wolfBoot Documentation

2025-04-30

Contents

1 Introduction

wolfBoot is a portable, OS-agnostic, secure bootloader solution for 32-bit microcontrollers, relying on wolfCrypt for firmware authentication, providing firmware update mechanisms.

Due to the minimalist design of the bootloader and the tiny HAL API, wolfBoot is completely independent from any OS or bare-metal application, and can be easily ported and integrated in existing embedded software projects to provide a secure firmware update mechanism.

Design based on RFC 9019 - A Firmware Update Architecture for Internet of Things.

1.1 Features

- Multi-slot partitioning of the flash device
- Integrity verification of the firmware image(s)
- Authenticity verification of the firmware image(s) using wolfCrypt's Digital Signature Algorithms (DSA)
- Minimalist hardware abstraction layer (HAL) interface to facilitate portability across different vendors/MCUs
- Copy/swap images from secondary slots into the primary slots to consent firmware update operations
- In-place chain-loading of the firmware image in the primary slot
- Support of Trusted Platform Module(TPM)
- Measured boot support, storing of the firmware image hash into a TPM Platform Configuration Register(PCR)

1.2 Components

This repository contains the following components: - the wolfBoot bootloader - key generator and image signing tools (requires python 3.x and wolfcrypt-py https://github.com/wolfSSL/wolfcrypt-py) - Baremetal test applications

2 Compiling wolfBoot

WolfBoot is portable across different types of embedded systems. The platform-specific code is contained in a single file under the `hal` directory, and implements the hardware-specific functions.

To enable specific compile options, use environment variables while calling make, e.g.

```
make CORTEX_M0=1
```

As an alternative, you can provide a .config file in the root directory of wolfBoot. Command line options have priority on `.config` options, as long as .config options are defined using the `?=` operator, e.g.:

```
WOLFBOOT_PARTITION_BOOT_ADDRESS?=0x14000
```

2.1 Generate a new configuration

A new `.config` file with a set of default parameters can be generated by running `make config`. The build script will ask to enter a default value for each configuration parameter. Enter confirm the current value, indicated in between `[]`.

Once a .config file is in place, it will change the default compile-time options when running `make` without parameters.

.config can be modified with a text editor to alter the default options later on.

Detailed parameters can be found at Appendix. N

2.2 Platform selection

If supported natively, the target platform can be specified using the TARGET variable. Make will automatically select the correct compile option, and include the corresponding HAL for the selected target.

For a list of the platforms currently supported, see the chapter on HAL.

To add a new platform, simply create the corresponding HAL driver and linker script file in the `hal` directory.

Default option if none specified: `TARGET=stm32f4`

Some platforms will require extra options, specific for the architecture. By default, wolfBoot is compiled for ARM Cortex-M3/4/7. To compile for Cortex-M0, use:

```
CORTEX_M0=1
```

2.2.1 Flash partitions

The file `include/target.h` is generated according to the configured flash geometry, partitions size and offset of the target system. The following values must be set to provide the desired flash configuration, either via the command line, or using the .config file:

- `WOLFBOOT_SECTOR_SIZE`

This variable determines the size of the physical sector on the flash memory. If areas with different block sizes are used for the two partitions (e.g. update partition on an external flash), this variable should indicate the size of the biggest sector shared between the two partitions.

WolfBoot uses this value as minimum unit when swapping the firmware images in place. For this reason, this value is also used to set the size of the SWAP partition.

- `WOLFBOOT_PARTITION_BOOT_ADDRESS`

This is the start address of the boot partition, aligned to the beginning of a new flash sector. The application code starts after a further offset, equal to the partition header size (256B for Ed25519 and ECC signature headers).

- WOLFBOOT_PARTITION_UPDATE_ADDRESS

This is the start address of the update partition. If an external memory is used via the EXT_FLASH option, this variable contains the offset of the update partition from the beginning of the external memory addressable space.

- WOLFBOOT_PARTITION_SWAP_ADDRESS

The address for the swap spaced used by wolfBoot to swap the two firmware images in place, in order to perform a reversable update. The size of the SWAP partition is exactly one sector on the flash. If an external memory is used, the variable contains the offset of the SWAP area from the beginning of its addressable space.

- WOLFBOOT_PARTITION_SIZE

The size of the BOOT and UPDATE partition. The size is the same for both partitions.

2.3 Bootloader features

A number of characteristics can be turned on/off during wolfBoot compilation. Bootloader size, performance and activated features are affected by compile-time flags.

2.3.1 Change DSA algorithm

By default, wolfBoot is compiled to use Ed25519 DSA. The implementation of ed25519 is smaller, while giving a good compromise in terms of boot-up time.

Better performance can be achieved using ECDSA with curve p-256. To activate ECC256 support, use

SIGN=ECC256 or SIGN=ECC384 or SIGN=ECC521 respectively.

when invoking make.

RSA is also supported, with different key length. To activate RSA2048, RSA3072 or RSA4096, use:

SIGN=RSA2048 or SIGN=RSA3072 or SIGN=RSA4096 respectively.

Ed448 is also supported via SIGN=ED448.

The default option, if no value is provided for the SIGN variable, is

SIGN=ED25519

Changing the DSA algorithm will also result in compiling a different set of tools for key generation and firmware signature.

Find the corresponding key generation and firmware signing tools in the tools directory.

It's possible to disable authentication of the firmware image by explicitly using:

SIGN=NONE

in the Makefile commandline. This will compile a minimal bootloader with no support for public-key authenticated secure boot.

2.3.2 Incremental updates

wolfBoot support incremental updates. To enable this feature, compile with `DELTA_UPDATES=1`.

