctx, wc_Sha224* sha,
                                  const uint8_t* in, uint32_t inLen,
                                  bool* requestSent);

int wh_Client_Sha224UpdateResponse(whClientContext* ctx, wc_Sha224* sha);

int wh_Client_Sha224FinalRequest(whClientContext* ctx, wc_Sha224* sha);

int wh_Client_Sha224FinalResponse(whClientContext* ctx, wc_Sha224* sha,
                                  uint8_t* out);

int wh_Client_Sha224Dma(whClientContext* ctx, wc_Sha224* sha, const uint8_t*
 ↪  in,
                        uint32_t inLen, uint8_t* out);

#ifdef WOLFHSM_CFG_DMA
```

```
int wh_Client_Sha224DmaUpdateRequest(whClientContext* ctx, wc_Sha224* sha,
                                     const uint8_t* in, uint32_t inLen,
                                     bool* requestSent);
int wh_Client_Sha224DmaUpdateResponse(whClientContext* ctx, wc_Sha224* sha);
int wh_Client_Sha224DmaFinalRequest(whClientContext* ctx, wc_Sha224* sha);
int wh_Client_Sha224DmaFinalResponse(whClientContext* ctx, wc_Sha224* sha,
                                     uint8_t* out);
#endif /* WOLFHSM_CFG_DMA */

#endif /* WOLFSSL_SHA224 */

#if defined(WOLFSSL_SHA384)
int wh_Client_Sha384(whClientContext* ctx, wc_Sha384* sha, const uint8_t* in,
                     uint32_t inLen, uint8_t* out);

int wh_Client_Sha384UpdateRequest(whClientContext* ctx, wc_Sha384* sha,
                                  const uint8_t* in, uint32_t inLen,
                                  bool* requestSent);

int wh_Client_Sha384UpdateResponse(whClientContext* ctx, wc_Sha384* sha);

int wh_Client_Sha384FinalRequest(whClientContext* ctx, wc_Sha384* sha);

int wh_Client_Sha384FinalResponse(whClientContext* ctx, wc_Sha384* sha,
                                  uint8_t* out);

int wh_Client_Sha384Dma(whClientContext* ctx, wc_Sha384* sha, const uint8_t*
 ↪  in,
                        uint32_t inLen, uint8_t* out);

#ifdef WOLFHSM_CFG_DMA
int wh_Client_Sha384DmaUpdateRequest(whClientContext* ctx, wc_Sha384* sha,
                                     const uint8_t* in, uint32_t inLen,
                                     bool* requestSent);
int wh_Client_Sha384DmaUpdateResponse(whClientContext* ctx, wc_Sha384* sha);
int wh_Client_Sha384DmaFinalRequest(whClientContext* ctx, wc_Sha384* sha);
int wh_Client_Sha384DmaFinalResponse(whClientContext* ctx, wc_Sha384* sha,
                                     uint8_t* out);
#endif /* WOLFHSM_CFG_DMA */

#endif /* WOLFSSL_SHA384 */

#if defined(WOLFSSL_SHA512)
int wh_Client_Sha512(whClientContext* ctx, wc_Sha512* sha, const uint8_t* in,
                     uint32_t inLen, uint8_t* out);

int wh_Client_Sha512UpdateRequest(whClientContext* ctx, wc_Sha512* sha,
                                  const uint8_t* in, uint32_t inLen,
                                  bool* requestSent);

int wh_Client_Sha512UpdateResponse(whClientContext* ctx, wc_Sha512* sha);

int wh_Client_Sha512FinalRequest(whClientContext* ctx, wc_Sha512* sha);
```

```
int wh_Client_Sha512FinalResponse(whClientContext* ctx, wc_Sha512* sha,
                                  uint8_t* out);

int wh_Client_Sha512Dma(whClientContext* ctx, wc_Sha512* sha, const uint8_t*
 ↪  in,
                        uint32_t inLen, uint8_t* out);

#ifdef WOLFHSM_CFG_DMA
int wh_Client_Sha512DmaUpdateRequest(whClientContext* ctx, wc_Sha512* sha,
                                     const uint8_t* in, uint32_t inLen,
                                     bool* requestSent);
int wh_Client_Sha512DmaUpdateResponse(whClientContext* ctx, wc_Sha512* sha);
int wh_Client_Sha512DmaFinalRequest(whClientContext* ctx, wc_Sha512* sha);
int wh_Client_Sha512DmaFinalResponse(whClientContext* ctx, wc_Sha512* sha,
                                     uint8_t* out);
#endif /* WOLFHSM_CFG_DMA */

#endif /* WOLFSSL_SHA512 */

#ifdef 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,
                            const byte* context, byte contextLen,
                            word32 preHashType);
int wh_Client_MlDsaVerify(whClientContext* ctx, const byte* sig,
                              word32 sig_len, const byte* msg, word32 msg_len,
                              int* res, MlDsaKey* key, const byte* context,
                              byte contextLen, word32 preHashType);

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, const byte* context,
                               byte contextLen, word32 preHashType);

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

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.

