wolfBoot Documentation

2025-04-22

Contents

1 イントロダクション

wolfBootは、32 ビットマイクロコントローラー向けのポータブルで OS に依存しないセキュアなブートローダーソリューションであり、ファームウェア認証用の wolfCrypt に依存して、ファームウェアの更新メカニズムを提供します。

ブートローダーと小さな HAL API の最小限設計により、wolfBoot はあらゆる OS または裸の金属アプリケーションから完全に独立しており、安全なファームウェア更新メカニズムを提供するために、既存の埋め込みソフトウェアプロジェクトに簡単に移植および統合できます。

機能

- フラッシュデバイスのマルチスロットパーティション
- ファームウェアイメージの整合性検証
- wolfCrypt のデジタル署名アルゴリズム (DSA) を使用したファームウェアイメージの信頼性検証
- 最小限のハードウェア抽象化レイヤー (HAL) インターフェースは、さまざまなベンダー/MCU の移植性を促進するためのインターフェース
- セカンダリスロットからプライマリスロットにイメージをコピー/スワップして、ファームウェアの更新操作に同意する
- プライマリスロットでのファームウェアイメージのインプレースチェーンロード
- TPM のサポート
- 測定されたブートサポート、ファームウェアイメージハッシュの TPM プラットフォーム構成レジスタ (PCR) への保存

コンポーネント

wolfBoot GitHub リポジトリには、次のコンポーネントが含まれています。

- wolfBoot ブートローダ
- 鍵生成とイメージ署名ツール (python 3.x および wolfcrypt-py https://github.com/wolfssl/wolfcrypt-py が必要です)
- 非 OS のテストアプリケーション

2 wolfBoot のビルド

wolfBoot は、さまざまな種類の組み込みシステムにわたってポータブルです。プラットフォーム固有のコードは、hal ディレクトリの下にある単一のファイルに含まれており、ハードウェア固有の機能を実装します。

特定のコンパイルオプションを有効にするには、make コマンドと共に環境変数を使用します。

```
make CORTEX_M0=1
```

別の方法として、wolfBoot のルートディレクトリに.config ファイルを提供できます。?= オペレーターを使用して定義されている限り、コマンドラインオプションは.config オプションで優先されます。

```
WOLFBOOT_PARTITION_BOOT_ADDRESS?=0x14000
```

2.1 コンフィギュレーションファイルの新規作成

デフォルトパラメーターのセットを備えた新しい.config ファイルは、make config を実行することで生成できます。ビルドスクリプトは、各構成パラメーターのデフォルト値を入力するように要求してきます。[] の間に示されている現在の値を確認します。

.config ファイルが設置されると、パラメーターなしで make を実行すると、デフォルトのコンパイル時オプションが変更されます。

.config は、テキストエディターで変更して、後でデフォルトのオプションを変更できます。

2.2 プラットフォームの選択

ネイティブにサポートされている場合、ターゲットプラットフォームは TARGET 変数を使用して指定できます。Make は、正しいコンパイルオプションを自動的に選択し、選択したターゲットに対応する HAL を含めます。

現在サポートされているプラットフォームのリストについては、HALの章を参照してください。

新しいプラットフォームを追加するには、hal ディレクトリに対応する HAL ドライバーとリンカースクリプトファイルを作成するだけです。

指定されていない場合のデフォルトオプション：TARGET=stm32f4

一部のプラットフォームには、アーキテクチャに固有の追加オプションが必要です。デフォルトでは、wolfBoot は ARM Cortex-M3/4/7 用にコンパイルされています。cortex-m0 をコンパイルするには、次を使用します。

```
CORTEX_M0=1
```

2.2.1 フラッシュパーティション

ファイル include/target.h は、構成されたフラッシュジオメトリ、パーティションサイズ、ターゲットシステムのオフセットに従って生成されます。次の値を、コマンドラインを介して、または.config ファイルを使用して、目的のフラッシュ構成を提供するように設定する必要があります。

- WOLFBOOT_SECTOR_SIZE

この変数は、フラッシュメモリ上の物理セクターのサイズを決定します。ブロックサイズが異なる領域が 2 つのパーティションに使用されている場合 (たとえば、外部フラッシュでパーティションを更新する)、この変数は 2 つのパーティション間で共有される最大のセクターのサイズを示す必要があります。

wolfBoot は、ファームウェアイメージを所定の位置に交換するときに、この値を最小ユニットとして使用します。このため、この値はスワップパーティションのサイズを設定するためにも使用されます。

- WOLFBOOT_PARTITION_BOOT_ADDRESS

これは、新しいフラッシュセクターの開始に合わせたブートパーティションの開始アドレスです。アプリケーションコードは、パーティションヘッダーサイズ (ED25519 の場合は 256B および ECC 署名ヘッダー) に等しく、さらにオフセットされた後に開始されます。

- WOLFBOOT_PARTITION_UPDATE_ADDRESS

これは、更新パーティションの開始アドレスです。EXT_FLASH オプションを介して外部メモリを使用する場合、この変数には、外部メモリアドレス指定可能なスペースの先頭からの更新パーティションのオフセットが含まれます。

- WOLFBOOT_PARTITION_SWAP_ADDRESS

wolfBoot で使用されているスワップ間隔のアドレスは、反転可能な更新を実行するために、2 つのファームウェアイメージを所定の位置に交換します。スワップパーティションのサイズは、フラッシュ上のまったく 1 つのセクターです。外部メモリが使用される場合、変数にはアドレス指定可能なスペースの先頭からスワップ領域のオフセットが含まれます。

- WOLFBOOT_PARTITION_SIZE

ブートと更新パーティションのサイズ。サイズは両方のパーティションで同じです。

2.3 ブートローダー機能

wolfBoot コンピレーション中に、多くの特性をオン/オフにすることができます。ブートローダーのサイズ、パフォーマンス、アクティブ化された機能は、コンパイル時間フラグの影響を受けます。

2.3.1 DSA アルゴリズムの変更

デフォルトでは、wolfBoot は ED25519 DSA を使用するようにコンパイルされています。ED25519 の実装は小さく、ブートアップ時間の観点からは良い妥協点を与えます。

curve P-256 の ECDSA を使用すると、パフォーマンスを向上できます。ECC256 サポートを有効にするには、make コマンドに以下のオプションを指定してください：

```
SIGN=ECC256
```

RSA でも異なる鍵長に変更できます。RSA2048 または RSA4096 をアクティブにするには、それぞれ以下を指定します：

```
SIGN=RSA2048
```

あるいは

```
SIGN=RSA4096
```

ED448 は SIGN=ED448 でサポートされています。

SIGN 変数の値が提供されていない場合、デフォルトオプションは以下の値です。

```
SIGN=ED25519
```

DSA アルゴリズムを変更すると、鍵生成とファームウェアの署名のために異なるツールセットをコンパイルします。

tools ディレクトリに、対応する鍵生成およびファームウェア署名ツールが格納されています。

以下を明示的に使用して、ファームウェアイメージの認証を無効にすることができます。

```
SIGN=NONE
```

これにより、パブリック鍵認証セキュアーブートをサポートせずに最小限のブートローダーをコンパイルします。

2.3.2 インクリメンタル更新

wolfBoot はインクリメンタル更新をサポートしています。この機能を有効にするには、DELTA_UPDATES=1 でコンパイルします。

署名ツールが--delta オプションで呼び出されたときに追加ファイルが生成されます。これは、現在ターゲットで実行されている古いファームウェアの違いのみを含み、新しいバージョンで実行されます。

詳細と例については、ファームウェアの更新セクションを参照してください。

2.3.3 デバッグシンボルの有効化

ブートローダーをデバッグするには、DEBUG=1 でコンパイルするだけです。ですがブートロードのサイズは一貫して増加します。そのため、WOLFBOOT_PARTITION_BOOT_ADDRESS の前にフラッシュの開始時に十分なスペースがあることを確認してください。

2.3.4 割り込みベクトルの再配置の無効化

一部のプラットフォームでは、起動する前に割り込みベクトルの再配置を回避するのが便利かもしれません。これは、システム上のコンポーネントが別の段階で割り込み再配置を既に管理している場合、または割り込みベクターの再配置をサポートしないこれらのプラットフォームで既に管理している場合に必要です。

割り込みベクトルテーブルの再配置を無効にするには、VTOR=0 でコンパイルします。デフォルトでは、wolfBoot はベクトル再配置オフセットレジスタ (VTOR) にオフセットを設定して割り込みベクターを再配置します。

2.3.5 スタック使用の制限

デフォルトでは、wolfBoot はメモリの割り当てを必要としません。スタックを使用してすべての操作を実行しています。アルゴリズムで使用されるスタックスペースはコンパイル時間で予測できますが、選択したアルゴリズムに応じて、スタックスペースの量は比較的大きくなります。

一部のターゲットは、一般的に、またはブートローダーステージ専用の構成で、スタックスペースとして使用する限られた量の RAM を提供します。

これらの場合、WOLFBOOT_SMALL_STACK=1 をアクティブにすると便利な場合があります。このオプションを使用すると、コンパイル時に固定サイズのプールが作成され、暗号化の実装に必要なオブジェクトの割り当てを支援します。WOLFBOOT_SMALL_STACK=1 でコンパイルされると、wolfBoot はスタックの使用量を大幅に削減し、専用の静的に割り当てられた事前サイズのメモリ領域を割り当てることにより、動的メモリ割り当てをシミュレートします。

2.3.6 現在実行中ファームウェアのバックアップ無効化

オプションで、アップデートのインストール時に現在の実行中のファームウェアのバックアップコピーを無効にすることができます。これは、フォールバックメカニズムが誤ったファームウェアのインストールからターゲットを保護していないことを意味しますが、ブートローダーから更新パーティションに書き込むことができない場合には役立つ場合があります。関連するコンパイル時間オプションはです

DISABLE_BACKUP=1

2.3.7 「ライトワンス」フラッシュメモリの回避策の有効化

一部のマイクロコントローラーでは、内部フラッシュメモリは、セクター全体が消去された後、セクターに追加の書き込み (ゼロを追加) を許可しません。wolfBoot は、両方のパーティションの最後にある「フラグ」フィールドにゼロを追加するメカニズムに依存して、フェイルセーフスワップメカニズムを提供します。

「ライトワンス」内部フラッシュの回避策を有効にするには、

NVM_FLASH_WRITEONCE=1

警告このオプションが有効になっている場合、フェールセーフスワップは保証されません。つまり、マイクロコントローラーを SWAP 操作中に安全に電源を入れたり再起動したりすることはできません。

2.3.8 バージョンロールバックの許可

wolfBoot は、現在のものよりも小さいバージョン番号があるファームウェアの更新を許可しません。ダウングレードを許可するには、ALLOW_DOWNGRADE=1 でコンパイルします。

警告：このオプションは、更新前にバージョンチェックを無効にし、システムを潜在的な強制ダウングレード攻撃の危険性にさらすことになります。

2.3.9 外部フラッシュメモリのオプションのサポートを有効にします

wolfBoot は MakeFile オプション EXT_FLASH=1 でコンパイルできます。外部フラッシュサポートが有効になっている場合、パーティションを更新およびスワップすることが外部メモリに関連付けられ、読み取り/書き込み/消去アクセスに代替 HAL 機能を使用します。更新またはスワップパーティションを外部メモリに関連付けるには、それぞれ PART_UPDATE_EXT および/または PART_SWAP_EXT を定義します。デフォルトでは、MakeFile は、外部メモリが存在する場合、PART_UPDATE_EXT と PART_SWAP_EXT の両方が定義されていると想定しています。

NO_XIP=1 MakeFile オプションが存在する場合、システムで実行されない場所がないため、PART_BOOT_EXT も想定されています。これは通常、MMU システム (ARM Cortex-A など) の場合です。オペレーティングシステムのイメージは、実行不可能な不揮発性メモリに保存されている位置に依存しない ELF イメージであり、検証後に起動するために RAM でコピーする必要があります。

外部メモリを使用する場合、HAL API を拡張して、カスタムメモリにアクセスするメソッドを定義する必要があります。ext_flash_* API の説明については、HALの章を参照してください。

2.3.9.1 SPI デバイス EXT_FLASH=1 構成パラメーターと組み合わせて、プラットフォーム固有の SPI ドライバーを使用することができます。外部 SPI フラッシュメモリにアクセスします。MakeFile オプション SPI_FLASH=1 で wolfBoot をコンパイルすることにより、外部メモリは追加の SPI レイヤーに直接マッピングされるため、ユーザーは ext_flash_* 関数を定義する必要はありません。

代わりに、SPI 関数を定義する必要があります。SPI ドライバーの例は、hal/spi ディレクトリの複数のプラットフォームで利用できます。

2.3.9.2 隣接システムとの間で UART ブリッジを使用 外部デバイスをマップするために利用できるもう 1 つの代替手段は、隣接システムに UART ブリッジを有効にするです。近隣システムは、wolfBoot プロトコルと互換性のある UART インターフェースを介してサービスを公開する必要があります。

SPI デバイスの場合と同じように、UART_FLASH=1 が使用される場合、ext_flash_* API は wolfBoot によって自動的に定義されます。

詳細については、セクションUART 経由のリモート外部フラッシュメモリサポートを参照してください

2.3.9.3 外部パーティションの暗号化サポート SPI_FLASH、UART_FLASH、またはカスタム外部マッピングと組み合わせて、EXT_FLASH=1 を使用して更新およびスワップパーティションが外部デバイスにマッピングされると、ブートローダからそれらのパーティションにアクセスするときに Chacha20、AES128、または AES256 暗号化を有効にすることができます。更新イメージは、鍵ツールを使用して提供元で事前に暗号化する必要があります。wolfBoot は、一時 Chacha20 対称鍵を使用して更新のコンテンツにアクセスするように指示する必要があります。

このオプション機能の詳細については、暗号化された外部パーティションセクションを参照してください。

2.3.10 RAM からフラッシュアクセスコードの実行

wolfBoot が実行されている同じデバイスに書き込むとき、またはフラッシュ自体の構成を変更するときなどに、一部のプラットフォームでは、Flash Access コードを RAM から実行する必要があります。

ライティング用の内部フラッシュにアクセスするすべてのコードを RAM のセクションに移動するには、コンパイル時間オプション `RAM_CODE=1` を使用します (一部のハードウェア構成では、ブートローダーが書き込みのためにフラッシュにアクセスするために必要です)。

2.3.11 デュアルバンクハードウェアアシストスワッピングの有効化

ターゲットプラットフォームでサポートされる場合、ハードウェアアシストデュアルバンクスワッピングを使用して更新を実行できます。この機能を有効にするには、`DUALBANK_SWAP=1` を使用してください。現在、STM32F76X と F77X のみがこの機能をサポートしています。

2.3.12 ブートパーティションセクターに更新パーティションフラグを保存

デフォルトでは、wolfBoot は、各パーティションの最後にある特定のエリアの単一セクターへの更新手順のステータスを追跡し、パーティション自体に関連付けられたフラグのセットを保存および取得することに専念しています。

場合によっては、更新パーティションに関連するステータスフラグとそのセクターを内部フラッシュに保存すると、ブートパーティションに使用される同じフラグのセットとともに保存することが役立つ場合があります。`FLAGS_HOME=1` MakeFile オプションで wolfBoot をコンパイルすることにより、更新パーティションに関連付けられたフラグがブートパーティション自体に保存されます。

一方では、このオプションはブートパーティションで使用可能なスペースをわずかに削減してファームウェアイメージを保存しますが、すべてのフラグをブートパーティションに保存します。

2.3.13 フラグの反転ロジック

デフォルトでは、ほとんどの NVMS は、消去されたページのコンテンツを `0xFF`(すべて) に設定します。一部のフラッシュメモリモデルは、消去ページに反転したロジックを使用し、消去後にコンテンツを `0x00`(すべてゼロ) に設定します。これらの特別なケースでは、オプション `FLAGS_INVERT=1` を使用して、wolfBoot で使用されるパーティション/セクターフラグのロジックを変更できます。

注：上記の `FLAGS_HOME=1` オプションを使用して、反転ロジックとフラッシュと組み合わせて外部フラッシュ (SPI) を使用している場合は、すべてのフラグを 1 つのパーティションに保存してください。

2.3.14 Mac OS/X の使用

Factory.bin で 0xc3 0xbf(C3BF) が繰り返されている場合、OS は Unicode 文字を使用しています。

`"bootloader" ... 0xFF ... "application"=factory.bin` の間に 0xff パディングを組み立てるための「TR」コマンド。「C」ロケールが必要です。

これを端末に設定します

```
LANG=
LC_COLLATE="C"
LC_CTYPE="C"
LC_MESSAGES="C"
LC_MONETARY="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_ALL=
```

次に、通常の `make` ステップを実行します。

2.3.15 グリッチとフォールトインジェクションに対する軽減策の有効化

安全なブートメカニズムに対する攻撃の 1 つのタイプは、強制電圧またはクロックアノマリー、または近距離での電磁干渉を介して CPU に障害を注入することにより、認証と検証のステップの実行をスキップすることです。