An additional file is generated when the sign tool is invoked with the `--delta` option, containing only the differences between the old firmware to replace, currently running on the target, and the new version.

For more information and examples, see the firmware update section.

2.3.3 Enable debug symbols

To debug the bootloader, simply compile with `DEBUG=1`. The size of the bootloade will increase consistently, so ensure that you have enough space at the beginning of the flash before `WOLFBOOT_PARTITION_BOOT_ADDRESS`.

2.3.4 Disable interrupt vector relocation

On some platforms, it might be convenient to avoid the interrupt vector relocation before boot-up. This is required when a component on the system already manages the interrupt relocation at a different stage, or on these platform that do not support interrupt vector relocation.

To disable interrupt vector table relocation, compile with `VTOR=0`. By default, wolfBoot will relocate the interrupt vector by setting the offset in the vector relocation offset register (VTOR).

2.3.5 Limit stack usage

By default, wolfBoot does not require any memory allocation. It does this by performing all the operations using the stack. Although the stack space used by the algorithms can be predicted at compile time, the amount of stack space be relatively big, depending on the algorithm selected.

Some targets offer limited amount of RAM to use as stack space, either in general, or in a configuration dedicated for the bootloader stage.

In these cases, it might be useful to activate `WOLFBOOT_SMALL_STACK=1`. With this option, a fixed-size pool is created at compile time to assist the allocation of the object needed by the cryptography implementation. When compiled with `WOLFBOOT_SMALL_STACK=1`, wolfBoot reduces the stack usage considerably, and simulates dynamic memory allocations by assigning dedicated, statically allocated, pre-sized memory areas.

2.3.6 Allow bigger stack size allocation

Some combinations of authentication algorithms, key sizes and math configuration in wolfCrypt require a large amount of memory to be allocated in the stack at runtime. By default, if your configuration falls in one of these cases, wolfBoot compilation will terminate with an explicit error.

In some cases you might have enough memory available to allow large stack allocations. To circumvent the compile-time checks on the maximum allowed stack size, use `WOLFBOOT_HUGE_STACK=1`.

2.3.7 Disable Backup of current running firmware

Optionally, it is possible to disable the backup copy of the current running firmware upon the installation of the update. This implies that no fall-back mechanism is protecting the target from a faulty firmware installation, but may be useful in some cases where it is not possible to write on the update partition from the bootloader. The associated compile-time option is

`DISABLE_BACKUP=1`

2.3.8 Enable workaround for 'write once' flash memories

On some microcontrollers, the internal flash memory does not allow subsequent writes (adding zeroes) to a sector, after the entire sector has been erased. WolfBoot relies on the mechanism of adding zeroes to the 'flags' fields at the end of both partitions to provide a fail-safe swap mechanism.

To enable the workaround for 'write once' internal flash, compile with

`NVM_FLASH_WRITEONCE=1`

warning When this option is enabled, the fail-safe swap is not guaranteed, i.e. the microcontroller cannot be safely powered down or restarted during a swap operation.

2.3.9 Allow version roll-back

WolfBoot will not allow updates to a firmware with a version number smaller than the current one. To allow downgrades, compile with `ALLOW_DOWNGRADE=1`.

Warning: this option will disable version checking before the updates, thus exposing the system to potential forced downgrade attacks.

2.3.10 Enable optional support for external flash memory

WolfBoot can be compiled with the makefile option `EXT_FLASH=1`. When the external flash support is enabled, update and swap partitions can be associated to an external memory, and will use alternative HAL function for read/write/erase access. To associate the update or the swap partition to an external memory, define `PART_UPDATE_EXT` and/or `PART_SWAP_EXT`, respectively. By default, the makefile assumes that if an external memory is present, both `PART_UPDATE_EXT` and `PART_SWAP_EXT` are defined.

If the `NO_XIP=1` makefile option is present, `PART_BOOT_EXT` is assumed too, as no execute-in-place is available on the system. This is typically the case of MMU system (e.g. Cortex-A) where the operating system image(s) are position-independent ELF images stored in a non-executable non-volatile memory, and must be copied in RAM to boot after verification.

When external memory is used, the HAL API must be extended to define methods to access the custom memory. Refer to the HAL chapter for the description of the `ext_flash_*` API.

The `EXT_FLASH` option can also be used if the target device requires special handling for flash reads (e.g. word size requirements or other restrictions), regardless of whether the flash is internal or external.

Note that the `EXT_FLASH` option is incompatible with the `NVM_FLASH_WRITEONCE` option. Targets that need both these options must implement the sector-based read-modify-erase-write sequence at the HAL layer.

For an example of using `EXT_FLASH` to bypass read restrictions, (in this case, the inability to read from erased flash due to ECC errors) on a platform with write-once flash, see the infineon tricore port(`hal/aurix_tc3xx.c`).

2.3.10.1 SPI devices In combination with the `EXT_FLASH=1` configuration parameter, it is possible to use a platform-specific SPI drivers, e.g. to access an external SPI flash memory. By compiling wolfBoot with the makefile option `SPI_FLASH=1`, the external memory is directly mapped to the additional SPI layer, so the user does not have to define the `ext_flash_*` functions.

SPI functions, instead, must be defined. Example SPI drivers are available for multiple platforms in the `hal/spi` directory.

2.3.10.2 UART bridge towards neighbor systems Another alternative available to map external devices consists in enabling a UART bridge towards a neighbor system. The neighbor system must expose a service through the UART interface that is compatible with the wolfBoot protocol.