	Name
int	**wh_Server_HandleRequestMessage**(whServerContext * server)Handles incoming request messages and dispatches them to the appropriate handlers.
int	**wh_Server_Cleanup**(whServerContext * server)Cleans up the server context and associated resources.
int	**wh_Server_RegisterCustomCb**(whServerContext * server, uint16_t action, whServerCustomCb handler)Registers a custom callback handler for a specific action.
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.

	Name
int	**wh_Server_NvmLock**(whServerContext * server)
int	**wh_Server_NvmUnlock**(whServerContext * server)

.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_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.

.4.2.6 function wh_Server_Cleanup

```
int wh_Server_Cleanup(
    whServerContext * server
)
```

Cleans up the server context and associated resources.

Parameters:

- **server** Pointer to the server context.

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

This function releases any resources associated with the server context, including communication server resources. It resets the server context to its initial state.

.4.2.7 function wh_Server_RegisterCustomCb

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

Registers a custom callback handler for a specific action.

Parameters:

- **server** Pointer to the server context.
- **actionId** The action ID for which the callback is being registered.
- **cb** The custom callback handler to register.

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

Server custom callback functions

This function allows the server to register a custom callback handler for a specific action ID. The callback will be invoked when a request with the corresponding action ID is received.

.4.2.8 function wh_Server_HandleCustomCbRequest

```
int wh_Server_HandleCustomCbRequest(
    whServerContext * server,
    uint16_t magic,
    uint16_t action,
    uint16_t seq,
    uint16_t req_size,
    const void * req_packet,
    uint16_t * out_resp_size,
    void * resp_packet
)
```

Handles incoming custom callback requests.

Parameters:

- **server** Pointer to the server context.
- **magic** The magic number for the request.
- **action** The action ID of the request.
- **seq** The sequence number of the request.
- **req_size** The size of the request packet.
- **req_packet** Pointer to the request packet data.
- **out_resp_size** Pointer to store the size of the response packet.
- **resp_packet** Pointer to store the response packet data.

Return: int Returns WH_ERROR_OK on success, WH_ERROR_BADARGS if the arguments are invalid, WH_ERROR_ABORTED if the request is malformed, or a negative error code on failure.

This function processes incoming custom callback requests by invoking the registered custom callback handler for the specified action. It translates the request and response messages and sends the appropriate response back to the client.

.4.2.9 function wh_Server_DmaRegisterCb

```
int wh_Server_DmaRegisterCb(
    struct whServerContext_t * server,
```

```
    whServerDmaClientMemCb cb
)
```

Registers a custom client DMA callback.

Parameters:

- **server** Pointer to the server context.
- **cb** The custom DMA callback handler to register.

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

Server DMA functions

This function allows the server to register a custom callback handler for processing client memory operations. The callback will be invoked during DMA operations to transform client addresses, manipulate caches, etc.

.4.2.10 function wh_Server_DmaRegisterMemCopyCb

```
int wh_Server_DmaRegisterMemCopyCb(
    whServerContext * server,
    whServerDmaMemCopyCb cb
)
```

Registers a custom memory copy callback for DMA operations. This function allows the server to register a callback that will be invoked during DMA memory copy operations. The callback overrides the use of memcpy when copying to and from client memory. This is useful if standard memcpy cannot be used to copy data back and forth between the client, even after client addresses are transformed through the standard DMA callbacks (e.g. if client memory can only be accessed though a hardware FIFO or register interface)

Parameters:

- **server** Pointer to the server context.
- **cb** The custom memory copy callback handler to register.

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

.4.2.11 function wh_Server_DmaRegisterAllowList

```
int wh_Server_DmaRegisterAllowList(
    struct whServerContext_t * server,
    const whServerDmaAddrAllowList * allowlist
)
```

Registers the allowable client read/write addresses for DMA.

Parameters:

- **server** Pointer to the server context.
- **allowlist** Pointer to the list of allowable client addresses.

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

This function allows the server to register a list of allowable client addresses for DMA read and write operations. The server will check these addresses during DMA operations to ensure they are within the allowed range for the client

.4.2.12 function wh_Server_DmaCheckMemOperAllowed

```
int wh_Server_DmaCheckMemOperAllowed(
    const struct whServerContext_t * server,
    whServerDmaOper oper,
    void * addr,
    size_t size
)
```

Checks if a DMA memory operation is allowed based on the server's allowlist.

Parameters:

- **server** Pointer to the server context.
- **oper** The DMA operation type (e.g., read or write).
- **addr** The address to be checked.
- **size** The size of the memory operation.