CPU 命令をスキップすることを目的とした特定の攻撃からの追加保護は、ARMOR=1 を使用して有効にできます。この機能は現在、ARM Cortex-M ターゲットでのみ利用可能です。

3 ターゲット

この章では、サポートされているターゲットの構成について説明します。

3.1 サポートされているターゲット

- Cortex A53 /Raspberry PI 3
- CypressPSOC-6
- NXP LPC54XXX
- NXP IMX-RT
- NXP kinetis
- NXP T2080 PPC
- sifive hifive1 risc-v
- STM32F4
- STM32L4
- STM32F7
- STM32G0
- STM32H7
- STM32L5
- STM32L0
- STM32WB55
- Ti Hercules TMS570LC435
- Xilinx Zynq Ultrascale
- QEMU X86_64 UEFI

3.2 STM32F4

STM32-F407 での 512KB パーティションの例

`test-app` で提供されるファームウェアの例は、アドレス 0x20000 から始まるプライマリパーティションから起動するように構成されています。フラッシュレイアウトは、`target.h` の次の構成を使用してデフォルトの例で提供されます。

```
#define WOLFBOOT_SECTOR_SIZE               0x20000
#define WOLFBOOT_PARTITION_SIZE            0x20000
#define WOLFBOOT_PARTITION_BOOT_ADDRESS    0x20000
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS 0x40000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS    0x60000
```

これにより、次のパーティション構成が得られます。

この構成は、可能なレイアウトの 1 つを示しており、フラッシュメモリの物理セクターの先頭にスロットが配置されています。

このターゲットのすべての実行可能なファームウェアイメージのエントリポイントは、最初のフラッシュパーティションの先頭から 256 バイト先の、`0x20100`, です。これは、パーティションの先頭にあるファームウェアイメージヘッダーの存在によるものです。

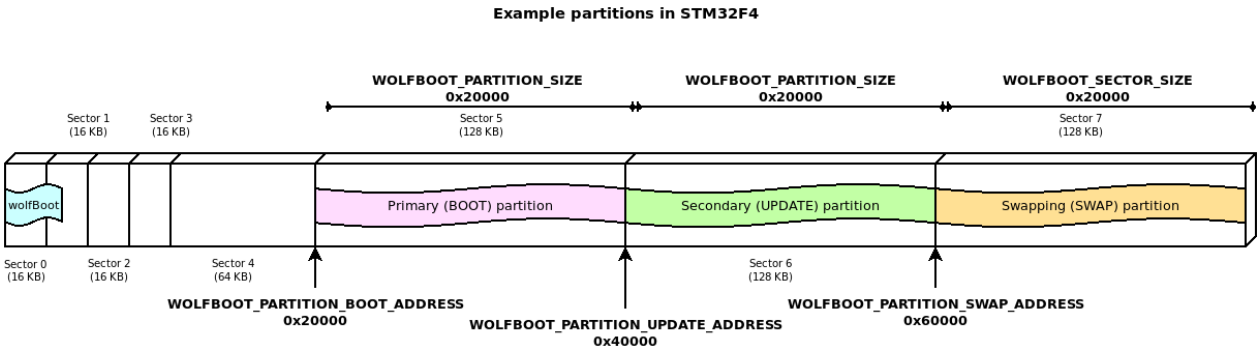

Figure 1: example partitions

この特定のケースでは、フラッシュジオメトリのため、2 つのイメージ間の適切なセクタースワッピングを考慮するには、スワップスペースが 128kb という大きさでなければなりません。

他のシステムでは、複数の小さなフラッシュブロックが使用される場合、スワップスペースは 512 バイトで済む場合があります。

フラッシュおよびアプリケーション内プログラミング (IAP) のジオメトリの詳細については、各ターゲットデバイスのメーカーマニュアルに記載されています。

3.2.1 STM32F4 プログラミング

```
st-flash write factory.bin 0x08000000
```

3.2.2 STM32F4 デバッグ

1. GDB サーバーを開始します

OpenOCD：`openocd --file ./config/openocd/openocd_stm32f4.cfg`ORST-LINK：`st-util -p 3333`

2. GDB クライアントを開始します

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf 0x20100
mon reset init
b main
c
```

3.3 STM32L4

STM32L4 での例 1MB パーティション

- セクターサイズ：4KB
- wolfBoot パーティションサイズ：40KB
- アプリケーションパーティションサイズ：488KB

```
#define WOLFBOOT_SECTOR_SIZE                0x1000   /* 4 KB */
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x0800A000
#define WOLFBOOT_PARTITION_SIZE             0x7A000  /* 488 KB */
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x08084000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x080FE000
```

3.4 STM32L5

3.4.1 シナリオ 1：TrustZone が有効なケース

3.4.1.1 サンプルプログラムの内容 サンプルプログラムの実装では、セキュアアプリケーションから非セキュアアプリケーションに切り替える方法を示しています。内部フラッシュと内部 SRAM メモリにシステムをを分割できるので最初の半分にセキュアアプリケーションを配置し、残り半分に非セキュアアプリケーションを配置しています。

3.4.1.2 ハードウェアおよびソフトウェア環境

- このサンプルプログラムは、セキュリティを有効にして STM32L562QEIXQ デバイスで実行されます (TZEN=1)。
- このサンプルプログラムは、STMicroelectronics STM32L562E-DK(MB1373) でテストされています。
- ユーザーオプションバイト要件 (STM32CubeProgrammer tool を使用 - 手順については以下を参照してください)

```
TZEN=1                                  System with TrustZone-M enabled
DBANK=1                                 Dual bank mode
SECWM1_PSTRT=0x0  SECWM1_PEND=0x7F  All 128 pages of internal Flash Bank1 set
    as secure
SECWM2_PSTRT=0x1  SECWM2_PEND=0x0   No page of internal Flash Bank2 set as
    secure, hence Bank2 non-secure
```

- 注：STM32CubeProgrammer v2.3.0 が必要です (v2.4.0 には STM32L5 の既知のバグがあります)

3.4.1.3 使い方

1. `cp ./config/examples/stm32l5.config .config`
2. `make TZEN=1`
3. 上記で報告されたオプションバイト構成を備えたボードを準備します

```
STM32_Programmer_CLI -c port=swd mode=hotplug -ob TZEN=1 DBANK=1
STM32_Programmer_CLI -c port=swd mode=hotplug -ob SECWM1_PSTRT=0x0 SECWM1_PEND
    =0x7F SECWM2_PSTRT=0x1 SECWM2_PEND=0x0
```

4. `wolfBoot.bin` をフラッシュの 0x0C000000 に配置

```
STM32_Programmer_CLI -c port=swd -d ./wolfboot.bin 0x0C000000
```

5. `.\test-app\image_v1_signed.bin` をフラッシュ 0x08040000 に配置

```
STM32_Programmer_CLI -c port=swd -d ./test-app/image_v1_signed.bin 0x08040000
```

6. 赤色 LED LD9 が点灯します
7. 注：STM32_Programmer_CLI デフォルトのロケーション

- Windows：`C:\Program Files\STMicroelectronics\STM32Cube\STM32CubeProgrammer\bin\STM32_Pr`
- Linux：`/usr/local/STMicroelectronics/STM32Cube/STM32CubeProgrammer/bin/STM32_Programmer_`
- Mac OS/X：`/Applications/STMicroelectronics/STM32Cube/STM32CubeProgrammer/STM32CubeProgra`

3.4.2 シナリオ 2：TrustZone が無効のケース

3.4.2.1 サンプルプログラムの内容 実装では、TrustZone が無効になっている Dual_Bank モードで STM32L5xx を使用する方法を示しています。Dual_Bank オプションは、TrustZone が無効になっている場合にのみこのターゲットで使用できます (Tzen=0)。

フラッシュメモリは、2 つの異なるバンクにセグメント化されています。

- バンク 0：(0x08000000)
- バンク 1：(0x08040000)

Bank 0 にはアドレス 0x08000000 にブートローダーが含まれ、アドレス 0x08040000 にアプリケーションが含まれています。有効なイメージがバンク 1 の同じオフセットで利用可能な場合、2 つの有効なイメージ間で起動するために候補として 0x08048000 が選択されます。

サンプルプログラムのコンフィギュレーションファイルは`/config/examples/stm32l5-nonsecure-dualbank.config` を使います。

イメージ `./test-app/image.bin` を Flash`0x08000000` に配置します。

```
STM32_Programmer_CLI -c port=swd -d ./test-app/image.bin 0x08000000
```

または以下を使用して各パーティションをプログラムします。

1. `wolfboot.bin` をフラッシュ 0x08000000 に配置します

```
STM32_Programmer_CLI -c port=swd -d ./wolfboot.bin
```

2. イメージ./test-app/image_v1_signed.bin をフラッシュ 0x08008000 に配置

```
STM32_Programmer_CLI -c port=swd -d ./test-app/image_v1_signed.bin 0x08008000
```

Red LD9 は、成功した boot() を示すことになります

3.4.3 デバッグ

`make DEBUG=1` を使用してファームウェアをリロードします。

- STM32Cube IDEv.1.3.0 が必要です
- 以下のコマンドでデバッガーを経由して実行します。

Linux:

```
ST-LINK_gdbserver -d -cp /opt/st/stm32cubeide_1.3.0/plugins/com.st.stm32cube.
   ide.mcu.externaltools.cubeprogrammer.linux64_1.3.0.202002181050/tools/bin -
   e -r 1 -p 3333`
```

Mac OS/X：

```
sudo ln -s /Application-
 ↪  s/STM32CubeIDE.app/Contents/Eclipse/plugins/com.st.stm32cube.ide.mcu.externaltools.stli
 ↪  gdb-
 ↪  server.macos64_1.6.0.202101291314/tools/bin/native/mac_x64/libSTLinkUSBDriver.dylib
 ↪  /usr/local/lib/libSTLinkUSBDriver.dylib
/Applications/STM32CubeIDE.app/Contents/Eclipse/plugins/com.st.stm32cube.ide.mcu.externalto
 ↪  gdb-server.macos64_1.6.0.202101291314/tools/bin/ST-LINK_gdbserver -d -cp
 ↪  ./Contents/Eclipse/plugins/-
 ↪  com.st.stm32cube.ide.mcu.externaltools.cubeprogrammer.macos64_1.6.0.202101291314/tools/
 ↪  -e -r 1 -p 3333
```

- ARM-NONE-EABI-GDB と接続します

wolfBoot には、構成する.gdbinit があります

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf
mon reset init
```

3.5 STM32U5

3.5.1 シナリオ 1：TrustZone が有効のケース

3.5.1.1 サンプルプログラムの内容 サンプルプログラムの実装では、セキュアアプリケーションから非セキュアアプリケーションに切り替える方法を示しています。内部フラッシュと内部 SRAM メモリにシステムをを分割できるので最初の半分にセキュアアプリケーションを配置し、残り半分に非セキュアアプリケーションを配置しています。

3.5.1.2 ハードウェアとソフトウエア環境

- サンプルプログラムはセキュリティ機能を有効にした（TZEN=1) STM32U585AII6Q 上で動作します。
- サンプルプログラムは STMicroelectoromics B-U585I-IOT02A(MB1551) でテスト済みです。

```
TZEN = 1                              System with TrustZone-M enabled
DBANK = 1                             Dual bank mode
SECWM1_PSTRT=0x0  SECWM1_PEND=0x7F  All 128 pages of internal Flash Bank1 set
   as secure
SECWM2_PSTRT=0x1  SECWM2_PEND=0x0   No page of internal Flash Bank2 set as
   secure, hence Bank2 non-secure
```

- 注意: STM32CubeProgrammer V2.8.0 以降が必要です

3.5.1.3 使用方法

1. コンフィギュレーションファイルをコピーします `cp ./config/examples/stm32u5.config .config`
2. make します `make TZEN=1`
3. 上記コンフィギュレーションを適用したボードを用意します `STM32_Programmer_CLI -c port=swd mode=hotplug -ob TZEN=1 DBANK=1 STM32_Programmer_CLI -c port=swd mode=hotplug -ob SECWM1_PSTRT=0x0 SECWM1_PEND=0x7F SECWM2_PSTRT=0x1 SECWM2_PEND=0x0`
4. wolfBoot.bin を 0x0c000000 に書き込みます `STM32_Programmer_CLI -c port=swd -d ./wolfboot.bin 0x0C000000`
5. イメージを 0x08040000 に書き込みます `STM32_Programmer_CLI -c port=swd -d./test-app/image_v1_signed.bin 0x08100000`
6. 赤色 LED9 が点灯します

STM32_Programme_CLI の存在位置 - Windows: `C:\Program Files\STMicroelectronics\STM32Cube\STM32Cube`
- Linux: `/usr/local/STMicroelectronics/STM32Cube/STM32CubeProgrammer/bin/STM32_Programmer_CLI`
- Mac OS/X: `/Applications/STMicroelectronics/STM32Cube/STM32CubeProgrammer/STM32CubeProgrammer`

3.5.2 シナリオ 2：TrustZone が無効のケース

3.5.2.1 サンプルプログラムの内容 実装では、TrustZone が無効になっている Dual_Bank モードで STM32U5xx を使用する方法を示しています。Dual_Bank オプションは、TrustZone が無効になっている場合にのみこのターゲットで使用できます (Tzen=0)。

フラッシュメモリは、2 つの異なるバンクにセグメント化されています。

- バンク 0：(0x08000000)
- バンク 1：(0x08100000)

Bank 0 にはアドレス 0x08000000 にブートローダーが含まれ、アドレス 0x08100000 にアプリケーションが含まれています。有効なイメージがバンク 1 の同じオフセットで利用可能な場合、2 つの有効なイメージ間で起動するために候補として 0x08108000 が選択されます。

サンプルプログラムのコンフィギュレーションファイルは config/examples/stm32u5-nonsecure-dualbank.config を使います。

1. イメージ./test-app/image.bin をフラッシュ 0x08000000 に配置します

```
STM32_Programmer_CLI -c port=swd -d ./test-app/image.elf 0x08000000
```

あるいは各パーティションを以下のよにプログラムします - イメージ wolfboot.bin をフラッシュ 0x08000000 に配置

```
STM32_Programmer_CLI -c port=swd -d ./wolfboot.elf
```

- イメージ image_v1_signed.bin をフラッシュ 0x08008000 に配置

```
STM32_Programmer_CLI -c port=swd -d ./test-app/image_v1_signed.bin 0x08008000
```

2. 赤色 LED LD9 が点灯してブートの成功を示します。

3.5.2.2 デバッグ

以下のコマンドでファームウェアをリロードします。

```
make DEBUG=1
```

STM32CubeIDE v.1.7.0 が必要です。デバッガーは各 OS で以下のように起動します：

Linux:

```
ST-LINK_gdbserver -d -cp /opt/st/stm32cubeide_1.3.0/plugins/com.st.stm32cube.
    ide.mcu.externaltools.cubeprogrammer.linux64_1.3.0.202002181050/tools/bin -
    e -r 1 -p 3333
```

Max OS/X:

```
sudo ln -s /Applications/STM32CubeIDE.app/Contents/Eclipse/plugins/com.st.
    stm32cube.ide.mcu.externaltools.stlink-gdb-server.macos64_1
    .6.0.202101291314/tools/bin/native/mac_x64/libSTLinkUSBDriver.dylib /usr/
    local/lib/libSTLinkUSBDriver.dylib

/Applications/STM32CubeIDE.app/Contents/Eclipse/plugins/com.st.stm32cube.ide.
    mcu.externaltools.stlink-gdb-server.macos64_1.6.0.202101291314/tools/bin/ST
    -LINK_gdbserver -d -cp ./Contents/Eclipse/plugins/com.st.stm32cube.ide.mcu.
    externaltools.cubeprogrammer.macos64_1.6.0.202101291314/tools/bin -e -r 1 -
    p 3333
```

Windows：

```
ST-LINK_gdbserver -d -cp C:\ST\STM32CubeIDE_1.7.0\STM32CubeIDE\plugins\com.st.
    stm32cube.ide.mcu.externaltools.cubeprogrammer.win32_2.0.0.202105311346\
    tools\bin -e -r 1 -p 3333
```

arm-none-eabi-gdb に接続します

wolfBoot は　.gdbinit ファイルに以下の内容を含んでいます

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf
mon reset init
```

3.6 STM32L0

STM32-L073 で 192KB パーティションの例

このデバイスは、単一のフラッシュページ (それぞれ 256B) を消去できます。

ただし、スワップにロジックセクターサイズ 4KB を使用して、スワップパーティションに書き込みの量を制限することを選択します。

この例 `target.h` で提案されたジオメトリは、wolfBoot に 32KB を使用し、それぞれ 64KB の 2 節を使用しているため、最大 8KB の Swap に使用する余地があります (ここでは 4K が使用されています)。