In the same way as for SPI devices, the `ext_flash_*` API is automatically defined by wolfBoot when the option `UART_FLASH=1` is used.

For more details, see the section Remote External flash memory support via UART

2.3.10.3 Encryption support for external partitions When update and swap partitions are mapped to an external device using `EXT_FLASH=1`, either in combination with `SPI_FLASH`, `UART_FLASH`, or any custom external mapping, it is possible to enable ChaCha20, Aes128 or Aes256 encryption when accessing those partition from the bootloader. The update images must be pre-encrypted at the source using the key tools, and wolfBoot should be instructed to use a temporary ChaCha20 symmetric key to access the content of the updates.

For more details about this optional feature, please refer to the Encrypted external partitions section.

2.3.11 Executing flash access code from RAM

On some platform, flash access code requires to be executed from RAM, to avoid conflict e.g. when writing to the same device where wolfBoot is executing, or when changing the configuration of the flash itself.

To move all the code accessing the internal flash for writing, into a section in RAM, use the compile time option `RAM_CODE=1` (on some hardware configurations this is required for the bootloader to access the flash for writing).

2.3.12 Enable Dual-bank hardware-assisted swapping

When supported by the target platform, hardware-assisted dual-bank swapping can be used to perform updates. To enable this functionality, use `DUALBANK_SWAP=1`. Currently, only STM32F76x and F77x support this feature.

2.3.13 Store UPDATE partition flags in a sector in the BOOT partition

By default, wolfBoot keeps track of the status of the update procedure to the single sectors in a specific area at the end of each partition, dedicated to store and retrieve a set of flags associated to the partition itself.

In some cases it might be helpful to store the status flags related to the UPDATE partition and its sectors in the internal flash, alongside with the same set of flags used for the BOOT partition. By compiling wolfBoot with the `FLAGS_HOME=1` makefile option, the flags associated to the UPDATE partition are stored in the BOOT partition itself.

While on one hand this option slightly reduces the space available in the BOOT partition to store the firmware image, it keeps all the flags in the BOOT partition.

2.3.14 Flash Erase value / Flag logic inversion

By default, most NVMs set the content of erased pages to `0xFF` (all ones).

Some FLASH memory models use inverted logic for erased page, setting the content to `0x00` (all zeroes) after erase.

For these special cases, the option `FLAGS_INVERT = 1` can be used to modify the logic of the partition/sector flags used in wolfBoot.

You can also manually override the fill bytes using FILL_BYTE= at build-time. It default to 0xFF, but will use 0x00 if FLAGS_INVERT is set.

Note: if you are using an external FLASH (e.g. SPI) in combination with a flash with inverted logic, ensure that you store all the flags in one partition, by using the FLAGS_HOME=1 option described above.

2.3.15 Using One-time programmable (OTP) flash as keystore

By default, keys are directly incorporated in the firmware image. To store the keys in a separate, one-time programmable (OTP) flash memory, use the FLASH_OTP_KEYSTORE=1 option. For more information, see Appendix C.

2.3.16 Prefer multi-sector flash erase operations

wolfBoot HAL flash erase function must be able to handle erase lengths larger than WOLFBOOT_SECTOR_SIZE, even if the underlying flash controller does not. However, in some cases, wolfBoot defaults to iterating over a range of flash sectors and erasing them one at a time. Setting the FLASH_MULTI_SECTOR_ERASE=1 config option prevents this behavior when possible, configuring wolfBoot to instead prefer a single HAL flash erase invocation with a larger erase length versus the iterative approach. On targets where multi-sector erases are more performant, this option can be used to dramatically speed up the image swap procedure. ### Using Mac OS/X

If you see 0xC3 0xBF (C3BF) repeated in your factory.bin then your OS is using Unicode characters.

The “tr” command for assembling the 0xFF padding between "bootloader" ... 0xFF ... "application" = factory.bin, which requires the “C” locale.

Set this in your terminal

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

Then run the normal make steps.

3 Targets

This chapter describes configuration of supported targets.

3.1 Supported Targets

- Cortex-A53 / Raspberry PI 3
- Cypress PSoC-6
- Microchip SAME51
- Nordic nRF52840
- NXP LPC54xxx
- NXP iMX-RT
- NXP Kinetis
- NXP P1021 PPC
- NXP T1024 PPC
- NXP T2080 PPC
- NXP MCXA153
- SiFive HiFive1 RISC-V
- STM32F4
- STM32F7
- STM32L0
- STM32L4
- STM32L5
- STM32G0
- STM32C0
- STM32H5
- STM32H7
- STM32U5
- STM32WB55
- TI Hercules TMS570LC435
- Xilinx Zynq UltraScale
- Renesas RX65N
- Renesas RX72N
- Renesas RA6M4
- Renesas RZN2L
- Qemu x86-64 UEFI
- Intel x86-64 Intel FSP

3.2 STM32F4

Example 512KB partitioning on STM32-F407

The example firmware provided in the `test-app` is configured to boot from the primary partition starting at address 0x20000. The flash layout is provided by the default example using the following configuration in `target.h`:

```
#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
```

This results in the following partition configuration:

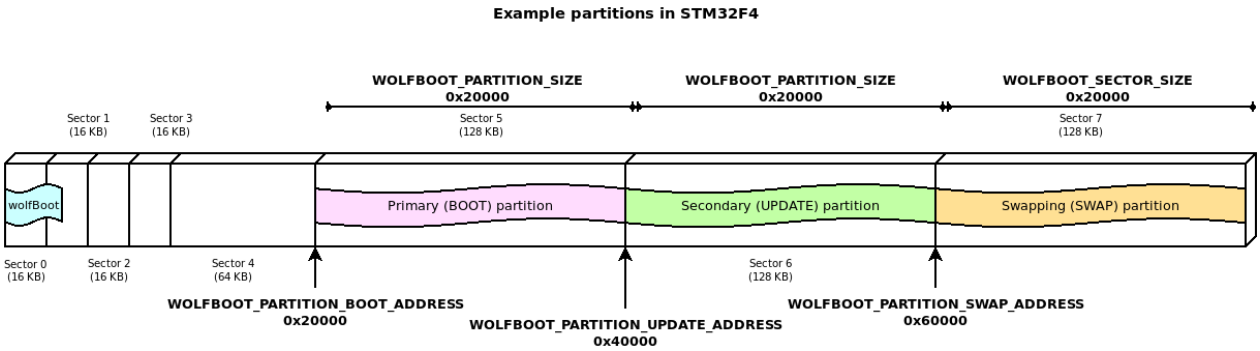

Figure 1: example partitions

This configuration demonstrates one of the possible layouts, with the slots aligned to the beginning of the physical sector on the flash.

The entry point for all the runnable firmware images on this target will be `0x20100`, 256 Bytes after the beginning of the first flash partition. This is due to the presence of the firmware image header at the beginning of the partition, as explained more in details in Firmware image

In this particular case, due to the flash geometry, the swap space must be as big as 128KB, to account for proper sector swapping between the two images.

On other systems, the SWAP space can be as small as 512B, if multiple smaller flash blocks are used.

More information about the geometry of the flash and in-application programming (IAP) can be found in the manufacturer manual of each target device.

3.2.1 STM32F4 Programming

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

3.2.2 STM32F4 Debugging

1. Start GDB server

OpenOCD: `openocd --file ./config/openocd/openocd_stm32f4.cfg` OR ST-Link: `st-util -p 3333`

2. Start GDB Client

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

3.3 STM32L4

Example 1MB partitioning on STM32L4

- Sector size: 4KB
- Wolfboot partition size: 40 KB
- Application partition size: 488 KB

```
#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 Scenario 1: TrustZone Enabled

3.4.1.1 Example Description The implementation shows how to switch from secure application to non-secure application, thanks to the system isolation performed, which splits the internal Flash and internal SRAM memories into two parts: - the first half is used by wolfboot running in secure mode and the secure application - the remaining available space is used for non-secure application and update partition

The example configuration for this scenario is available in `/config/examples/stm32l5.config`.

3.4.1.2 Hardware and Software environment

- This example runs on STM32L562QEIxQ devices with security enabled (TZEN=1).
- This example has been tested with STMicroelectronics STM32L562E-DK (MB1373)
- User Option Bytes requirement (with STM32CubeProgrammer tool - see below for instructions)

```
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
```

- NOTE: STM32CubeProgrammer V2.3.0 is required (v2.4.0 has a known bug for STM32L5)

3.4.1.3 How to use it

1. `cp ./config/examples/stm32l5.config .config`
2. `make`
3. Prepare board with option bytes configuration reported above
   - `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. flash wolfBoot.bin to 0x0c00 0000
   - `STM32_Programmer_CLI -c port=swd -d ./wolfboot.bin 0x0C000000`
5. flash `.\test-app\image_v1_signed.bin` to 0x0804 0000
   - `STM32_Programmer_CLI -c port=swd -d ./test-app/image_v1_signed.bin 0x08040000`
6. RED LD9 will be on

- NOTE: STM32_Programmer_CLI Default Locations
- Windows: `C:\Program Files\STMicroelectronics\STM32Cube\STM32CubeProgrammer\bin\STM32_Programmer_CLI.exe`
- Linux: `/usr/local/STMicroelectronics/STM32Cube/STM32CubeProgrammer/bin/STM32_Programmer_CLI`
- Mac OS/X: `/Applications/STMicroelectronics/STM32Cube/STM32CubeProgrammer/STM32CubeProgrammer.app/Contents/MacOs/bin/STM32_Programmer_CLI`

3.4.2 Scenario 2: Trustzone Enabled, wolfCrypt as secure engine for NS applications

This is similar to Scenario 1, but also includes wolfCrypt in secure mode, and that can be accessed via PKCS11 interface by non-secure applications.

This option can be enabled with the `WOLFCRYPT_TZ=1` and `WOLFCRYPT_TZ_PKCS11=1` options in your configuration. This enables a PKCS11 accessible from NS domain via non-secure callables (NSC).

The example configuration for this scenario is available in `/config/examples/stm32l5-wolfcrypt-tz.config`.

For more information, see Appendix L.

3.4.3 Scenario 3: Trustzone Disabled, using DUAL BANK

3.4.3.1 Example Description The implementation shows how to use STM32L5xx in DUAL BANK mode, with TrustZone disabled. The DUAL_BANK option is only available on this target when TrustZone is disabled (TZEN = 0).

The flash memory is segmented into two different banks:

- Bank 0: (0x08000000)
- Bank 1: (0x08040000)

Bank 0 contains the bootloader at address 0x08000000, and the application at address 0x08040000. When a valid image is available at the same offset in Bank 1, a candidate is selected for booting between the two valid images. A firmware update can be uploaded at address 0x08048000.

The example configuration is available in `/config/examples/stm32l5-nonsecure-dualbank.config`.

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

Or program each partition using: 1. flash `wolfboot.bin` to 0x08000000: - `STM32_Programmer_CLI -c port=swd -d ./wolfboot.elf` 2. flash main application to 0x0800 a000 - `STM32_Programmer_CLI -c port=swd -d ./test-app/image_v1_signed.bin 0x0800a000`

RED LD9 will be on indicating successful boot ().

Updates can be flashed at 0x0804a000:

- `STM32_Programmer_CLI -c port=swd -d ./test-app/image_v2_signed.bin 0x0804a000`

The two partition are logically remapped by using BANK_SWAP capabilities. This partition swap is immediate and does not require a SWAP partition.

3.4.4 Debugging

Use `make DEBUG=1` and reload firmware.

- STM32CubeIDE v.1.3.0 required
- Run the debugger via:

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 /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
```

- Connect with arm-none-eabi-gdb

wolfBoot has a .gdbinit to configure

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

3.5 STM32U5

The STM32U5 is a Cortex-M33 (ARMv8-M).