Return: int Returns WH_ERROR_OK if the operation is allowed, WH_ERROR_BADARGS if the arguments are invalid, or WH_ERROR_ACCESS if the operation is not allowed.

This function verifies whether a specified DMA memory operation is permitted by checking the operation type and the address range against the server's registered allowlist. If no allowlist is registered, the operation is allowed.

.4.2.13 function wh_Server_DmaProcessClientAddress

```
int wh_Server_DmaProcessClientAddress(
    struct whServerContext_t * server,
    uintptr_t clientAddr,
    void ** serverPtr,
    size_t len,
    whServerDmaOper oper,
    whServerDmaFlags flags
)
```

Processes a client address for DMA operations, using the native pointer size of the system.

Parameters:

- **server** Pointer to the server context.
- **clientAddr** The client address to be processed.
- **serverPtr** Pointer to store the transformed server address.
- **len** The length of the memory operation.
- **oper** The DMA operation type (e.g., read or write).
- **flags** Flags for the DMA operation.

Return: int Returns WH_ERROR_OK on success, WH_ERROR_BADARGS if the arguments are invalid, or a negative error code on failure.

This function transforms a client address for DMA operations. It performs user-supplied address transformations, cache manipulations, and checks the transformed address against the server's allowlist if registered.

.4.2.14 function wh_Server_NvmLock

```
int wh_Server_NvmLock(
    whServerContext * server
)
```

Server NVM Locking API for handler-level thread safety

.4.2.15 function wh_Server_NvmUnlock

```
int wh_Server_NvmUnlock(
    whServerContext * server
)
```

.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"
#ifdef WOLFHSM_CFG_ENABLE_AUTHENTICATION
#include "wolfhsm/wh_auth.h"
```

```c
#endif /* WOLFHSM_CFG_ENABLE_AUTHENTICATION */
#include "wolfhsm/wh_message_customcb.h"
#include "wolfhsm/wh_log.h"
#ifdef WOLFHSM_CFG_DMA
#include "wolfhsm/wh_dma.h"
#endif /* WOLFHSM_CFG_DMA */

#ifndef WOLFHSM_CFG_NO_CRYPTO
#include "wolfssl/wolfcrypt/settings.h"
#include "wolfssl/wolfcrypt/types.h"
#include "wolfssl/wolfcrypt/random.h"
#include "wolfssl/wolfcrypt/rsa.h"
#include "wolfssl/wolfcrypt/ecc.h"
#include "wolfssl/wolfcrypt/curve25519.h"
#include "wolfssl/wolfcrypt/cryptocb.h"
#include "wolfssl/wolfcrypt/sha256.h"
#endif /* !WOLFHSM_CFG_NO_CRYPTO */

#ifdef WOLFHSM_CFG_SHE_EXTENSION
#include "wolfhsm/wh_she_common.h"
#include "wolfhsm/wh_server_she.h"
#endif


#ifndef WOLFHSM_CFG_NO_CRYPTO

typedef struct whServerCryptoContext {
#ifndef WC_NO_RNG
    WC_RNG rng[1];
#else
    /* Placeholder to prevent empty struct in C90 */
    uint8_t WH_PAD[1];
#endif
} whServerCryptoContext;


#endif /* !WOLFHSM_CFG_NO_CRYPTO */

/* Type definition for a custom server callback  */
typedef int (*whServerCustomCb)(
    whServerContext* server,             /* points to dispatching server ctx */
    const whMessageCustomCb_Request* req, /* request from client to callback */
    whMessageCustomCb_Response*      resp /* response from callback to client */
);


#ifdef WOLFHSM_CFG_DMA

/* Maintain existing naming for common DMA types */
typedef whDmaAddrAllowList whServerDmaAddrAllowList;
typedef whDmaOper          whServerDmaOper;
typedef whDmaFlags         whServerDmaFlags;
typedef whDmaAddr          whServerDmaAddr;
typedef whDmaAddrList      whServerDmaAddrList;
```

```
#ifdef WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY
typedef whDmaCopyOper whServerDmaCopyOper;
#endif /* WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY */

/* DMA callbacks invoked internally by wolfHSM before and after every client
 * memory operation. */
typedef int (*whServerDmaClientMemCb)(struct whServerContext_t* server,
                                      uintptr_t clientAddr, void** serverPtr,
                                      size_t len, whServerDmaOper oper,
                                      whServerDmaFlags flags);

#ifdef WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY
/* DMA callback invoked to copy from the client */
typedef int (*whServerDmaMemCopyCb)(struct whServerContext_t* server,
                                    uintptr_t clientAddr, uintptr_t serverPtr,
                                     size_t len, whServerDmaCopyOper oper,
                                     whServerDmaFlags flags);
#endif /* WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY */

/* Server DMA configuration struct for initializing a server */
typedef struct {
    whServerDmaClientMemCb          cb;               /* DMA callback */
#ifdef WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY
    whServerDmaMemCopyCb            memCopyCb;       /* DMA memory copy callback
 ↪  */
#endif /* WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY */
    const whServerDmaAddrAllowList* dmaAddrAllowList; /* allowed addresses */
} whServerDmaConfig;

typedef struct {
    whServerDmaClientMemCb          cb;               /* DMA callback */
#ifdef WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY
    whServerDmaMemCopyCb            memCopyCb;       /* DMA memory copy callback
 ↪  */
#endif /* WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY */
    const whServerDmaAddrAllowList* dmaAddrAllowList; /* allowed addresses */
} whServerDmaContext;
#endif /* WOLFHSM_CFG_DMA */

typedef struct whServerConfig_t {
    whCommServerConfig* comm_config;
    whNvmContext*       nvm;
#ifdef WOLFHSM_CFG_ENABLE_AUTHENTICATION
    whAuthContext* auth;
#endif /* WOLFHSM_CFG_ENABLE_AUTHENTICATION */

#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;
#ifdef WOLFHSM_CFG_ENABLE_AUTHENTICATION
    whAuthContext* auth;
#endif /* WOLFHSM_CFG_ENABLE_AUTHENTICATION */
    whCommServer  comm[1];
#ifndef WOLFHSM_CFG_NO_CRYPTO
    whServerCryptoContext* crypto;
    int                    devId;
    whKeyCacheContext      localCache; /* Unified cache structure */
#ifdef WOLFHSM_CFG_SHE_EXTENSION
    whServerSheContext* she;
#endif
#endif /* !WOLFHSM_CFG_NO_CRYPTO */
    whServerCustomCb   customHandlerTable[WOLFHSM_CFG_SERVER_CUSTOMCB_COUNT];
#ifdef WOLFHSM_CFG_DMA
    whServerDmaContext dma;
#endif /* WOLFHSM_CFG_DMA */
    int connected;
#ifdef WOLFHSM_CFG_LOGGING
    whLogContext log;
#endif /* WOLFHSM_CFG_LOGGING */
};


/* 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);

int wh_Server_HandleRequestMessage(whServerContext* server);

int wh_Server_Cleanup(whServerContext* server);

int wh_Server_RegisterCustomCb(whServerContext* server, uint16_t action,
```

```
                                whServerCustomCb handler);

int wh_Server_HandleCustomCbRequest(whServerContext* server, uint16_t magic,
                                    uint16_t action, uint16_t seq,
                                    uint16_t req_size, const void* req_packet,
                                   uint16_t* out_resp_size, void* resp_packet);

#ifdef WOLFHSM_CFG_DMA

int wh_Server_DmaRegisterCb(struct whServerContext_t* server,
                            whServerDmaClientMemCb    cb);


#ifdef WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY
int wh_Server_DmaRegisterMemCopyCb(whServerContext*     server,
                                   whServerDmaMemCopyCb cb);
#endif /* WOLFHSM_CFG_DMA_CUSTOM_CLIENT_COPY */

int wh_Server_DmaRegisterAllowList(struct whServerContext_t*       server,
                                   const whServerDmaAddrAllowList* allowlist);

int wh_Server_DmaCheckMemOperAllowed(const struct whServerContext_t* server,
                                     whServerDmaOper oper, void* addr,
                                     size_t size);

int wh_Server_DmaProcessClientAddress(struct whServerContext_t* server,
                                      uintptr_t clientAddr, void** serverPtr,
                                      size_t len, whServerDmaOper oper,
                                      whServerDmaFlags flags);

int whServerDma_CopyFromClient(struct whServerContext_t* server,
                               void* serverPtr, uintptr_t clientAddr,
                               size_t len, whServerDmaFlags flags);

int whServerDma_CopyToClient(struct whServerContext_t* server,
                             uintptr_t clientAddr, void* serverPtr, size_t len,
                             whServerDmaFlags flags);
#endif /* WOLFHSM_CFG_DMA */

#ifdef WOLFHSM_CFG_THREADSAFE
int wh_Server_NvmLock(whServerContext* server);
int wh_Server_NvmUnlock(whServerContext* server);
#define WH_SERVER_NVM_LOCK(server) wh_Server_NvmLock(server)
#define WH_SERVER_NVM_UNLOCK(server) wh_Server_NvmUnlock(server)
#else
#define WH_SERVER_NVM_LOCK(server) (WH_ERROR_OK)
#define WH_SERVER_NVM_UNLOCK(server) (WH_ERROR_OK)
#endif

#endif /* !WOLFHSM_WH_SERVER_H_ */
```