```
#define WOLFBOOT_SECTOR_SIZE                0x1000   /* 4 KB */
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x8000
#define WOLFBOOT_PARTITION_SIZE             0x10000 /* 64 KB */
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x18000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x28000
```

3.6.1 STM32L0 ビルド

```
make TARGET=stm32l0
```

を使用してビルドします。オプション `CORTEX_M0` が、このターゲットに対して自動的に選択されます。

3.7 STM32G0

STM32G0x0x0/STM32G0x1 をサポートします。

STM32-G070 での例 128KB パーティション：

- セクターサイズ：2KB
- wolfBoot パーティションサイズ：32KB
- アプリケーションパーティションサイズ：44KB

```
#define WOLFBOOT_SECTOR_SIZE                0x800   /* 2 KB */
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x08008000
#define WOLFBOOT_PARTITION_SIZE             0xB000  /* 44 KB */
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x08013000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x0801E000
```

3.7.1 STM32G0 のビルド

リファレンスコンフィギュレーションとして`/config/examples/stm32g0.config` を参照してください。このコンフィギュレーションファイルを wolfBoot Root に

```
cp ./config/examples/stm32g0.config .config
```

でコピーしてください。その後、make コマンドを使用してビルドします。

これのターゲットは `stm32g0`：`make TARGET=stm32g0` です。オプション `CORTEX_M0` は、このターゲットに対して自動的に選択されます。オプション `NVM_FLASH_WRITEONCE=1` は、このターゲットで必須です。

このターゲットは、FLASH_CR:SEC_PROT および FLASH_SECT:SEC_SIZE レジスタを使用して、ブートローダー領域の安全なメモリ保護もサポートします。これは、0x8000000 ベースアドレスからアクセスをブロックする 2KB ページの数です。

```
STM32_Programmer_CLI -c port=swd mode=hotplug -ob SEC_SIZE=0x10
```

RAMFUNCTION のサポート (SEC_PROT に必要) の場合は、RAM_CODE=1 を指定してください。

コンパイルは以下が必要です：

```
make TARGET=stm32g0 NVM_FLASH_WRITEONCE=1
```

3.7.2 STM32G0 のデバッグ

ビルド生成物は wolfboot.bin と test-app/image_v1_signed.bin を併せた factory.bin の一つだけとなります。このイメージはフラッシュの 0x08000000 に配置する必要があります。STM32CubeProgrammer を使ったコマンドラインは:

```
STM32_Programmer_CLI -c port=swd -d factory.bin 0x08000000
```

となります。

```
make DEBUG=1
```

を使用してファームウェアを再ビルドします。

GDB をポート 3333 で起動します:

```
ST-LINK_gdbserver -d -e -r 1 -p 3333
 or
st-util -p 3333
```

wolfBoot は、GDB 構成のための.gdbinit があります。

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf 0x08008100
mon reset init
```

3.8 STM32WB55

Nucleo-68 ボードでの分割の例：

- セクターサイズ：4KB
- wolfBoot パーティションサイズ：32KB
- アプリケーションパーティションサイズ：128KB

```
#define WOLFBOOT_SECTOR_SIZE                0x1000   /* 4 KB */
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x8000
#define WOLFBOOT_PARTITION_SIZE             0x20000 /* 128 KB */
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x28000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x48000
```

3.8.1 STM32WB55 ビルド

TARGET=stm32wb を指定して make します。

IAP ドライバーは､各消去操作の後に Multiple をサポートしていないため､オプション NVM_FLASH_WRITEONCE=1 はこのターゲットで必須です。

ビルド：

```
make TARGET=stm32wb NVM_FLASH_WRITEONCE=1
```

3.8.2 STM32WB55 を OpenOCD で使う

```
openocd --file ./config/openocd/openocd_stm32wbx.cfg

telnet localhost 4444
reset halt
flash write_image unlock erase factory.bin 0x08000000
flash verify_bank 0 factory.bin
reset
```

3.8.3 STM32WB55 を ST-Linkde で使う

```
git clone https://github.com/stlink-org/stlink.git
cd stlink
cmake .
make
sudo make install

st-flash write factory.bin 0x08000000
# Start GDB server
st-util -p 3333
```

3.8.4 STM32WB55 デバッグ

`make DEBUG=1` を使用してファームウェアをリロードします。

wolfBoot は、GDB 構成のための.gdbinit があります。

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf 0x08008100
mon reset init
```

3.9 SiFive HiFive1 RISC-V

3.9.1 機能

- E31 RISC-V 320MHz 32 ビットプロセッサ
- オンボード 16kb スクラッチパッド RAM
- 外部 4MB QSPI フラッシュ

3.9.2 デフォルトのリンカー設定

- フラッシュ：アドレス 0x20000000、サイズ 0x6a120(424KB)
- RAM：アドレス 0x80000000、サイズ 0x4000(16KB)

3.9.3 ストックブートローダー

アドレスの開始：0x20000000 は 64KB です。「ダブルタップ」リセット機能を提供して、HALT ブートを使用し、デバッガーが再プログラミングのために接続できるようにします。リセットボタンを押すと、緑色に点灯します、そこで再びリセットボタンを押すと、ボードは赤い点滅を始めます。

3.9.4 アプリケーションコード

アドレスの開始：0x20010000

3.9.5 wolfBoot 構成

デフォルトの wolfBoot 構成により、セカンドステージブートローダーが追加され、ストックは「ダブルタップ」ブートローダーを回復のためのフォールバックとして残します。制作の実装は、これを `target.h` のパーティションアドレスとパーティションアドレスに置き換える必要があるため、`0x10000` だけ少なくなります。

Freedom SDK の場所を設定するには、`FREEDOM_E_SDK=~/src/freedom-e-sdk` を使用します。

wolfBoot をテストするために必要な変更は次のとおりです。

1. MakeFile の引数：

- ARCH=RISCV
- TARGET= hifive1

```
make ARCH=RISCV TARGET=hifive1 RAM_CODE=1 clean
make ARCH=RISCV TARGET=hifive1 RAM_CODE=1
```

riscv64-unknown-elf-クロスコンパイラを使用する場合は、make に CROSS_COMPILE=riscv64-unknown-elf-を追加するか、次のように arch.mk'を変更できます。

```
ifeq ($(ARCH),RISCV)
-   CROSS_COMPILE:=riscv32-unknown-elf-
+   CROSS_COMPILE:=riscv64-unknown-elf-
```

2. `include/target.h`

ブートローダーサイズ：0x10000(64KB) アプリケーションサイズ 0x40000(256KB) スワップセクターサイズ：0x1000(4KB)

```
#define WOLFBOOT_SECTOR_SIZE                0x1000
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x20020000
#define WOLFBOOT_PARTITION_SIZE             0x40000
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x20060000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x200A0000
```

3.9.6 ビルドオプション

- ED25519 の代わりに ECC を使用するには、引数 `SIGN=ECC256` を作成します
- JLink を使用するためのヘックスとして wolfboot を出力するには、引数 `wolfboot.hex` を指定します

3.9.7 ロード

JLink でロードする：

```
JLinkExe -device FE310 -if JTAG -speed 4000 -jtagconf -1,-1 -autoconnect 1
loadbin factory.bin 0x20010000
rnh
```

3.9.8 デバッグ

Jlink でのデバッグ：

1 つの端末：

```
JLinkGDBServer -device FE310 -port 3333
```

別の端末で：

```
riscv64-unknown-elf-gdb wolfboot.elf -ex "set remotetimeout 240" -ex "target
    extended-remote localhost:3333"
add-symbol-file test-app/image.elf 0x20020100
```

3.10 STM32F7

STM32-F76x および F77x は、デュアルバンクハードウェアアシストスワッピング機能を提供します。フラッシュジオメトリを事前に定義する必要があり、wolfBoot をコンパイルして、HardWareAssisted Bank-Swapping を使用して更新を実行できます。

STM32-F769 での 2MB パーティションの例：

- デュアルバンク構成
  - バンク A：0x08000000〜0x080fffffff(1MB)
  - バンク B：0x08100000〜0x081fffffff(1MB)
- wolfBoot は再起動後にバンク A から実行されます (アドレス：0x08000000)
- ブートパーティション @ バンク A + 0x20000=0x08020000
- Partition@Bank B + 0x20000=0x08120000 を更新します
- アプリケーションエントリポイント：0x08020100

```
#define WOLFBOOT_SECTOR_SIZE                0x20000
#define WOLFBOOT_PARTITION_SIZE             0x40000
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x08020000
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS 0x08120000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x0    /* Unused, swap is hw-assisted
 ↪  */
```

3.10.1 ビルドオプション

STM32F76x/77x でデュアルバンクハードウェアアシストスワップ機能を有効にするには、TheDUALBANK_SWAP=1 コンパイル時オプションを使用します。一部のコードでは、イメージのスワッピング中に RAM で実行する必要があるため、この場合にはコンパイル時オプション RAMCODE=1 も必要です。

デュアルバンク STM32F7 ビルドは、以下を使用してビルドできます。

```
make TARGET=stm32f7 DUALBANK_SWAP=1 RAM_CODE=1
```

3.10.2 ファームウェアのロード

シングルバンク (1x2MB) とデュアルバンク (2 x 1MB) モードマッピングを切り替えるには、このSTM32F7-DUALBANK-TOOLを使用することができます。OpenOCD を開始する前に、フラッシュモードをデュアルバンクに切り替えます (例：デュアルバンクツールを使用して make dualbank を介して)。

Flashing/Debugging の OpenOCD 構成は、作業ディレクトリの openocd.cfg にコピーできます。

```
source [find interface/stlink.cfg]
source [find board/stm32f7discovery.cfg]
$_TARGETNAME configure -event reset-init {
    mmw 0xe0042004 0x7 0x0
}
init
reset
halt
```

OpenOCD は、コマンドラインから順番にターミナルスクリプトを実行するために、バックグラウンドで実行し、端子接続を監視し、端子接続を監視するために実行できます。

OpenOCD が実行されている場合、ローカル TCP ポート 4444 を使用して、インタラクティブ端末プロンプトにアクセスできます。`telnet localhost 4444`

次の OpenOCD コマンドを使用して、wolfBoot の初期イメージと Bank 0 でフラッシュするためにロードされたテストアプリケーションがロードされます。

```
flash write_image unlock erase wolfboot.bin 0x08000000
flash verify_bank 0 wolfboot.bin
flash write_image unlock erase test-app/image_v1_signed.bin 0x08020000
flash verify_bank 0 test-app/image_v1_signed.bin 0x20000
reset
resume 0x0000001
```

新しいバージョン (2) と同じアプリケーションイメージに署名するには、以下のコマンドで Python スクリプト `sign.py` を使用してください。

```
tools/keytools/sign.py test-app/image.bin ed25519.der 2
```

OpenOCD から、更新されたイメージ (バージョン 2) を 2 番目のバンクにフラッシュ書き込みできます。

```
flash write_image unlock erase test-app/image_v2_signed.bin 0x08120000
flash verify_bank 0 test-app/image_v1_signed.bin 0x20000
```

再起動すると、wolfBoot は最高の候補者 (この場合はバージョン 2) を選択し、イメージを認証します。受け入れられた候補のイメージが Bank B に存在する場合 (この場合など)、wolfBoot はブート前に 1 つのバンク交換を実行します。

この場合、バンクのスワップ操作は即時であり、スワップイメージは必要ありません。フォールバックメカニズムは、他のバンクの 2 番目の選択肢 (古いファームウェア) に依存する可能性があります。

3.10.3 STM32F7 デバッグ

OpenOCD でのデバッグ：

前のセクションの OpenOCD 構成を使用して、OpenOCD を実行します。

別のコンソールから、GDB を使用して接続します。例えば：

```
arm-none-eabi-gdb
(gdb) target remote:3333
```

3.11 STM32H7

STM32H7 フラッシュジオメトリを事前に定義する必要があります。

“make config” を使用して.config ファイルを生成するか、テンプレートコンフィギュレーションファイルをコピーします。

```
cp ./config/examples/stm32h7.config .config
```

STM32-H753 での例 2MB パーティション：

```
WOLFBOOT_SECTOR_SIZE?=0x20000
WOLFBOOT_PARTITION_SIZE?=0xD0000
WOLFBOOT_PARTITION_BOOT_ADDRESS?=0x8020000
WOLFBOOT_PARTITION_UPDATE_ADDRESS?=0x80F0000
WOLFBOOT_PARTITION_SWAP_ADDRESS?=0x81C0000
```

3.11.1 ビルドオプション

STM32H7 ビルドは、以下を使用してビルドできます。

```
make TARGET=stm32h7 SIGN=ECC256
```

3.11.2 STM32H7 のプログラミング

ST-Link Flash Tool を使った書き込み:

```
st-flash write factory.bin 0x08000000
```

あるいは

```
st-flash write wolfboot.bin 0x08000000
st-flash write test-app/image_v1_signed.bin 0x08020000
```

3.11.3 STM32H7 のテスト

新しいバージョン (2) と同じアプリケーションイメージに署名するには、以下のコマンドで Python スクリプト `sign.py` を使用してください。

Python:

```
tools/keytools/sign.py test-app/image.bin ed25519.der 2
```

C Tool:

```
tools/keytools/sign --ecc256 --sha256 test-app/image.bin
   wolfboot_signing_private_key.der 2
```

更新イメージバージョン２を書き込みます：

```
st-flash write test-app/image_v2_signed.bin 0x08120000
```

リブート時には wolfBoot が最も適したアプリケーションイメージ（この場合にはバージョン２）を選択して認証します。認証に成功すると、イメージはバンク B に残り wolfBoot がブート前にスワップを実行します。

3.11.4 STM32H7 デバッグ

1. GDB サーバーを起動

```
st-util -p 3333
```

2. GDB クライアントを wolfBoot のルートフォルダから起動

```
arm-none-eabi-gdb
add-symbol-file test-app/image.elf 0x08020000
mon reset init
b main
c
```

3.12 NXP LPC54xxx

3.12.1 ビルドオプション

LPC54XXX ビルドは、コンパイル時に CPU タイプと MCUXPresso SDK パスを指定して実行します。

次の構成は、LPC54606J512BD208 に対してテストされています。

```
make TARGET=lpc SIGN=ECC256 MCUXPRESSO?=/path/to/LPC54606J512/SDK
    MCUXPRESSO_CPU?=LPC54606J512BD208 \
    MCUXPRESSO_DRIVERS?=$(MCUXPRESSO)/devices/LPC54606 \
    MCUXPRESSO_CMSIS?=$(MCUXPRESSO)/CMSIS
```

3.12.2 ファームウェアのロード

Jlink のロード (例：LPC54606J512)

```
JLinkExe -device LPC606J512 -if SWD -speed 4000
erase
loadbin factory.bin 0
r
h
```

3.12.3 Jlink でデバッグ

```
JLinkGDBServer -device LPC606J512 -if SWD -speed 4000 -port 3333
```

次に、別のコンソールから：

```
arm-none-eabi-gdb wolfboot.elf -ex "target remote localhost:3333"
(gdb) add-symbol-file test-app/image.elf 0x0000a100
```

3.13 Cortex-a53/raspberry pi 3(実験)

Ubuntu20 上で `https://github.com/raspberrypi/linux` を使用してテストしました

前提条件として以下が必要です。

```
sudo apt install gcc-aarch64-linux-gnu qemu-system-aarch64
```

3.13.1 カーネルをコンパイル

- Raspberry-Pi Linux カーネルを入手：

```
git clone https://github.com/raspberrypi/linux linux-rpi -b rpi-4.19.y --depth
    =1
```

- カーネルイメージをビルド：

```
export wolfboot_dir=`pwd`
cd linux-rpi
patch -p1 < $wolfboot_dir/tools/wolfboot-rpi-devicetree.diff
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcmrpi3_defconfig
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
```

- イメージと.dtb を wolfboot ディレクトリにコピーします

```
cp Image arch/arm64/boot/dts/broadcom/bcm2710-rpi-3-b.dtb $wolfboot_dir
cd $wolfboot_dir
```

3.13.2 qmenu-System-aarch64 でのテスト

- サンプル構成 (RSA4096、SHA3) を使用して wolfBoot をビルド

```
cp config/examples/raspi3.config .config
make clean
make wolfboot.bin CROSS_COMPILE=aarch64-linux-gnu-
```

- Linux カーネルイメージに署名

```
make keytools
./tools/keytools/sign --rsa4096 --sha3 Image wolfboot_signing_private_key.der
    1
```

- イメージ作成

```
tools/bin-assemble/bin-assemble wolfboot_linux_raspi.bin 0x0 wolfboot.bin 0
    xc0000 Image_v1_signed.bin
dd if=bcm2710-rpi-3-b.dtb of=wolfboot_linux_raspi.bin bs=1 seek=128K conv=
    notrunc
```

- qmenu を使用したテストブート