Note: We have seen issues with vector table alignment, so the default image header size (IMAGE_HEADER_SIZE) has been increased to 1024 bytes to avoid potential issues.

3.5.1 Scenario 1: TrustZone enabled, staging non-secure application

3.5.1.1 Example description The implementation shows how to switch from secure application to non-secure application, thanks to the system isolation performed, which splits the internal Flash and internal SRAM memories into two parts: - the first 256KB are used by wolfboot running in secure mode and the secure application - the remaining available space is used for non-secure application and update partition

The example configuration for this scenario is available in `/config/examples/stm32u5.config`.

3.5.1.2 Example Description The implementation shows how to switch from secure application to non-secure application, thanks to the system isolation performed, which splits the internal Flash and internal SRAM memories into two parts: - the first half for secure application - the second half for non-secure application

3.5.1.3 Hardware and Software environment

- This example runs on STM32U585AII6Q devices with security enabled (TZEN=1).
- This example has been tested with STMicroelectronics B-U585I-IOT02A (MB1551)
- User Option Bytes requirement (with STM32CubeProgrammer tool - see below for instructions)

```
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
```

- NOTE: STM32CubeProgrammer V2.8.0 or newer is required

3.5.1.4 How to use it

1. `cp ./config/examples/stm32u5.config .config`
2. `make TZEN=1`
3. Prepare board with option bytes configuration reported above
   - `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. flash wolfBoot.bin to 0x0c000000
   - `STM32_Programmer_CLI -c port=swd -d ./wolfboot.bin 0x0C000000`
5. flash `.\test-app\image_v1_signed.bin` to 0x08010000
   - `STM32_Programmer_CLI -c port=swd -d ./test-app/image_v1_signed.bin 0x08100000`
6. RED LD9 will be on

- NOTE: STM32_Programmer_CLI Default Locations
- Windows: `C:\Program Files\STMicroelectronics\STM32Cube\STM32CubeProgrammer\bin\STM32_Programmer_CLI.exe`
- Linux: `/usr/local/STMicroelectronics/STM32Cube/STM32CubeProgrammer/bin/STM32_Programmer_CLI`
- Mac OS/X: `/Applications/STMicroelectronics/STM32Cube/STM32CubeProgrammer/STM32CubeProgrammer.app/Contents/MacOs/bin/STM32_Programmer_CLI`

3.5.2 Scenario 2: TrustZone Enabled, wolfCrypt as secure engine for NS applications

This is similar to Scenario 1, but also includes wolfCrypt in secure mode, and that can be accessed via PKCS11 interface by non-secure applications.

This option can be enabled with the `WOLFCRYPT_TZ=1` and `WOLFCRYPT_TZ_PKCS11=1` options in your configuration. This enables a PKCS11 accessible from NS domain via non-secure callables (NSC).

The example configuration for this scenario is available in `/config/examples/stm32u5-wolfcrypt-tz.config`.

For more information, see Appendix L.

3.5.3 Scenario 3: TrustZone Disabled (DUAL BANK mode)

3.5.3.1 Example Description The implementation shows how to use STM32U5xx in DUAL_BANK mode, with TrustZone disabled. The DUAL_BANK option is only available on this target when TrustZone is disabled (TZEN = 0).

The flash memory is segmented into two different banks:

- Bank 0: (0x08000000)
- Bank 1: (0x08100000)

Bank 0 contains the bootloader at address 0x08000000, and the application at address 0x08100000. When a valid image is available at the same offset in Bank 1, a candidate is selected for booting between the two valid images. A firmware update can be uploaded at address 0x08108000.

The example configuration is available in `config/examples/stm32u5-nonsecure-dualbank.config`.

Program each partition using: 1. flash `wolfboot.bin` to 0x08000000: - `STM32_Programmer_CLI -c port=swd -d ./wolfboot.bin 0x08000000` 2. flash `image_v1_signed.bin` to 0x08008000 - `STM32_Programmer_CLI -c port=swd -d ./test-app/image_v1_signed.bin 0x08008000`

RED LD9 will be on indicating successful boot ()

3.5.4 Debugging

Use `make DEBUG=1` and reload firmware.

- STM32CubeIDE v.1.7.0 required
- Run the debugger via:

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:

```
/Applications/STM32CubeIDE.app/Contents/Eclipse/plugins/\
com.st.stm32cube.ide.mcu.externaltools.\
stlink-gdb-server.macos64_2.1.300.202403291623/tools/bin/\
ST-LINK_gdbserver -d -cp /Applications/STM32CubeIDE.app/\
Contents/Eclipse/plugins/com.st.stm32cube.ide.mcu.\
externaltools.cubeprogrammer.macos64_2.1.201.202404072231/tools/\
bin -e -r 1 -p 3333
```

Win:

```
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`
```

- Connect with arm-none-eabi-gdb or gdb-multiarch

wolfBoot has a .gdbinit to configure

```
add-symbol-file test-app/image.elf
```

3.6 STM32L0

Example 192KB partitioning on STM32-L073

This device is capable of erasing single flash pages (256B each).

