wolfHSM Documentation

2025-01-23

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。

```c
#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 {
   .devID = INVALID_DEVID; /* あるいは、カスタム暗号コールバック devID を設定 */
};

/* サーバー設定の割り当てと初期化 */
whServerConfig serverCfg = {
       .comm   = commServerCfg,
       .nvm    = nvmCtx,
       .crypto = cryptoCtx,
};

/* 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。

```c
#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。wolfHSM がリモート HSM 実行でサポートしているアルゴリズムの完全なリストについては、サポートされている wolfCrypt アルゴリズム (後日提供予定)をご参照ください。

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 (現在対応中です。) 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 (現在対応中です。) 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。

A wolfHSM API reference