```
qemu-system-aarch64 -M raspi3 -m 512 -serial stdio -kernel
    wolfboot_linux_raspi.bin -append "terminal=ttyS0 rootwait" -dtb ./bcm2710-
    rpi-3-b.dtb -cpu cortex-a53
```

3.14 Xilinx Zynq Ultrascale

Xilinx UltraScale+ ZCU102(Aarch64)

構成オプションをビルドする (.config)：

```
TARGET=zynq
ARCH=AARCH64
SIGN=RSA4096
HASH=SHA3
```

3.14.1 QNX

```
cd ~
source qnx700/qnxsdp-env.sh
cd wolfBoot
cp ./config/examples/zynqmp.config .config
make clean
make CROSS_COMPILE=aarch64-unknown-nto-qnx7.0.0-
```

3.14.1.1 デバッグ

```
qemu-system-aarch64 -M raspi3 -kernel /path/to/wolfboot/factory.bin -serial
    stdio -gdb tcp::3333 -S
```

3.14.1.2 署名 `tools/keytools/sign.py --rsa4096 --sha3 /srv/linux-rpi4/vmlinux.bin rsa4096.der 1`

3.15 CypressPSOC-6

Cypress PSOC 62S2 は、デュアルコア Cortex-M4 & Cortex-M0+ MCU です。Secure Boot プロセスは、M0+.wolfBoot によって管理され、アプリケーションの確認とファームウェアの更新を管理するために、Second Stage Flash ブートローダとしてコンパイルできます。

3.15.1 ビルド

次の構成は、PSOC 62S2 Wi-Fi BT Pioneer Kit(CY8CKIT-052S2-43012) を使用してテストされています。

3.15.1.1 ターゲット固有の要件 wolfBoot は、次のコンポーネントを使用して、PSoC の周辺機能にアクセスします：

- Cypress コアライブラリ
- PSoC 6 周辺ドライバーライブラリ
- CY8CKIT-062S2-43012 BSP

Cypress は、フラッシュとデバッグをプログラミングするためのカスタマイズ済み OpenOCDを提供します。

3.15.2 クロック設定

wolfBoot は、PLL1 を 100 MHz で実行するように構成し、その周波数で `CLK_FAST`、`CLK_PERI`、および `CLK_SLOW` を駆動しています。

3.15.2.1 ビルドコンフィグレーション 次のコンフィグレーションは、PSoC CY8CKIT-62S2-43012 でテストされています:

```
make TARGET=psoc6 \
    NVM_FLASH_WRITEONCE=1 \
    CYPRESS_PDL=./lib/psoc6pdl \
    CYPRESS_TARGET_LIB=./lib/TARGET_CY8CKIT-062S2-43012 \
    CYPRESS_CORE_LIB=./lib/core-lib \
    WOLFBOOT_SECTOR_SIZE=4096
```

注：コンフィギュレーションファイル `.config` は`/config/examples/cypsoc6.config` にあります。

ハードウェアアクセラレーションは、PSoC6 Crypto HW サポートを使用してデフォルトで有効になります。

ハードウェアアクセラレーションを無効にしてコンパイルするには、wolfBoot のコンフィグレーションで次のオプションを使用してください。

```
PSOC6_CRYPTO=0
```

3.15.2.2 OpenOCD インストール

カスタマイズ済み OpenOCD をコンパイルしてインストールします。

openocd を実行しているときに次のコンフィギュレーションファイルを使用して、PSoC6 ボードに接続します:

```
### openocd.cfg for PSoC-62S2
source [find interface/kitprog3.cfg]
transport select swd
adapter speed 1000
source [find target/psoc6_2m.cfg]
init
reset init
```

3.15.3 ファームウェアのロード

`factory.bin` を OpenOCD でデバイスにアップロードするには、デバイスを接続し、OpenOCD を前のセクションから構成とともに実行し、`telnet localhost 4444` を使用して TCP ポート 4444 で実行されているローカル OpenOCD サーバーに接続します。

Telnet コンソールから、次を入力：

```
program factory.bin 0x10000000
```

転送が終了したら、OpenOCD を閉じるか、デバッグセッションを開始できます。

3.15.4 デバッグ

OpenOCD でのデバッグ：

OpenOCD を実行するには、以前のセクションの OpenOCD 構成を使用します。

別のコンソールから、GDB を使用して接続します、例えば：

```
arm-none-eabi-gdb
(gdb) target remote:3333
```

ボードをリセットして、M0+ フラッシュブートローダーの位置 (wolfBoot reset handler) から開始するには、以下のモニターコマンドシーケンスを使用します。

```
(gdb) mon init
(gdb) mon reset init
(gdb) mon psoc6 reset_halt
```

3.16 NXP IMX-RT

NXP RT1060/1062 および RT1050

NXP IMX-RT1060 は、SHA256 アクセラレータである DCP コプロセッサを備えた ARM Cortex-M7 です。このターゲットのサンプルコンフィグレーションファイルは`/config/examples/imx-rt1060.config`で提供されます。

3.16.1 wolfBoot のビルド

wolfBoot がこのプラットフォーム上のデバイスドライバーにアクセスするためには MCUXPresso SDK が必要です。パッケージは、ターゲットを選択し、コンポーネントのデフォルトの選択を維持することにより、MCUXpresso SDK Builderから取得できます。

- RT1060 を使用するには EVKB-IMXRT1060 を使用します。config/examples/imx-rt1060.config のコンフィグレーション例を参照してください。
- RT1050 を使用するには EVKB-IMXRT1050 を使用します。config/examples/imx-rt1050.config のコンフィグレーション例を参照してください。

wolfBoot MCUXPRESSO コンフィグレーション変数を SDK パッケージが抽出されるパスに設定し、make を実行して wolfBoot を通常ビルドします。

iMX-RT1060/iMX-RT1050 の wolfBoot サポートは、MCUXPresso SDK バージョン 2.11.1 を使用してテストされています。

DCP サポート (SHA256 のハードウェアアクセラレーション) は、コンフィグレーションファイルで PKA=1 を使用して有効にできます。ファームウェアは、factory.bin をデバイスに関連付けられた仮想 USB ドライブにコピーすることにより、ターゲットに直接アップロードできます。

3.17 NXP Kinetis

暗号ハードウェアアクセラレーションで K64 と K82 をサポートします。

3.17.1 ビルドオプション

サンプルプログラムのコンフィギュレーションファイルは、/config/examples/kinetis-k82f.config を参照してください。

ターゲットは kinetis です。LTC PKA をサポートする場合には PKA= で指定します。

MCUXpresso 構成については、MCUXPRESSO、MCUXPRESSO_CPU、MCUXPRESSO_DRIVERS、MCUXPRESSO_CMSIS を設定します。

3.17.2 K82 のパーティション分割の例

```
WOLFBOOT_PARTITION_SIZE?=0x7A000
WOLFBOOT_SECTOR_SIZE?=0x1000
WOLFBOOT_PARTITION_BOOT_ADDRESS?=0xA000
WOLFBOOT_PARTITION_UPDATE_ADDRESS?=0x84000
WOLFBOOT_PARTITION_SWAP_ADDRESS?=0xff000
```

3.18 NXP T2080 PPC

T2080 は PPC e6500 ベースのプロセッサです。

このターゲットのコンフィギュレーションファイルは、/config/examples/t2080.config で提供されます。

3.18.1 wolfBoot のビルド

wolfBoot は、gcc powerpc ツールでビルドできます。たとえば、aptinstall gcc-powerpc-linux-gnu。これで make が正しいツールを使ってビルドできます。

3.19 TI Hercules TMS570LC435

サンプルプログラムのコンフィギュレーションファイルについては、/config/examples/ti-tms570lc435.config を参照してください。

3.20 QEMU X86-64 UEFI

UEFI bios を備えた X86-64 ビットマシンは、EFI アプリケーションとして wolfBoot を実行できます。

3.20.1 前提要件:

- Qemu-system-x86_64
- gnu-efi
- Open Virtual Machine firmware bios images (OVMF)

Debian のようなシステムでは、次のようにパッケージをインストールするだけで十分です。

```
# for wolfBoot and others
apt install git make gcc

# for test scripts
apt install sudo dosfstools curl
apt install qemu qemu-system-x86 ovmf gnu-efi

# for buildroot
apt install file bzip2 g++ wget cpio unzip rsync bc
```

3.20.2 コンフィグレーション

サンプルプログラムのコンフィギュレーションファイルは `config/examples/x86_64_efi.config` で提供されます

3.20.3 qemu を使ってのビルドと実行

EFI 環境で実行するためのブートローダーと初期化スクリプト `startup.nsh` は、ループバック FAT パーティションに保存されます。

スクリプト `tools/efi/prepare_uefi_partition.sh` は、新しい空の FAT ループバックパーティションを作成し、`startup.nsh` を追加します。

埋め込まれた rootfs パーティションを備えたカーネルを作成して、スクリプト `tools/efi/compile_efi_linux.sh` を介してイメージに追加できます。スクリプトは、実際にターゲットシステムの 2 つのインスタンスを追加します：`kernel.img` および `update.img` は、両方とも認証に署名し、それぞれバージョン 1 および 2 でタグ付けされています。

`make` でコンパイルすると、`wolfboot.efi` にブートローダーイメージが生成されます。

スクリプト `tools/efi/run_efi.sh` は、`wolfboot.efi` をブートローダーループバックパーティションに追加し、QEMU でシステムを実行します。両方のカーネルイメージが存在していて有効な場合、wolfBoot はより高いバージョン番号を使用してイメージを選択します。そのため、`update.img` はバージョン 2 でタグ付けされたときにステージングされます。

シーケンスを以下にまとめます。

```
cp config/examples/x86_64_efi.config .config
tools/efi/prepare_efi_partition.sh
make
tools/efi/compile_efi_linux.sh
tools/efi/run_efi.sh

EFI v2.70 (EDK II, 0x00010000)
[700/1832]
```

```
Mapping table
      FS0: Alias(s):F0a:;BLK0:
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
     BLK1: Alias(s):
               PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
Press ESC in 1 seconds to skip startup.nsh or any other key to continue.
Starting wolfBoot EFI...
Image base: 0xE3C6000
Opening file: kernel.img, size: 6658272
Opening file: update.img, size: 6658272
Active Part 1
Firmware Valid
Booting at 0D630000
Staging kernel at address D630100, size: 6658016
```

Ctrol-C あるいは root としてログインし、poweroff で qemu を終了します。

3.21 Nordic nRF52840

Contiki と RIOT-OS 向けの Nordic nRF5280 サンプルプログラムをwolfBoot-example repoで提供しています。

nRF52 用サンプルプログラム: * RIOT-OS: https://github.com/wolfSSL/wolfBoot-examples/tree/master/riotOS-nrf52840dk-ble * Contiki-OS: https://github.com/wolfSSL/wolfBoot-examples/tree/master/contiki-nrf52

nRF52 向けフラッシュメモリレイアウト例：

- 0x000000 - 0x01efff : Reserved for Nordic SoftDevice binary
- 0x01f000 - 0x02efff : Bootloader partition for wolfBoot
- 0x02f000 - 0x056fff : Active (boot) partition
- 0x057000 - 0x057fff : Unused
- 0x058000 - 0x07ffff : Upgrade partition

```
#define WOLFBOOT_SECTOR_SIZE               4096
#define WOLFBOOT_PARTITION_SIZE            0x28000

#define WOLFBOOT_PARTITION_BOOT_ADDRESS    0x2f000
#define WOLFBOOT_PARTITION_SWAP_ADDRESS    0x57000
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS 0x58000
```

3.22 シミュレートターゲット

内部あるいは外部フラッシュメモリを模したファイルを使うシミュレートターゲットを生成することができます。ビルドすると wolfBoot.elf ファイルを生成します。また、別のファームウエアイメージとして実行可能な ELF ファイルを提供することもできます。サンプルプログラム test-app/app_sim.c は libwolfboot.c と対話するために引数を使用し、機能テストを自動化します。このサンプルプログラムのコンフィギュレーションファイルは config/examples/sim.config を参照してください。

test-app/sim.c を使ったファームウェア更新プログラムは以下をコマンドで生成します：

```
cp ./config/examples/sim.config .config
make

# create the file internal_flash.dd with firmware v1 on the boot partition and
# firmware v2 on the update partition
```

```
make test-sim-internal-flash-with-update
# it should print 1
./wolfboot.elf success get_version
# trigger an update
./wolfboot.elf update_trigger
# it should print 2
./wolfboot.elf success get_version
# it should print 2
./wolfboot.elf success get_version
```

4 ハードウェア抽象化レイヤー

ターゲットマイクロコントローラーで wolfBoot を実行するには、HAL の実装を提供する必要があります。

HAL の目的は、ブートローダーからの書き込み/消去操作と、アプリケーションライブラリを介してファームウェアのアップグレードを開始するアプリケーションを許可し、ブート中に MCU がフルスピードで実行されるようにすることです (署名の検証を最適化するため)。

各プラットフォームのハードウェア固有の呼び出しの実装は、hal ディレクトリの単一の C ファイルにグループ化されます。

ディレクトリには、サポートされている各 MCU のプラットフォーム固有のリンカースクリプトも含まれており、同じ名前と .ld 拡張機能があります。これは、特定のハードウェアにブートローダーのファームウェアをリンクし、フラッシュ境界と RAM 境界に必要なすべてのシンボルをエクスポートするために使用されます。

4.1 サポートされているプラットフォーム

現在のバージョンでは、次のプラットフォームがサポートされています。- STM32F4、STM32L5、STM32L0、STM32F7、STM32H7、STM32G0 - NRF52 - ATMEL SAMR21 - TI CC26X2 - KINETIS - SiFive HiFive1 RISC-V

4.2 API

ハードウェア抽象化レイヤー (HAL) は、サポートされているターゲットごとに 6 つの関数呼び出しで構成されています。

```
void hal_init(void)
```

この関数は、実行の最初にブートローダーによって呼び出されます。理想的には、提供された実装は、ターゲットマイクロコントローラーのクロック設定を構成し、暗号化プリミティブがファームウェアイメージを確認するために必要な時間を短縮するために必要な速度で実行されるようにします。

```
void hal_flash_unlock(void)
```

ターゲットのフラッシュメモリの IAP インターフェースがそれを必要とする場合、この関数はすべての書き込みおよび消去操作の前に呼び出され、フラッシュへの書き込みアクセスを解除します。一部のターゲットでは、この関数が空になる場合があります。

```
int hal_flash_write(uint32_t address, const uint8_t *data, int len)
```

この関数は、ターゲットの IAP インターフェースを使用して、フラッシュ書き込み関数の実装を提供します。address はフラッシュ領域の先頭からのオフセットであり、data は IAP インターフェースを使用してフラッシュに保存するペイロード、len はペイロードのサイズです。hal_flash_write は、成功すると 0 を返す必要があります。

```
void hal_flash_lock(void)
```

フラッシュメモリの IAP インターフェースにロック/ロック解除が必要な場合、この関数は、書き込みアクセスを除外してフラッシュ書き込み保護を復元します。この関数は、すべての書き込みおよび消去操作の最後にブートローダーによって呼び出されます。

```
int hal_flash_erase(uint32_t address, int len)
```

ブートローダーによって呼び出されて、フラッシュメモリの一部を消去して、後続のブートを許可します。ターゲットマイクロコントローラーの特定の IAP インターフェースを介して、消去操作を実行する必要があります。address ブートローダーが消去したいエリアの開始をマークし、len は消去するエリアのサイズを指定します。この関数は、フラッシュセクターのジオメトリを考慮し、その間のすべてのセクターを消去する必要があります。


```
void hal_prepare_boot(void)
```

この関数は、次の段階でファームウェアをチェーンでロードする前に、非常に遅い段階でブートローダーによって呼び出されます。これを使用して、マイクロコントローラーの状態が元の設定に復元されるように、クロック設定に行われたすべての変更を戻すことができます。

4.2.1 外部フラッシュメモリのオプションのサポート

wolfBoot は makefile コマンドへのオプション EXT_FLASH=1 でコンパイルできます。外部フラッシュサポートが有効になっている場合、パーティションを更新およびスワップすることが外部メモリに関連付けられ、読み取り/書き込み/消去アクセスに代替 HAL 機能を使用します。更新またはスワップパーティションを外部メモリに関連付けるには、それぞれ PART_UPDATE_EXT および/または PART_SWAP_EXT を定義します。

以下の関数は、外部メモリにアクセスするために使用され、EXT_FLASH がオンになっている場合に定義する必要があります。

```
int  ext_flash_write(uintptr_t address, const uint8_t *data, int len)
```

この関数は、外部メモリの特定のインターフェースを使用して、フラッシュ書き込み関数の実装を提供します。address は、デバイス内のアドレス指定可能なスペースの先頭からのオフセット、data は保存するペイロード、len はペイロードのサイズです。ext_flash_write は、成功すると 0 を返す必要があります。

```
int  ext_flash_read(uintptr_t address, uint8_t *data, int len)
```

この関数は、ドライバーの特定のインターフェースを使用して、外部メモリの間接的な読み取りを提供します。address は、デバイス内のアドレス指定可能なスペースの先頭からのオフセットであり、data はコールの成功にペイロードが保存されるポインターであり、len はペイロードに許容される最大サイズです。ext_flash_read は、成功すると 0 を返す必要があります。