However, we choose to use a logic sector size of 4KB for the swaps, to limit the amount of writes to the swap partition.

The proposed geometry in this example `target.h` uses 32KB for wolfBoot, and two partitions of 64KB each, leaving room for up to 8KB to use for swap (4K are being used here).

```
#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 Building

Use `make TARGET=stm32l0`. The option `CORTEX_M0` is automatically selected for this target.

3.7 STM32G0

Supports STM32G0x0x0/STM32G0x1.

Example 128KB partitioning on STM32-G070:

- Sector size: 2KB
- Wolfboot partition size: 32KB
- Application partition size: 44 KB

```
#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 Building STM32G0

Reference configuration (see `/config/examples/stm32g0.config`). You can copy this to wolfBoot root as `.config`: `cp ./config/examples/stm32g0.config .config`. To build you can use `make`.

The TARGET for this is `stm32g0`: `make TARGET=stm32g0`. The option `CORTEX_M0` is automatically selected for this target. The option `NVM_FLASH_WRITEONCE=1` is mandatory on this target, since the IAP driver does not support multiple writes after each erase operation.

3.7.1.1 STM32G0 Secure Hide Protection Feature (Optional)

This part supports a "secure memory protection" feature makes the wolfBoot partition unaccessible after jump to application.

It uses the `FLASH_CR:SEC_PROT` and `FLASH_SECT:SEC_SIZE` registers. This is the number of 2KB pages to block access to from the 0x8000000 base address.

Command example to enable this for 32KB bootloader:

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

Enabled with `CFLAGS_EXTRA+=-DFLASH_SECURABLE_MEMORY_SUPPORT`. Requires `RAM_CODE=1` to enable RAMFUNCTION support.

3.7.2 STM32G0 Programming

Compile requirements: `make TARGET=stm32g0 NVM_FLASH_WRITEONCE=1`

The output is a single `factory.bin` that includes `wolfboot.bin` and `test-app/image_v1_signed.bin` combined together. This should be programmed to the flash start address `0x08000000`.

Flash using the STM32CubeProgrammer CLI:

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

3.7.3 STM32G0 Debugging

Use `make DEBUG=1` and program firmware again.

Start GDB server on port 3333:

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

wolfBoot has a .gdbinit to configure GDB

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

3.8 STM32C0

Supports STM32C0x0/STM32C0x1. Instructions are for the STM Nucleo-C031C6 dev board.

Tested build configurations: * With RSA2048 and SHA2-256 the code size is 10988 and it boots in under 1 second. * With ED25519 and SHA2-384 the code size is 10024 and takes about 10 seconds for the LED to turn on. * With LMS-8-10-1 and SHA2-256 the code size is 8164 on gcc-13 (could fit in 8KB partition)

3.8.1 Example 32KB partitioning on STM32-G070

with ED25519 or LMS-8-10-1:

- Sector size: 2KB
- Wolfboot partition size: 10KB
- Application partition size: 10 KB
- Swap size 2KB

```
#define WOLFBOOT_SECTOR_SIZE                0x800   /* 2 KB */
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x08002800 /* at 10KB */
#define WOLFBOOT_PARTITION_SIZE             0x2800  /* 10 KB */
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x08005000 /* at 20KB */
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x08007800 /* at 30KB */
```

with RSA2048:

- Sector size: 2KB
- Wolfboot partition size: 12KB
- Application partition size: 8 KB
- Swap size 2KB

```
#define WOLFBOOT_SECTOR_SIZE                0x800      /* 2 KB */
#define WOLFBOOT_PARTITION_BOOT_ADDRESS     0x08003000 /* at 12KB */
#define WOLFBOOT_PARTITION_SIZE             0x2000     /* 8 KB */
#define WOLFBOOT_PARTITION_UPDATE_ADDRESS   0x08005000 /* at 20KB */
#define WOLFBOOT_PARTITION_SWAP_ADDRESS     0x08007800 /* at 30KB */
```

3.8.2 Building STM32C0

Reference configuration files (see `config/examples/stm32c0.config`, `config/examples/stm32c0-rsa2048.config` and `config/examples/stm32c0-lms-8-10-1.config`).

You can copy one of these to wolfBoot root as `.config`: `cp ./config/examples/stm32c0.config .config`. To build you can use `make`.

The TARGET for this is `stm32c0`: `make TARGET=stm32c0`. The option `CORTEX_M0` is automatically selected for this target. The option `NVM_FLASH_WRITEONCE=1` is mandatory on this target, since the IAP driver does not support multiple writes after each erase operation.

3.8.2.1 STM32C0 Secure Hide Protection Feature (Optional)

This part supports a "secure memory protection" feature makes the wolfBoot partition unaccessible after jump to application.

It uses the `FLASH_CR:SEC_PROT` and `FLASH_SECT:SEC_SIZE` registers. This is the number of 2KB pages to block access to from the 0x8000000 base address.

Command example to enable this for 10KB bootloader:

```
STM32_Programmer_CLI -c port=swd mode=hotplug -ob SEC_SIZE=0x05
```

Enabled with CFLAGS_EXTRA+=-DFLASH_SECURABLE_MEMORY_SUPPORT. Requires RAM_CODE=1 to enable RAMFUNCTION support.