```
int  ext_flash_erase(uintptr_t address, int len)
```

ブートローダによって呼び出され、外部メモリの一部を消去します。消去操作は、ターゲットドライバーの特定のインターフェース (SPI フラッシュなど) を介して実行する必要があります。address は、デバイスに対するエリアの開始をマークし、ブートローダーが消去したいと考え、len は消去するエリアのサイズを指定します。この関数は、セクターのジオメトリを考慮し、その間のすべてのセクターを消去する必要があります。

```
void ext_flash_lock(void)
```

外部フラッシュメモリのインターフェースにロック/ロック解除が必要な場合、この関数を使用してフラッシュ書き込み保護を復元するか、書き込みアクセスを除外することができます。この関数は、外部デバイスのすべての書き込みおよび消去操作の最後にブートローダーによって呼び出されます。

```
void ext_flash_unlock(void)
```

外部メモリの IAP インターフェースがそれを必要とする場合、この関数は、すべての書き込みおよび消去操作の前に呼び出され、デバイスへの書き込みアクセスを解除します。一部のドライバーでは、この機能が空になる場合があります。

5 フラッシュパーティション

5.1 フラッシュメモリパーティション

wolfBoot を統合するには、フラッシュメモリのジオメトリを考慮して、フラッシュを別々の領域 (パーティション) に分割する必要があります。

イメージの境界は、新しいファームウェアイメージを保存する前にすべてのフラッシュセクターを消去し、2 つのパーティションのコンテンツを一度に 1 つずつスワップするため、**かならず** 物理セクターにアラインする必要があります。

このため、ターゲットシステムのパーティションを進める前に、次の側面を考慮する必要があります。

- ブートパーティションと更新パーティションのサイズは同じで、実行システムを保持できること大きさでなければなりません
- スワップパーティションは、ブートパーティションと更新パーティションの両方で最大のセクターと同じ大きさでなければなりません。

ターゲットのフラッシュメモリは、次の領域に分割されます。

- ブートローダパーティション、フラッシュメモリの先頭アドレスに位置し一般的に非常に小 (16-32KB)
- アドレス `WOLFBOOT_PARTITION_BOOT_ADDRESS` から始まるプライマリスロット (ブートパーティション)
- セカンダリスロット (更新パーティション) アドレス `WOLFBOOT_PARTITION_UPDATE_ADDRESS`
- 両方のパーティションは同じサイズを共有します。AS `WOLFBOOT_PARTITION_SIZE`-アドレス `WOLFBOOT_PARTITION_SWAP_ADDRESS` から始まるスペース (スワップパーティション)
- スワップスペースサイズは `WOLFBOOT_SECTOR_SIZE` として定義され、ブート/更新パーティションで使用される最大のセクターと同じ大きさでなければなりません。

`include/target.h` のオフセットとサイズの値を設定することにより、特定の使用のために適切なパーティション構成を設定する必要があります。

5.1.1 ブートローダーパーティション

このパーティションは通常非常に小さく、ブートローダーコードとデータのみが含まれています。工場のイメージの作成中に事前に許可されたパブリック鍵は、ファームウェアイメージの一部として自動的に保存されます。

5.1.2 ブートパーティション

これは、ファームウェアイメージをチェーンロードして実行できる唯一のパーティションです。ファームウェアイメージは、そのエントリポイントがアドレス `WOLFBOOT_PARTITION_BOOT_ADDRESS + 256` にあるようにリンクする必要があります。

5.1.3 更新パーティション

実行中のファームウェアは、安全なチャネルを介して新しいファームウェアイメージを転送し、セカンダリスロットに保存する責任があります。更新が開始された場合、ブートローダーは次の再起動時にブートパーティションのファームウェアを交換またはスワップします。

5.2 パーティションステータスとセクターフラグ

パーティションは、現在使用されているファームウェアイメージ (ブート) を保存するか、(更新)(更新) の準備ができているために使用されます。各パーティションのファームウェアのステータスを追跡するために、各パーティションスペースの端に 1 バイト状態フィールドが保存されます。このバイトは、パーティションが初めて消去およびアクセスされるときに初期化されます。

可能な状態は次のとおりです。

- STATE_NEW(0xff)：ブートのためにイメージがステージングされることはなく、更新用にトリガーされました。イメージが存在する場合、フラグはアクティブではありません。
- STATE_UPDATING(0x70)：更新パーティションでのみ有効です。イメージは更新用にマークされており、ブートの現在のイメージを置き換える必要があります。
- STATE_TESTING(0x10)：ブートパーティションでのみ有効です。イメージは更新されたばかりで、ブートを完成させませんでした。再起動後に存在する場合、正しく検証されているにもかかわらず、更新されたイメージが起動に失敗したことを意味します。この特定の状況は、ロールバックを引き起こします。
- STATE_SUCCESS(0x00)：ブートパーティションでのみ有効です。ブートに保存されたイメージは、少なくとも一度は正常にステージングされており、更新が完了しました。

州のバイトから始めて後方に成長しているブートローダーは、更新パーティションの最後にセクターごとに 4 ビットを使用して、各セクターの状態を追跡します。更新が開始されるたびに、ファームウェアはアップデートから一度に 1 つのセクターを起動するために転送され、元のファームウェアのバックアップをブートから更新まで保存します。各フラッシュアクセス操作は、セクターフラグエリアのセクターのフラグの異なる値に対応するため、操作が中断された場合、再起動時に再開できます。

5.3 フラッシュパーティションのコンテンツの概要

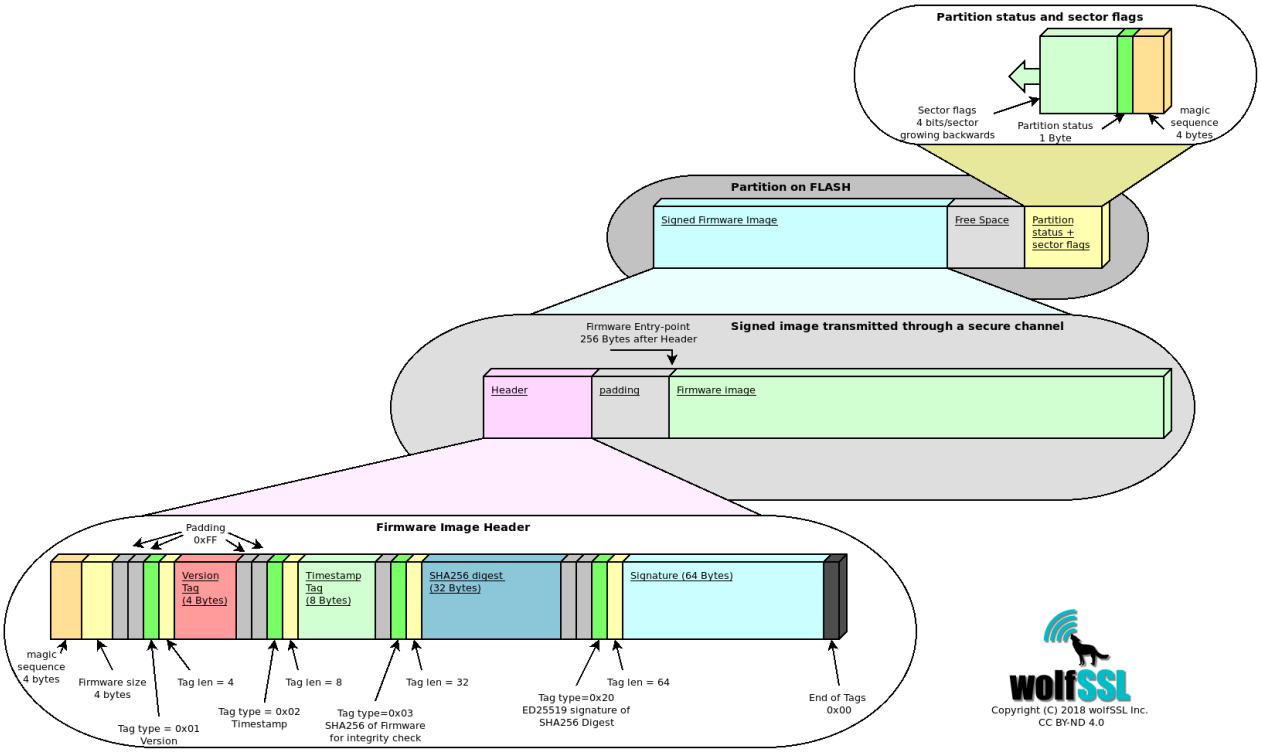

Figure 2: wolfBoot partition

6 wolfBoot の機能

6.1 署名

6.1.1 wolfBoot 鍵ツールのインストール

Python、wolfCrypt-Py モジュール、ファームウェアの署名と鍵生成のための wolfBoot をセットアップするための手順を説明します。

注：利用可能な鍵ツールの純粋な C バージョンもあります。以下のC 言語鍵ツールを参照してください。

6.1.2 Python3 のインストール

1. 最新の Python 3.x をダウンロードして、インストーラーを実行します。
2. Python 3.x をパスに追加するというボックスにチェックをいれてください。

6.1.3 wolfcrypt のインストール

```
git clone https://github.com/wolfSSL/wolfssl.git
cd wolfssl
./configure --enable-keygen --enable-rsa --enable-ecc --enable-ed25519 --
    enable-ed448 --enable-des3 CFLAGS="-DWOLFSSL_PUBLIC_MP"
make
sudo make install
```

6.1.4 wolfCrypt-py のインストール

```
git clone https://github.com/wolfSSL/wolfcrypt-py.git
cd wolfcrypt-py
sudo USE_LOCAL_WOLFSSL=/usr/local pip3 install .
```

6.1.5 wolfBoot のインストール

```
git clone https://github.com/wolfSSL/wolfBoot.git
cd wolfBoot
git submodule update --init
## Setup configuration (or copy template from ./config/examples)
make config
## Build the wolfBoot binary and sign an example test application
make
```

6.1.6 C 言語-鍵ツール

Keygen ツールのスタンドアロン C バージョンは、`./tools/keytools` に格納されています。

これらは、make を使用して `tools/keytools` に生成されます。または wolfBoot のルートから `make keytools` を使用してビルドすることもできます。

鍵ツールの C バージョンが存在する場合、wolfBoot で使用されます (デフォルトは Python スクリプトです)。

6.1.6.1 Windows Visual Studio `wolfBootSignTool.vcxproj` Visual Studio Project を使用して、Windows で使用する `sign.exe` および `keygen.exe` ツールをビルドできます。

6.1.7 コマンドラインの使用方法

6.1.7.1 Keygen tool

```
./tools/keytools/keygen [--ed25519 | --ed448 | --ecc256 | --rsa2048 | --
    rsa4096 ]  pub_key_file.c
```

keygen は鍵ストアを有効にし、既存あるいは新規に作成した公開鍵を管理するために使われます。２つのオプションがサポートされています：- [-g privkey.der] は新規鍵ペアを生成します。生成した公開鍵は鍵ストアに追加し、秘密鍵は prickey.der ファイルとして出力します。- [-i existing.der] は既存の公開鍵をファイル existing.der ファイルからインポートします。

引数は排他的ではありませんし、複数の鍵を鍵ストアに格納するために一度以上繰り返して指定できます。鍵ストアで使用するアルゴリズムを一つは指定する必要があります（即ち、–ed25519 か–rsa3072)。利用可能なオプションは” 公開鍵署名オプション” を参照してください。

keygen ツールで生成されるファイルは以下のものがあります：- C ファイル　src/keystore.c は生成された C コードでプロビジョニングされる場合は wolfBoot イメージとリンクされます - バイナリーファイル keystore.img は代替ストレージを通じてプロビジョニングされた公開鍵が使われる場合には利用可能です - コマンドラインから “-g” オプションとともに指定された秘密鍵

6.1.7.2 署名ツール sign と sign.py は wolfBoot がサポートする形式のマニフェストヘッダーを生成することで単一のファームウェアイメージを生成します。

```
sign[.py] [OPTIONS] IMAGE.BIN KEY.DER VERSION
```

- IMAGE.BIN: 署名対象のバイナリーファームウェア
- KEY.DER: バイナリーファームウェアに署名を行う秘密鍵で、DER 形式
- VERSION:　署名されたイメージに関連付けられたバージョン
- OPTIONS: なしあるいは以下に示すオプション：
  - –ed25519 ED25519 アルゴリズムを署名に使用する。KEY.DER ファイルはこの鍵フォーマットであることを期待している
  - –ed448 ED448 アルゴリズムを署名に使用する。KEY.DER ファイルはこの鍵フォーマットであることを期待している
  - –ecc256 ecc256 アルゴリズムを署名に使用する。KEY.DER ファイルはこの鍵フォーマットであることを期待している
  - –ecc384 ecc448 アルゴリズムを署名に使用する。KEY.DER ファイルはこの鍵フォーマットであることを期待している
  - –rsa2048 rsa2048 アルゴリズムを署名に使用する。KEY.DER ファイルはこの鍵フォーマットであることを期待している
  - –rsa3072 rsa3072 アルゴリズムを署名に使用する。KEY.DER ファイルはこの鍵フォーマットであることを期待している
  - –rsa4096 rsa4096 アルゴリズムを署名に使用する。KEY.DER ファイルはこの鍵フォーマットであることを期待している
  - –no-sign セキュアブートで署名検証を使用しない。KEY.DER 引数は無視される

6.1.8 鍵生成と管理

KeyStore は wolfBoot によって使用されるメカニズムの呼び名です。ここでは、ファームウエアの更新の署名検証に使われるすべての公開鍵の保管を行っています。

wolfBoot の鍵生成ツールは一つ以上の鍵を生成することができます。make コマンドを最初に使用する際に単一の秘密鍵 wolfboot_signing_private_key.der を生成し keystore モジュールに追加します。この鍵はどのファームウエアの実行あるいは更新に於いて署名するのに使用されるべきです。

加えて、keygen ツールは、KeyStore の異なる表現を持つ追加ファイルを作成します - .c ファイル (src/keystore.c)　wolfboot.elf でキーストアをリンクすることにより、ブートローダー自体の一部として公開鍵を

展開するために使用できる - .bin ファイル (keystore.bin)　カスタム メモリ サポートでホストできるキーストアを含む。

キーストアにアクセスするには、小さなドライバーが必要です (以下のセクション「インターフェース API」を参照)。

デフォルトでは、src/keystore.c の KeyStore オブジェクトは、ビルドにそのシンボルを含めることにより、wolfBoot によってアクセスされます。生成されると、このファイルには、ターゲットシステム上の wolfBoot で使用できる各公開鍵を記述する構造体の配列が含まれます。さらに、公開鍵スロットの詳細とコンテンツにアクセスするために wolfBoot キーストア API に接続する関数が含まれています。

公開鍵は以下の構造体で記述されます：

```
struct keystore_slot {
     uint32_t slot_id;
     uint32_t key_type;
     uint32_t part_id_mask;
     uint32_t pubkey_size;
     uint8_t  pubkey[KEYSTORE_PUBKEY_SIZE];
 };
```

- slot_id は、鍵スロット識別子で、0 から始まります。
- key_type は鍵のアルゴリズムを記述します。AUTH_KEY_ECC256 または AUTH_KEY_RSA3072
- mask は、鍵のアクセス許可を記述します。これは、この鍵を検証に使用できるパーティション ID のビットマップです
- pubkey_size 公開鍵バッファーのサイズ
- pubkey 公開鍵を生の形式で保持する実際のバッファ

起動時に、wolfBoot は署名付きファームウェアイメージに関連付けられた公開鍵を自動的に選択し、検証が実行されているパーティション ID の許可マスクと一致することを確認してから、選択した公開鍵スロットを使用してイメージの署名を認証します。

6.1.8.1 複数鍵の生成

KeyGen は複数の秘密鍵生成のサポートのために複数のファイル名を受け付けます。

- “-g priv.der” は新たに鍵ペアを生成します。秘密鍵は priv.der ファイルに、公開鍵は KeyStore に格納します
- “-i pub.der” は既存の公開鍵を pub.der ファイルからインポートし KeyStorea に格納します

ED25519 鍵を使って KeyStore を作成する例を示します：

```
./tools/keytools/keygen.py --ed25519 -g first.der -g second.der
```

この例は次のファイルを生成します：

- first.der 第 1 の秘密鍵
- second.der 第 2 の秘密鍵
- src/keystore.c 第１，第２の秘密鍵に対応した 2 つの公開鍵を含んだ C KeyStore

keystore.c は以下の様に見えるはずです：

```
#define NUM_PUBKEYS 2
const struct keystore_slot PubKeys[NUM_PUBKEYS] = {

     /* Key associated to private key 'first.der' */
    {
        .slot_id = 0,
        .key_type = AUTH_KEY_ED25519,
        .part_id_mask = KEY_VERIFY_ALL,
```

```
        .pubkey_size = KEYSTORE_PUBKEY_SIZE_ED25519,
        .pubkey = {
            0x21, 0x7B, 0x8E, 0x64, 0x4A, 0xB7, 0xF2, 0x2F,
            0x22, 0x5E, 0x9A, 0xC9, 0x86, 0xDF, 0x42, 0x14,
            0xA0, 0x40, 0x2C, 0x52, 0x32, 0x2C, 0xF8, 0x9C,
            0x6E, 0xB8, 0xC8, 0x74, 0xFA, 0xA5, 0x24, 0x84
        },
    },

     /* Key associated to private key 'second.der' */
    {
        .slot_id = 1,
        .key_type = AUTH_KEY_ED25519,
        .part_id_mask = KEY_VERIFY_ALL,
        .pubkey_size = KEYSTORE_PUBKEY_SIZE_ED25519,
        .pubkey = {
            0x41, 0xC8, 0xB6, 0x6C, 0xB5, 0x4C, 0x8E, 0xA4,
            0xA7, 0x15, 0x40, 0x99, 0x8E, 0x6F, 0xD9, 0xCF,
            0x00, 0xD0, 0x86, 0xB0, 0x0F, 0xF4, 0xA8, 0xAB,
            0xA3, 0x35, 0x40, 0x26, 0xAB, 0xA0, 0x2A, 0xD5
        },
    },


};
```

6.1.8.2 公開鍵とパーミッション デフォルトでは、新しい KeyStore が作成されると、パーミッションマスクが KEY_VERIFY_ALL に設定されます。これは、キーを使用して、任意のパーティション ID を対象とするファームウェアを検証できることを意味します。

単一のキーのアクセス許可を制限するには、part_id_mask 属性の値を変更するだけで十分です。

part_id_mask 値はビットマスクで、各ビットは異なるパーティションを表します。ビット「0」は wolfBoot の自己更新用に予約されていますが、通常、メイン ファームウェア パーティションは ID 1 に関連付けられているため、ビット「1」が設定された鍵が必要です。つまり、–id 3 でパーティションに署名するには、マスクのビット '3' をオンにする必要があります。つまり、(1U « 3) を追加する必要があります。

KEY_VERIFY_ALL のほかに、定義済みのマスク値もここで使用できます。

- KEY_VERIFY_APP_ONLY は、パーティション ID が 1 のメイン アプリケーションのみを検証します
- KEY_VERIFY_SELF_ONLY は、wolfBoot 自己更新の認証にのみ使用できます (id = 0)
- キーの使用を特定のパーティション ID N に制限するために使用できる KEY_VERIFY_ONLY_ID(N) マクロ

6.1.9 ファームウェアへの署名

1. ./rsa2048.der、./rsa4096.der、./ed25519.der、ecc256.der、または./ed448.der にサインするために使用する秘密鍵をロードする
2. 非対称アルゴリズム、ハッシュアルゴリズム、ファイルへのファイル、鍵、バージョンを使用して、署名ツールを実行します。

```
./tools/keytools/sign --rsa2048 --sha256 test-app/image.bin rsa2048.der 1
## OR
python3 ./tools/keytools/sign.py --rsa2048 --sha256 test-app/image.bin rsa2048
    .der 1
```

注：最後の引数は「バージョン」番号です。

6.1.10 外部秘密鍵 (HSM) でファームウェアに署名する

外部鍵ソースを使用してファームウェアに手動で署名するための手順。

```
## 公開鍵ファイルを生成
openssl rsa -inform DER -outform DER -in rsa2048.der -out rsa2048_pub.der -
    pubout
## 署名のためのハッシュを生成
./tools/keytools/sign --rsa2048 --sha-only --sha256 test-app/image.bin
    rsa2048_pub.der 1
## または
python3 ./tools/keytools/sign.py --rsa2048 --sha-only --sha256 test-app/image.
    bin rsa4096_pub.der 1
## ハッシュで署名 (HSMを使用する場合)
openssl rsautl -sign -keyform der -inkey rsa2048.der -in test-app/
    image_v1_digest.bin > test-app/image_v1.sig
## 最終バイナリを生成
./tools/keytools/sign --rsa2048 --sha256 --manual-sign test-app/image.bin
    rsa2048_pub.der 1 test-app/image_v1.sig
## または
python3 ./tools/keytools/sign.py --rsa2048 --sha256 --manual-sign test-app/
    image.bin rsa4096_pub.der 1 test-app/image_v1.sig
## ファクトリーイメージに組み込み
cat wolfboot-align.bin test-app/image_v1_signed.bin > factory.bin
```

6.2 wolfBoot を使用した管理ブート

wolfBoot は、信頼できるプラットフォームモジュール (TPM) を使用してシステムブートプロセスの状態を記録および追跡する方法である、簡略化された管理されたブート実装を提供します。

レコードは、Platform Configuration Register と呼ばれる TPM の特別なレジスタによって改ざん防止されています。次に、ファームウェアアプリケーションである RTOS または RICH OS(Linux) は、TPM の PCR を読み取ることにより、情報のログにアクセスできます。

wolfTPM との統合により、wolfBoot は TPM2.0 チップと対話できます。wolfTPM は、Microsoft Windows と Linux のネイティブサポートを備えており、Standalone または wolfBoot と一緒に使用できます。wolfBoot と wolfTPM の組み合わせにより、開発者は、ブート中および起動後にシステムを保護するための改ざん防止セキュアなストレージを提供します。

6.2.1 コンセプト

通常、システムは安全なブートを使用して、正しいファームウェアとその署名を確認することで起動されることを保証します。その後、この知識はシステムに知られていません。アプリケーションは、システムが良好な既知の状態で始まったかどうかを知りません。時には、この保証がファームウェア自体によって必要です。そのようなメカニズムを提供するために、測定されたブーツの概念が存在します。

管理ブートを使用して、設定やユーザー情報 (ユーザーパーティション) など、すべての起動コンポーネントを確認できます。チェックの結果は、PCR と呼ばれる特別なレジスタに保存されます。このプロセスは PCR 拡張と呼ばれ、TPM 測定と呼ばれます。PCR レジスタは、TPM Power-On でのみリセットできます。

TPM 測定値を使用すると、Windows や Linux などのファームウェアまたはオペレーティングシステム (OS) が、システムを制御する前にロードされたソフトウェアが信頼でき、変更されていないことを知る方法を提供します。

wolfBoot では、メインファームウェアイメージである単一のコンポーネントを測定するために、コンセプトが簡素化されます。ただし、これは、より多くの PCR レジスタを使用することで簡単に拡張できます。

6.2.2 コンフィグレーション

管理ブートを有効にするには、wolfBoot Config に MEASURED_BOOT=1 設定を追加します。

また、管理が保存される PCR(インデックス) を選択する必要があります。

MEASURED_BOOT_PCR_A=[index] 設定を使用して選択が行われます。この設定を wolfboot config に追加し、[index] を 0〜23 の数字に置き換えます。以下に、PCR インデックスを選択するためのガイドラインがあります。

すべての TPM には、最低 24 の PCR レジスタがあります。それらの典型的な使用目的は次のとおりです。

インデックス	典型的な使用目的	推奨する環境
0	信頼および/または BIOS 測定のコアルート	ベアメタル、RTOS
1	プラットフォーム構成データの測定	ベアメタル、RTOS
2-3	オプション ROM コード測定	ベアメタル、RTOS
4-5	マスターブートレコード測定	ベアメタル、RTOS
6	状態移行	ベアメタル、RTOS
7	ベンダー固有の	ベアメタル、RTOS
8-9	パーティション測定	ベアメタル、RTOS
10	ブートマネージャーの測定	ベアメタル、RTOS
11	通常、Microsoft BitLocker で使用されます	ベアメタル、RTOS
12-15	あらゆる用途で利用可能	ベアメタル、RTOS、Linux、Windows
16	デバッグ	テスト目的でのみ使用
17	DRTM	信頼できるブートローダ
18-22	信頼できる OS	信頼できる実行環境 (TEE)
23	アプリケーション	一時的な測定にのみ使用

PCR インデックスを選択するための推奨事項：

- 開発中、テストを目的とした PCR16 を使用することをお勧めします。
- 生産時には、ベアメタルファームウェアまたは RTOS を実行している場合は、DRTM および信頼できる OS(PCR17-23) を除き、ほぼすべての PCR(PCR0-15) を使用できます。
- Linux または Windows を実行している場合、Linux IMA や Microsoft BitLocker などの Linux 内から PCR を使用している可能性のある他のソフトウェアとの競合を回避するために、PCR12-15 を生産対応ファームウェアに選択できます。

開発中の wolfboot .config の一部です。