3.8.3 STM32C0 Programming

Compile requirements: `make TARGET=stm32c0 NVM_FLASH_WRITEONCE=1`

The output is a single `factory.bin` that includes `wolfboot.bin` and `test-app/image_v1_signed.bin` combined together. This should be programmed to the flash start address `0x08000000`.

Flash using the STM32CubeProgrammer CLI:

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

3.8.4 STM32C0 Debugging

Use `make DEBUG=1` and program firmware again.

Start GDB server on port 3333:

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

wolfBoot has a .gdbinit to configure GDB

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

3.9 STM32WB55

Example partitioning on Nucleo-68 board:

- Sector size: 4KB
- Wolfboot partition size: 32 KB
- Application partition size: 128 KB

```
#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
```

3.9.1 STM32WB55 Building

Use `make TARGET=stm32wb`.

The option NVM_FLASH_WRITEONCE=1 is mandatory on this target, since the IAP driver does not support multiple writes after each erase operation.

Compile with:

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

3.9.2 STM32WB55 with OpenOCD

```
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
```

3.9.3 STM32WB55 with ST-Link

```
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
```

3.9.4 STM32WB55 Debugging

Use `make DEBUG=1` and reload firmware.

wolfBoot has a .gdbinit to configure

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

3.10 SiFive HiFive1 RISC-V

3.10.1 Features

- E31 RISC-V 320MHz 32-bit processor
- Onboard 16KB scratchpad RAM
- External 4MB QSPI Flash

3.10.2 Default Linker Settings

- FLASH: Address 0x20000000, Len 0x6a120 (424 KB)
- RAM: Address 0x80000000, Len 0x4000 (16 KB)

3.10.3 Stock bootloader

Start Address: 0x20000000 is 64KB. Provides a “double tap” reset feature to halt boot and allow debugger to attach for reprogramming. Press reset button, when green light comes on press reset button again, then board will flash red.

3.10.4 Application Code

Start Address: 0x20010000

3.10.5 wolfBoot configuration

The default wolfBoot configuration will add a second stage bootloader, leaving the stock "double tap" bootloader as a fallback for recovery. Your production implementation should replace this and partition addresses in `target.h` will need updated, so they are `0x10000` less.

To set the Freedom SDK location use `FREEDOM_E_SDK=~/src/freedom-e-sdk`.

For testing wolfBoot here are the changes required:

1. Makefile arguments:
   - ARCH=RISCV
   - TARGET=hifive1

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

   If using the `riscv64-unknown-elf-` cross compiler you can add `CROSS_COMPILE=riscv64-unknown-elf-` to your `make` or modify `arch.mk` as follows:

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

2. `include/target.h`

Bootloader Size: 0x10000 (64KB) Application Size 0x40000 (256KB) Swap Sector Size: 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
```

3.10.6 Build Options

- To use ECC instead of ED25519 use make argument `SIGN=ECC256`
- To output wolfboot as hex for loading with JLink use make argument `wolfboot.hex`

3.10.7 Loading

Loading with JLink:

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

3.10.8 Debugging

Debugging with JLink:

In one terminal: `JLinkGDBServer -device FE310 -port 3333`

In another terminal:

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

3.11 STM32F7

The STM32-F76x and F77x offer dual-bank hardware-assisted swapping. The flash geometry must be defined beforehand, and wolfBoot can be compiled to use hardware assisted bank-swapping to perform updates.

Example 2MB partitioning on STM32-F769:

- Dual-bank configuration

BANK A: 0x08000000 to 0x080FFFFFF (1MB) BANK B: 0x08100000 to 0x081FFFFFF (1MB)

- WolfBoot executes from BANK A after reboot (address: 0x08000000)
- Boot partition @ BANK A + 0x20000 = 0x08020000
- Update partition @ BANK B + 0x20000 = 0x08120000
- Application entry point: 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
 ↪  */
```

3.11.1 Build Options

To activate the dual-bank hardware-assisted swap feature on STM32F76x/77x, use the `DUALBANK_SWAP=1` compile time option. Some code requires to run in RAM during the swapping of the images, so the compile-time option `RAMCODE=1` is also required in this case.

Dual-bank STM32F7 build can be built using:

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

3.11.2 Loading the firmware

To switch between single-bank (1x2MB) and dual-bank (2 x 1MB) mode mapping, this stm32f7-dualbank-tool can be used. Before starting openocd, switch the flash mode to dualbank (e.g. via `make dualbank` using the dualbank tool).

OpenOCD configuration for flashing/debugging, can be copied into `openocd.cfg` in your working directory:

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

OpenOCD can be either run in background (to allow remote GDB and monitor terminal connections), or directly from command line, to execute terminal scripts.

If OpenOCD is running, local TCP port 4444 can be used to access an interactive terminal prompt. `telnet localhost 4444`

Using the following openocd commands, the initial images for wolfBoot and the test application are loaded to flash in bank 0:

```
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
```

To sign the same application image as new version (2), use the `sign` tool provided:

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

From OpenOCD, the updated image (version 2) can be flashed to the second bank:

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

Upon reboot, wolfboot will elect the best candidate (version 2 in this case) and authenticate the image. If the accepted candidate image resides on BANK B (like in this case), wolfBoot will perform one bank swap before booting.

The bank-swap operation is immediate and a SWAP image is not required in this case. Fallback mechanism can rely on a second choice (older firmware) in the other bank.

3.11.3 STM32F7 Debugging

Debugging with OpenOCD:

Use the OpenOCD configuration from the previous section to run OpenOCD.

From another console, connect using gdb, e.g.:

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

3.12 STM32H5

Like STM32L5 and STM32U5, STM32H5 support is also demonstrated through different scenarios.