```
MEASURED_BOOT?=1
MEASURED_PCR_A?=16
```

6.2.2.1 コード wolfBoot は、すぐに使えるソリューションを提供しています。測定されたブートを使用するために、開発者が wolfBoot コードにタッチする必要がありません。コードを確認する場合は、src/image.c、より具体的には measure_boot() 関数を調べます。そこには、wolfTPM へのいくつかの TPM2 ネイティブ API 呼び出しがあります。wolfTPM の詳細については、GitHub リポジトリを確認できます。

6.3 ファームウェアイメージ

6.3.1 ファームウェアエントリポイント

wolfBoot は、メモリ内の特定のエントリポイントからファームウェアイメージをチェーンロードおよび実行できます。これは、埋め込みアプリケーションのリンカースクリプトのフラッシュメモリの原点として指定する必要があります。これは、フラッシュメモリの最初のパーティションに対応します。

複数のファームウェアイメージをこの方法で作成し、2 つの異なるパーティションに保存できます。ブートローダーは、選択したファームウェアを最初の (ブート) パーティションに移動する前に、イメージをチェーンする前に処理します。

イメージヘッダーが存在するため、アプリケーションのエントリポイントには、フラッシュパーティションの開始から 256B の固定追加オフセットがあります。

6.3.2 ファームウェアイメージヘッダー

各 (署名された) ファームウェアイメージには、ファームウェアに関する有用な情報が含まれている固定サイズ **image header** が事前に塗装されています。**image header** は、実際のファームウェアのエントリポイントが 256 バイトのアラインされたアドレスから開始されるフラッシュに保存されることを保証するために、256B に収まるようにパディングされています。これにより、ブートローダーがベクトルテーブルを再配置することができます。

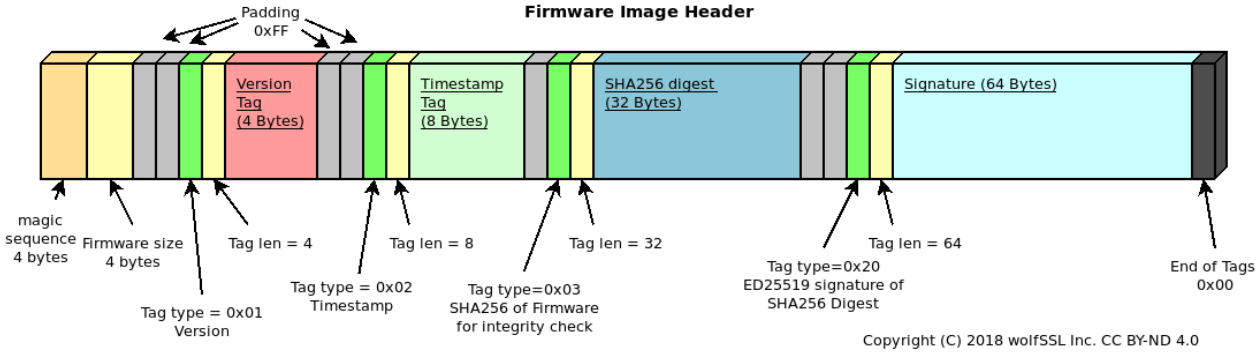

Figure 3: Image header

イメージヘッダーはスロットの先頭に保存され、実際のファームウェアイメージは *256* バイトから始まります

6.3.2.1 イメージヘッダー：タグ

image header には、単一の 4 バイトのマジック番号が追加され、その後にファームウェアイメージ (ヘッダーを除く) を示す 4 バイトフィールドが続きます。ヘッダーのすべての数値は、リトルエンディアン形式で保存されます。

2 つの固定フィールドの後に 1 つ以上のタグが続きます。各タグは次のように構成されています。

-**typer** - タグの **size** を示す **typer** -2 バイトを示す 2 バイト、タイプとサイズのバイト - ***N*** タグコンテンツのバイトを除く

次の例外を除きます。-タイプフィールドの「0xff」は、単純なパディングバイトを示します。「パディング」バイトには **size** フィールドはありません。次のバイトは **typer** として処理する必要があります。各 **typer** には異なる意味があり、ファームウェアに関する情報を統合します。ファームウェアイメージを検証するには、次のタグが必須です。-「バージョン」タグ (タイプ：0x0001、サイズ：4 バイト) イメージに保存されているファームウェアのバージョン番号を示す - 「タイムスタンプ」タグ (タイプ：0x0002、サイズ 8 バイト) ファームウェアの作成のための Unix 秒のタイムスタンプを示す - ファームウェアの整合性チェックに使用される「SHA256 ダイジェスト」タグ (タイプ：0x0003、サイズ：32 バイト) - 「ファームウェア署名」タグ (タイプ：0x0020：0x0020、サイズ：64 バイト) 既知の公開鍵に対してファームウェアで保存されて

いる署名を検証するために使用されます - 「ファームウェアタイプ」タグ (タイプ：0x0030、サイズ：2 バイト) のファームウェアの種類と認証メカニズム使用する。

オプションで、「公開鍵ヒントダイジェスト」タグをヘッダーに送信できます (タイプ：0x10、サイズ：32 バイト)。このタグには、署名ツールで使用される公開鍵の SHA256 ダイジェストが含まれています。ブートローダーは、このフィールドを使用して、複数の鍵が利用可能な場合に正しい公開鍵を見つけることができます。

wolfBoot は、すべての場合において、組み込みのデジタル署名認証メカニズムを使用して検証および認証できないイメージの起動を拒否します。

6.3.2.2 イメージ署名ツール イメージ署名ツールは、コンパイルされたイメージに必要なすべてのタグを使用してヘッダーを生成し、デバイス上のプライマリスロットに保存するか、後でデバイスに送信して安全なチャネルを介してデバイスに送信できる出力ファイルに追加します。アップデート。

6.3.2.3 ファームウェアイメージの保存 ファームウェアイメージは、システム上のパーティションの先頭にフルヘッダーで保存されます。wolfBoot は、更新パーティションに 2 番目のファームウェアイメージを保持しながら、ブートパーティションからイメージのみを起動できます。

別のイメージを起動するには、wolfBoot は 2 つのイメージのコンテンツを交換する必要があります。

ファームウェアイメージの保存方法の詳細については、2 つのパーティション内で、フラッシュパーティションを参照してください。

6.4 ファームウェアの更新

このセクションでは、完全なファームウェア更新手順を文書化し、既存の組み込みアプリケーションのセキュアブートを有効にします。

6.4.1 マイクロコントローラーフラッシュの更新

wolfBoot でファームウェアアップデートを完了する手順は次のとおりです。-正しいエントリポイントでファームウェアをコンパイルします - ファームウェアに署名します - 安全な接続を使用してイメージを転送し、セカンダリファームウェアスロットに保存 - イメージスワップをトリガーします - 再起動してブートローダーはイメージスワップを開始します

いつでも、wolfBoot システムで実行されているアプリケーションまたは OS は、それ自体の更新されたバージョンを受信し、Flash メモリの 2 番目のパーティションに更新されたイメージを保存できます。

アプリケーションまたは OS スレッドは、API をエクスポートして次の再起動時にアップデートをトリガーするlibwolfboot ライブラリにリンクし、一部のヘルパー関数はフラッシュパーティションにアクセスして、ターゲット固有のハルを介して消去/書き込みを行うことができます。

6.4.2 更新手順の説明

wolfBoot は、アプリケーションに提供されているAPIを使用して、更新を開始、確認、またはロールバックする可能性を提供します。

更新パーティションに新しいファームウェアイメージを保存した後、アプリケーションは `wolfBoot_update_trigger()` を呼び出して更新を開始する必要があります。次の再起動時に、wolfBoot は次の手順を実行します：

- 更新パーティションに保存されている新しいファームウェアイメージを検証します
- ブートローダーイメージに保存されている既知の公開鍵に対して添付された署名を確認します
- ブートコンテンツと更新パーティションのコンテンツを交換します
- 新しいファームウェアに状態 `STATE_TESTING` のマークを付けます

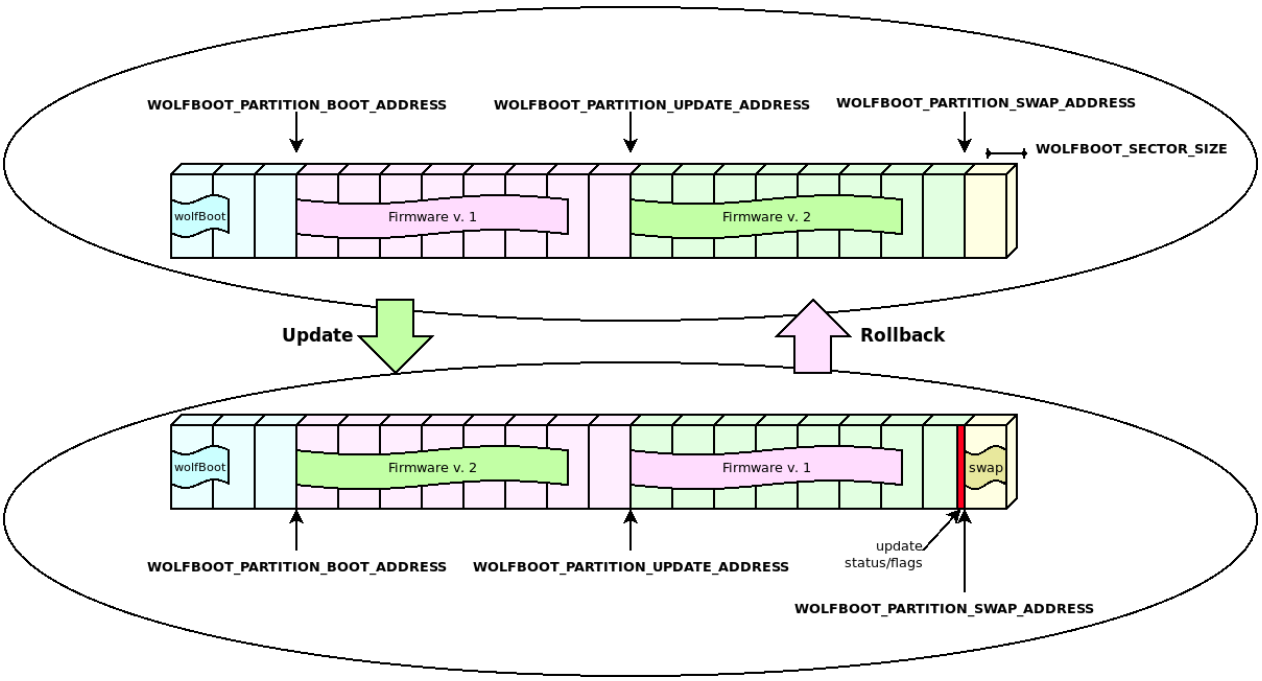

Figure 4: Update and Rollback

- 新しく受信したファームウェアを起動

スワップ操作と再起動中にシステムが中断された場合、wolfBoot は中断したところからピックアップし、更新手順を継続します。

6.4.2.1 ブート成功 ブートが成功すると、システムが再び稼働していることを確認した後、`wolfBoot_success()` を呼び出してブートローダーに通知する必要があります。この操作により、新しいファームウェアの更新が確認されます。

次の再起動前にブートパーティションを STATE_SUCCESS に設定するのに失敗すると、ロールバック操作がトリガーされます。ロールバックは、新しいアップデートをトリガーすることによりブートローダーによって開始されます。今回は、元の (プレ・アップデート前) ファームウェアのバックアップコピーから始まります。これは、以前に発生したスワップのために更新パーティションに保存されています。

6.4.2.2 新しいファームウェアイメージのビルド ファームウェアイメージは位置に依存しており、Flash の**ブート**パーティションの原点からのみ起動できます。この設計上の制約は、選択したファームウェアが常に ** boot ** パーティションに保存されていることを意味し、wolfBoot は更新イメージを事前に検証し、正しいアドレスにコピーする責任があります。

したがって、すべてのファームウェアイメージには、** boot ** パーティションの開始に対応するアドレスにエントリポイントを設定する必要があります。さらに、イメージヘッダーを考慮して 256 バイトのオフセットが必要です。

ファームウェアがコンパイルされてリンクされたら、`sign` ツールを使用して署名する必要があります。このツールは、検証に現在使用されている公開鍵に対応する同じ鍵を使用して、安全な接続を使用してターゲットに転送できる署名付きイメージを生成します。

このツールは、ファームウェアの署名と SHA256 ハッシュを含む、すべての必要なタグをイメージヘッダーに追加します。

6.4.2.3 セルフアップデート RAM_CODE が設定されている場合、wolfBoot は自分自身を更新できます。この手順は、いくつかの重要な違いがありますが、通常のファームウェアアップデートとほぼ同じ動作をします。アップデートのヘッダーは、ブートローダーアップデートとしてマークされています (サインツールに--wolfboot-update を使用)。

署名されている新しい wolfboot イメージは、更新パーティションにロードされ、ファームウェアの更新と同じようにトリガーされます。スワップを実行する代わりに、イメージが検証され、署名検証された後、ブートローダーが消去され、新しいイメージが Flash に書き込まれます。この操作は、中断されると「安全ではありません」。中断すると、デバイスが再起動できなくなります。

wolfBoot は、新しいブートローダーバージョンと更新鍵を展開するために使用できます。

6.4.2.4 インクリメンタルアップデート (別名：「デルタ」更新) wolfBoot は、特定の古いバージョンに基づいて、インクリメンタル更新をサポートしています。サインツールは、ターゲットで現在実行されているバージョンと更新パッケージのバージョンのバイナリの違いのみを含む小さな「パッチ」を作成できます。これにより、ターゲットに転送されるイメージのサイズが縮小され、公開鍵の検証を通じて同じレベルのセキュリティを維持し、繰り返しチェック (パッチと結果のイメージ) による整合性を維持します。

パッチの形式は、Bentley/Mcllroy によって提案されたメカニズムに基づいています。これは、小さなバイナリパッチを生成するのに特に効果的です。これは、更新を転送、認証、インストールするために必要な時間とリソースを最小限に抑えるのに役立ちます。

6.4.2.4.1 どのように動作するのか ファームウェアイメージ全体を転送する代わりに、鍵ツールは、以前にアップロードされたベースバージョンと新しい更新されたイメージの間にバイナリ diff を作成します。

結果のバンドル (Delta Update) には、基本バージョンから始まるファームウェアのバージョン「2」のコンテンツを導き出すための情報が含まれています。バージョン「2」をバージョンに戻すには、新しいバージョンを実行している場合にバージョンに戻ります。

デバイス側では、wolfBoot は、パッチを現在のファームウェアに適用する前に、Delta アップデートの信頼性を認識して検証します。新しいファームウェアは適切に再ビルドされ、(認証された)「デルタアップデート」バンドルの表示に従ってブートパーティションのコンテンツを置き換えます。

6.4.2.4.2 2 ステップ検証 バイナリパッチは、署名されたファームウェアイメージを比較することによって作成されます。wolfBoot は、パッチ後の結果のイメージの整合性と信頼性をチェックすることにより、パッチが正しく適用されることを確認します。

パッチを含むデルタアップデートバンドル自体には、パッチの詳細を説明するマニフェストヘッダーが付いており、通常のフルアップデートバンドルのように署名されています。

これは、wolfBoot が 2 つのレベルの認証を適用することを意味します。デルタバンドルが処理されたときの最初のレベル (アップデートがトリガーされたとき)、2 番目のレベルは、パッチが適用されるか、または逆になって、起動前にファームウェアイメージを検証するために、。

これらの手順は、例で説明されているように、--delta オプションを使用する場合、鍵ツールによって自動的に実行されます。

6.4.2.4.3 更新の確認 アプリケーションの観点から見ると、通常の「完全な」更新ケースから変わるものはありません。アプリケーションは、更新されたバージョンを使用して最初のブーツで wolfBoot_success() を呼び出して、更新が確認されていることを確認する必要があります。

アップデートの成功を確認できないと、wolfBoot が更新中に適用されたパッチを元に戻します。「Delta Update」バンドルには逆パッチも含まれており、更新を戻してファームウェアのベースバージョンを復元できます。

以下の図は、認証手順と両方向の diff/パッチプロセスを示しています (確認のための更新とロールバック)。

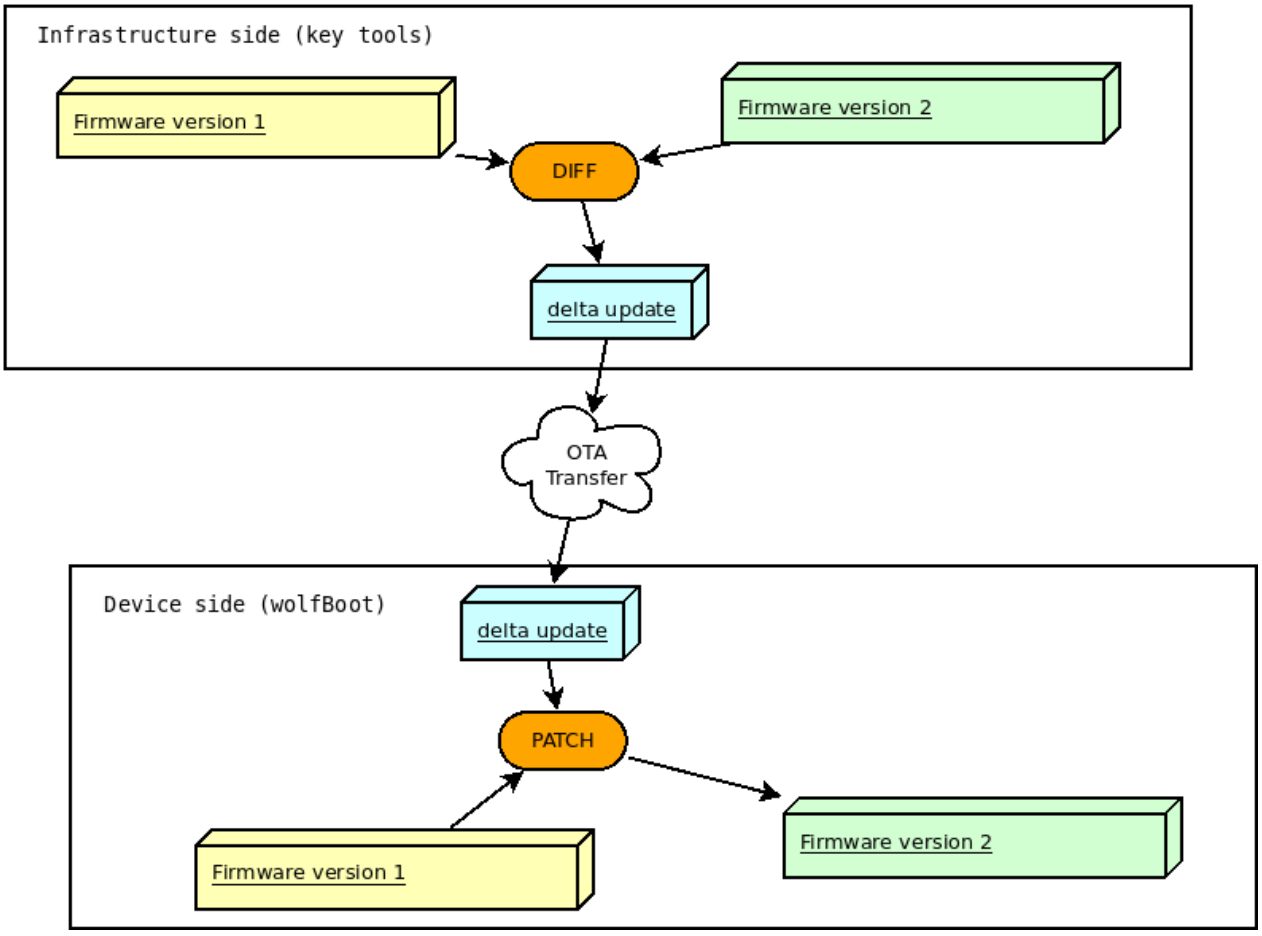

Figure 5: Delta update

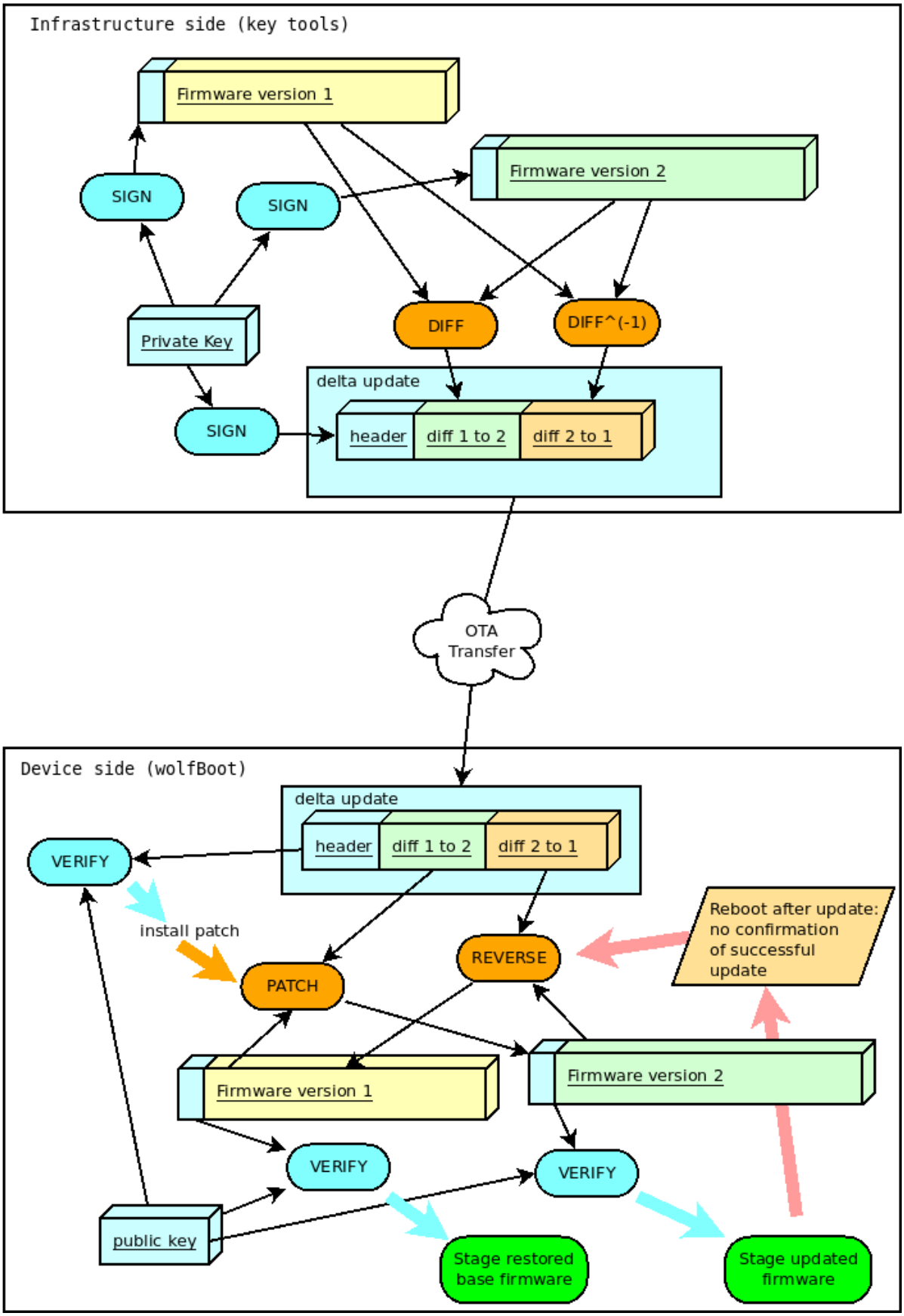

Figure 6: Delta update: details

6.4.2.4.4 インクリメンタル更新：例 要件：wolfBoot は DELTA_UPDATES=1 でコンパイルされています

バージョン「1」は、スタンドアロンのイメージとして、通常どおり署名されています。

```
tools/keytools/sign.py --ecc256 --sha256 test-app/image.bin ecc256.der 1
```

バージョン 1 からバージョン 2 に更新する場合、サインツールを次のように呼び出すことができます。

```
tools/keytools/sign.py --delta test-app/image_v1_signed.bin --ecc256 --sha256
    test-app/image.bin ecc256.der 2
```

通常の出力ファイル image_v2_signed.bin に加えて、符号ツールは、2 つのバイナリファイルに重複領域が含まれている限り、サイズが著しく小さくなる必要がある追加の image_v2_signed_diff.bin を作成します。

これは、最初のパッチが適用された後、バージョン 1 からバージョン 2 を更新するためのパッチを含む署名付きパッケージ、および必要に応じてバージョン 1 にロールバックするデルタアップデートバンドルです。

デルタバンドル image_v2_signed_diff.bin は、完全な更新イメージのようにターゲットの更新パーティションに転送できるようになりました。

次回の再起動では、wolfBoot はインクリメンタルアップデートを認識し、パッチの整合性、信頼性、およびバージョンをチェックします。すべてのチェックが成功した場合、現在のファームウェアイメージにパッチを適用することにより、新しいバージョンがインストールされます。

更新が確認されていない場合、次回の再起動時に wolfBoot は、Delta Update バンドルに含まれる逆パッチを使用して、元のベース image_v1_signed.bin を復元します。

6.5 UART 経由のリモート外部フラッシュメモリサポート

wolfBoot は、近隣システムとの UART 通信を使用して外部パーティションをエミュレートできます。この機能は、外部処理ユニットの支援を受けて更新を保存できる非同期マルチプロセスアーキテクチャで特に役立ちます。

6.5.1 ブートローダセットアップ

この機能をアクティブにするオプションは UART_FLASH=1 です。この構成オプションは、外部フラッシュ API に依存します。つまり、オプション EXT_FLASH=1 はブートローダーをコンパイルするためにも必須です。

ターゲットシステムの HAL は、搭載された UART コントローラーの 1 つを使用してリモートフラッシュのコンテンツにアクセスするためにブートローダーが使用する単純な UART ドライバーを含むように拡張する必要があります。

サポートされているいくつかのプラットフォームの UART ドライバーの例は、hal/uart ディレクトリにあります。

サポートされているターゲットの UARTHAR 拡張機能によって公開された API は、次の機能によって構成されています。

```
int uart_init(uint32_t bitrate, uint8_t data, char parity, uint8_t stop);
int uart_tx(const uint8_t c);
int uart_rx(uint8_t *c);
```

まだ正式にサポートされていない場合、プラットフォームで外部フラッシュメモリサポートを使用する場合は、提供された例に基づいてこれら 3 つの機能を実装することを検討してください。

6.5.2 ホスト側：UART Flash Server

ターゲットの外部パーティションイメージをホストするリモートシステムでは、Flash-Access 固有の呼び出しを提供するために、UART メッセージの上に簡単なプロトコルを実装できます。

GNU/Linux ホストで実行し、ファイルシステム上のローカルファイルを使用して外部パーティションをエミュレートするように設計された UART-Flash-Server デーモンの例は、ツール/uart-flash-serverで入手できます。

6.5.3 外部フラッシュ更新メカニズム

wolfBoot は、外部の更新を扱い、パーティションをローカル SPI フラッシュにマッピングしたときと同じ方法でパーティションを交換します。読み取りおよび書き込み操作は、UART を介してリモートプロシージャコールに翻訳されます。これは、リモートアプリケーションによって解釈され、ホストがのみアクセスできる実際のストレージ要素への読み取りおよび書き込みアクセスを提供できます。

これは、更新が成功した後、以前のファームウェアのコピーがリモートパーティションに保存され、他のすべてのユースケースで利用可能なまったく同じ更新メカニズムを提供することを意味します。唯一の違いは、物理的な保管エリアにアクセスする方法にありますが、より高いレベルのすべてのメカニズムは同じままです。

6.6 暗号化された外部パーティション

wolfBoot は、更新パーティション全体のコンテンツを暗号化する可能性を提供します。この暗号化には、より安全な非揮発性メモリ領域に一時的に保存できる事前共有対称鍵を使用します。

スワップパーティションは同じ鍵を使用して一時的に暗号化されるため、外部フラッシュのダンプでは、ファームウェアアップデートパッケージのコンテンツが表示されません。

6.6.1 根拠

外部パーティションの暗号化は、外部フラッシュインターフェイスのレベルで機能します。

ブートローダーから外部パーティションへのすべての書き込みコールは、追加の暗号化ステップを実行して、外部の不揮発性メモリの実際のコンテンツを非表示にします。

逆に、すべての読み取り操作は、機能が有効になったときに保存されたデータを復号化します。

署名後にファームウェアアップデートを暗号化するための `sign.py` サインツールに追加のオプションが提供されます。これにより、アプリケーションによって外部メモリに保存され、更新を確認して開始するためにブートローダーによって復号化されます。インストール。

6.6.2 一時的な鍵ストレージ

デフォルトでは、wolfBoot は、内部フラッシュ上の一時的な領域に暗号化に使用される事前共有対称鍵を保存します。これにより、一時的な鍵を隠すために読み出しの保護を使用できます。

あるいは、一時的な鍵を別の鍵ストレージに保存するために、より安全なメカニズムを利用できます (たとえば、ハードウェアセキュリティモジュールまたは TPM デバイスを使用)。

一時的な鍵は、アプリケーションによって実行時に設定でき、ブートローダーが次の更新を確認してインストールするために、ブートローダーで 1 回だけ使用できます。鍵は、たとえば、安全な通信を使用して更新プロセス中にバックエンドから受信し、`libwolfboot` API を使用してアプリケーションによって設定され、次のブート時に wolfBoot が使用します。

一時的な鍵を設定することとは別に、更新メカニズムは、wolfBoot を介したファームウェアの更新の配布、アップロード、インストールの場合と同じままです。

6.6.3 libwolfboot ライブラリー API

アプリケーションからブートローダーと通信する API は、この機能が有効になっているときに拡張され、一時的な鍵を設定して次の更新を処理します。

関数

```
int wolfBoot_set_encrypt_key(const uint8_t *key, const uint8_t *nonce);
int wolfBoot_erase_encrypt_key(void);
```

外部パーティションの一時的な暗号化鍵を設定するために、またはそれぞれ以前に設定された鍵を消去するために使用できます。

さらに、libwolfboot を使用して、アプリケーションから wolfBoot HAL を使用して外部フラッシュにアクセスしても、暗号化は使用されません。このようにして、既に Origin で暗号化された受信した更新は、変更されていない外部メモリに保存でき、暗号化された形式で取得できます。再起動する前に転送が成功していることを確認します。

6.6.4 対称暗号アルゴリズム

暗号化は、ENCRYPT=1 を使用して wolfBoot で有効にできます。

外部パーティションでデータを暗号化および復号化するために使用されるデフォルトのアルゴリズムは Chacha20-256 です。AES-128、AES-256 オプションも利用可能で、ENCRYPT_WITH_AES128=1 または ENCRYPT_WITH_AES256=1 を使用して選択できます。

6.6.5 Chacha20-256

Chacha20 が選択されたとき：

-wolfBoot_set_encrypt_key() に提供される key は、正確に 32 バイトの長さでなければなりません。
-nonce 引数は、暗号化と復号化のために IV として使用するには、96 ビット (12 バイト) ランダムに生成されたバッファーでなければなりません。

6.6.5.1 Chacha20-256 での使用例 sign.py ツールは、単一のコマンドでイメージに署名して暗号化できます。暗号化のシークレットは、32B Chacha-256 鍵と 12B NonCE の連結を含むバイナリファイルで提供されます。

提供されている例では、テストアプリケーションは次のパラメーターを使用します。

```
key="0123456789abcdef0123456789abcdef"
nonce="0123456789ab"
```

したがって、テストスクリプトまたはコマンドラインから暗号化のシークレットを次のコマンドで簡単に準備できます。

```
echo -n "0123456789abcdef0123456789abcdef0123456789ab" > enc_key.der
```

sign.py スクリプトを呼び出して、追加の引数--encrypt を使用して署名 + 暗号化されたイメージを作成するように呼び出すことができます。

```
./tools/keytools/sign.py --encrypt enc_key.der test-app/image.bin ecc256.der
    24
```

ファイル test-app/image_v24_signed_and_encrypted.bin を出力すると生成され、ターゲットの外部デバイスに転送できます。

6.6.6 AES-CTR

AES-CTR モードが使用されます。AES が選択された場合：-`wolfBoot_set_encrypt_key()` に提供される key は、16 バイト (AES128) または 32 バイト (AES256) の長さでなければなりません。-nonce 引数は、暗号化と復号化の初期カウンターとして使用される 128 ビット (16Byte) ランダムに生成されたバッファーです。

6.6.6.1 AES-256 での使用例 AES-256 の場合、暗号化のシークレットは、32 バイトの鍵と 16 バイトの IV の連結を含むバイナリファイルで提供されます。

提供されている例では、テストアプリケーションは次のパラメーターを使用します。

```
key="0123456789abcdef0123456789abcdef"
iv="0123456789abcdef"
```

したがって、テストスクリプトまたはコマンドラインから暗号化のシークレットを次のコマンドで簡単に準備できます。

```
echo -n "0123456789abcdef0123456789abcdef0123456789abcdef" > enc_key.der
```

`sign.py` スクリプトを呼び出して、追加の引数`--encrypt` に続いて eCeCret ファイルを使用して、署名 + 暗号化されたイメージを作成するように呼び出すことができます。AES-256 を選択するには、`--aes256` オプションを使用します。

```
./tools/keytools/sign.py --aes256 --encrypt enc_key.der test-app/image.bin
    ecc256.der 24
```

ファイル `test-app/image_v24_signed_and_encrypted.bin` を出力すると生成され、ターゲットの外部デバイスに転送できます。

6.6.7 アプリケーションでの API の使用

イメージを転送する場合、アプリケーションは引き続き Libwolfboot API 関数を使用して、暗号化されたファームウェアを保存できます。アプリケーションから呼び出されると、関数 `ext_flash_write` は、誘引 payload が暗号化されていない保存されます。

更新をトリガーするには、`wolfBoot_update_trigger` を呼び出す前に、`wolfBoot_set_encrypt_key` を呼び出してブートローダーが使用する一時鍵を設定する必要があります。

暗号化された更新トリガーの例は、STM32WB テストアプリケーションソースコード (`../test-app/app_stm32wb.c`) に記載されています。

6.7 ブートローダーとの対話のためのアプリケーションインターフェイス

wolfBoot は、パーティションに保存されているイメージと対話し、更新を明示的に開始し、以前にスケジュールした更新の成功を確認するための小さなインターフェイスを提供します。

6.7.1 libwolfboot とのコンパイルとリンク

wolfBoot との対話を必要とするアプリケーションには、ヘッダーファイルを含める必要があります。

```
#include <wolfboot/wolfboot.h>
```

これにより、API 関数宣言と、2 つのパーティションのファームウェアイメージと一緒に保存されたフラグとタグの事前定義値をエクスポートします。

フラッシュパーティション、フラグ、および状態の詳細については、フラッシュパーティションを参照してください。

6.7.2 API

libwolfboot は、フラッシュパーティション状態に低レベルのアクセスインターフェイスを提供します。各パーティションの状態は、アプリケーションによって取得および変更できます。

アプリケーションからの基本的な相互作用は、次の高レベル関数呼び出しを介して提供されます。

```
uint32_t wolfBoot_get_image_version(uint8_t part)
void wolfBoot_update_trigger(void)
void wolfBoot_success(void)
```

6.7.2.1 ファームウェアバージョン 現在 (ブート) ファームウェアと更新ファームウェアバージョンは、以下を使用してアプリケーションから取得できます。

```
uint32_t wolfBoot_get_image_version(uint8_t part)
```

またはショートカットマクロを介して：

```
wolfBoot_current_firmware_version()
```

と

```
wolfBoot_update_firmware_version()
```

6.7.2.2 更新をトリガー -`wolfBoot_update_trigger()` は、次の再起動時に更新をトリガーするために使用され、通常、実行中のファームウェアの新しいバージョンを取得し、フラッシュ上の更新パーティションに保存した更新アプリケーションで使用されます。この関数は、更新パーティションの状態を STATE_UPDATING に設定し、ブートローダーに次の実行時に更新を実行するように指示します (再起動後)。

wolfBoot Update プロセスは、一時的なシングルブロックスワップスペースを使用して、アップデートの内容とブートパーティションをスワップします。

6.7.2.3 現在のイメージの確認

- `wolfBoot_success()` は、新しいファームウェアのブートが成功したことを示します。これはいつでもアプリケーションで呼び出すことができますが、現在のファームウェア (ブートパーティション内) を状態 STATE_SUCCESS でマークするのは効果的であり、ロールバックが不要であることを示します。通常、アプリケーションは、基本的なシステム機能が稼働していることを確認した後にのみ、`wolfBoot_success()` を呼び出す必要があります。

アップグレードと再起動の後、wolfBoot がアクティブなファームウェアがまだ STATE_TESTING 状態にあることを検出した場合、それはアプリケーションのために成功したブートが確認されておらず、2 つのイメージを再度交換して更新を戻そうとすることを意味します。

更新プロセスの詳細については、ファームウェアの更新を参照してください

イメージ形式については、ファームウェアイメージを参照してください

7　wolfBoot の既存のプロジェクトへの統合

7.1　必要な手順

- 参照実装の例については、ターゲットの章を参照してください。
- ターゲットプラットフォームの HAL 実装を提供します (ハードウェア抽象化レイヤーを参照)
- フラッシュパーティションの方針を決定し、それに応じて `include/target.h` を変更します (フラッシュパーティションを参照)
- ブートローダーの存在を考慮して、ファームウェアイメージのエントリポイントを変更します
- アプリケーションにwolfBoot ライブラリを装備して、ブートローダーと対話します
- 構成してコンパイルします単一の「make」コマンドを備えた起動可能なイメージ
- ファームウェアの署名については、wolfBoot 署名を参照してください
- メジャードブートを有効にするには、wolfBoot 管理ブートを参照してください

7.2　提供されているサンプルプログラム

GitHub wolfBoot-Examples リポジトリでも別のサンプルプログラムが入手できます。

次の手順は、工場イメージ（工場出荷時のアプリケーションイメージ）を作成するための例として非 OS のテストアプリケーションを使用して、デフォルトの `Makefile` ターゲットで自動化されています。`make` を実行することにより、ビルドシステムは次のとおりです。

- ed25519_keygen ツールを使用して、ED25519 鍵ペアを作成します
- ブートローダーをコンパイルします。上記のステップで生成された公開鍵はビルドに含まれています
- 「test_app」ディレクトリにあるテスト アプリケーションからファームウェア イメージをコンパイルします。
- ファームウェアを再リンクして、エントリポイントをプライマリパーティションの開始アドレスに変更します
- ed25519_sign ツールを使用してファームウェアイメージに署名します
- ブートローダーとファームウェアイメージを連結して、工場イメージを作成します

工場イメージはターゲットデバイスにフラッシュできます。フラッシュ上の指定されたアドレスにブートローダーと署名された初期ファームウェアが含まれています。

`sign.py` ツールは、ブートローダーが必要とするファームウェアイメージ形式に準拠するように、起動可能なファームウェアイメージを変換します。

ファームウェアイメージ形式の詳細については、ファームウェアイメージを参照してください。

ターゲットシステムの構成オプションの詳細については、WolfBoot のコンパイルを参照してください。

7.3　ファームウェアのアップグレード

- 新しいファームウェアイメージをコンパイルし、そのエントリポイントがプライマリパーティションの開始アドレスにあるようにリンクします
- `sign.py` ツールと、工場のイメージ用に生成された秘密鍵を使用してファームウェアに署名します
- 安全な接続を使用してイメージを転送し、セカンダリファームウェアスロットに保存します

- libwolfboot `wolfBoot_update_trigger()` 関数を使用してイメージスワップをトリガーします。操作の説明については、wolfBoot Library APIを参照してください
- 再起動して、ブートローダーがイメージスワップを開始します
- libwolfboot `wolfBoot_success()` 関数を使用して、更新の成功を確認します。操作の説明については、wolfBoot Library APIを参照してください

ファームウェアの更新実装の詳細については、ファームウェアの更新を参照してください。

8 トラブルシューティング

8.1 鍵に署名するときの Python エラー：

```
Traceback (most recent call last):
  File "tools/keytools/keygen.py", line 135, in <module>
    rsa=ciphers.RsaPrivate.make_key(2048)
AttributeError: type object 'RsaPrivate' has no attribute 'make_key'

Traceback (most recent call last):
  File "tools/keytools/sign.py", line 189, in <module>
    r, s=ecc.sign_raw(digest)
AttributeError: 'EccPrivate' object has no attribute 'sign_raw'
```

最新の wolfCrypt-pyをインストールする必要があります

```
pip3 install wolfcrypt
```

を使用します。

または、ローカルの wolfSSL に基づいてインストールするには：

```
cd wolfssl
./configure --enable-keygen --enable-rsa --enable-ecc --enable-ed25519 --
   enable-des3 CFLAGS="-DFP_MAX_BITS=8192 -DWOLFSSL_PUBLIC_MP"
make
sudo make install
cd wolfcrypt-py
USE_LOCAL_WOLFSSL=/usr/local pip3 install .
```

8.2 keyden.py 実行時の Python エラー：

```
Traceback (most recent call last):
  File "tools/keytools/keygen.py", line 173, in <module>
    parser.add_argument('-i', dest='pubfile', nargs='+', action='extend')
  File "/usr/lib/python3.7/argparse.py", line 1361, in add_argument
    raise ValueError('unknown action "%s"' % (action_class,))
ValueError: unknown action "extend"
```

インストールされている Python インタープリターが古すぎます。keygen.py を実行するには python を v3.8 以上に更新してください。

8.3 サポートへの問い合わせ

問題が発生してサポートが必要な場合は、support@wolfssl.com までお問い合わせください