Additionally, wolfBoot can be compiled with `FLASH_OTP_KEYSTORE` option, to store the public key(s) used for firmware authentication into a dedicated, one-time programmable flash area that can be write protected. For more information, see Appendix C.

3.12.1 Scenario 1: TrustZone enabled, staging non-secure application

3.12.1.1 Example description The implementation shows how to switch from secure application to non-secure application, thanks to the system isolation performed, which splits the internal Flash and internal SRAM memories into two parts: - the first 256KB are used by wolfboot running in secure mode and the secure application - the remaining available space is used for non-secure application and update partition

The example configuration for this scenario is available in `/config/examples/stm32h5.config`.

3.12.1.2 How to use it

- set the option bytes to enable trustzone:

```
STM32_Programmer_CLI -c port=swd -ob TZEN=0xB4
```

- set the option bytes to enable flash secure protection of first 256KB: `STM32_Programmer_CLI -c port=swd -ob SECWM1_PSTRT=0x0 SECWM1_PEND=0x1F SECWM2_PSTRT=0x1F SECWM2_PEND=0x0`
- flash the wolfboot image to the secure partition: `STM32_Programmer_CLI -c port=swd -d wolfboot.bin 0x0C000000`
- flash the application image to the non-secure partition: `STM32_Programmer_CLI -c port=swd -d test-app/image_v1_signed.bin 0x08040000`

For a full list of all the option bytes tested with this configuration, refer to Appendix L.

3.12.2 Scenario 2: TrustZone Enabled, wolfCrypt as secure engine for NS applications

This is similar to Scenario 1, but also includes wolfCrypt in secure mode, and that can be accessed via PKCS11 interface by non-secure applications.

This option can be enabled with the `WOLFCRYPT_TZ=1` and `WOLFCRYPT_TZ_PKCS11=1` options in your configuration. This enables a PKCS11 accessible from NS domain via non-secure callables (NSC).

The example configuration for this scenario is available in `/config/examples/stm32h5-tz.config`.

For more information, see Appendix L.

3.12.3 Scenario 3: DUALBANK mode

The STM32H5 can be configured to use hardware-assisted bank swapping to facilitate the update. The configuration file to copy into `.config` is `config/examples/stm32h5-dualbank.config`.

For DUALBANK with TrustZone use `stm32h5-tz-dualbank-otp.config`.

DUALBANK configuration (Tested on NUCLEO-STM32H563ZI):

BANK A: 0x08000000 to 0x080FFFFFF (1MB) BANK B: 0x08100000 to 0x081FFFFFF (1MB)

First of all, ensure that the `SWAP_BANK` option byte is off when running wolfBoot for the first time:

```
STM32_Programmer_CLI -c port=swd -ob SWAP_BANK=0
```

It is a good idea to start with an empty flash, by erasing all sectors via:

```
STM32_Programmer_CLI -c port=swd -e 0 255
```

Compile wolfBoot with `make`. The file `factory.bin` contains both wolfboot and the version 1 of the application, and can be uploaded to the board at the beginning of the first bank using `STM32_Programmer_CLI` tool:

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

Optionally, you can upload another copy of wolfboot.bin to the beginning of the second bank. Wolfboot should take care of copying itself to the second bank upon first boot if you don't.:

```
STM32_Programmer_CLI -c port=swd -d wolfboot.bin 0x08100000
```

After uploading the images, reboot your board. The green LED should indicate that v1 of the test application is running.

To initiate an update, sign a new version of the app and upload the v3 to the update partition on the second bank:

```
tools/keytools/sign --ecc256 test-app/image.bin wolfboot_signing_private_key.
    der 3
STM32_Programmer_CLI -c port=swd -d test-app/image_v3_signed.bin 0x08110000
```

Reboot the board to initiate an update via DUALBANK hw-assisted swap. Any version except the first one will also turn on the orange LED.

3.13 STM32H7

The STM32H7 flash geometry must be defined beforehand.

Use the "make config" operation to generate a .config file or copy the template using `cp ./config/examples/stm32h7.config .config`.

Example 2MB partitioning on STM32-H753:

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

3.13.1 Build Options

The STM32H7 build can be built using:

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

The STM32H7 also supports using the QSPI for external flash. To enable use `QSPI_FLASH=1` in your configuration. The pins are defined in `hal/spi/spi_drv_stm32.h`. A built-in alternate pin configuration can be used with `QSPI_ALT_CONFIGURATION`. The flash and QSPI parameters are defined in `src/qspi_flash.c` and can be overridden at build time.

3.13.2 STM32H7 Programming

ST-Link Flash Tools:

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

OR

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

3.13.3 STM32H7 Testing

To sign the same application image as new version (2), use the sign tool

Python:

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

C Tool:

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

Flash the updated version 2 image: `st-flash write test-app/image_v2_signed.bin 0x08120000`

Upon reboot, wolfboot will elect the best candidate (version 2 in this case) and authenticate the image. If the accepted candidate image resides on BANK B (like in this case), wolfBoot will perform one bank swap before booting.

3.13.4 STM32H7 Debugging

1. Start GDB server

ST-Link: `st-util -p 3333`

ST-Link: `ST-LINK_gdbserver -d -e -r 1 -p 3333`

Mac OS:

```
/Applications/STM32CubeIDE.app/Contents/Eclipse/plugins/\
com.st.stm32cube.ide.mcu.externaltools.stlink-gdb-server.\
macos64_2.0.300.202203231527/tools/bin/\
ST-LINK_gdbserver -d -cp /Applications/STM32CubeIDE.app/\
Contents/Eclipse/plugins/com.st.stm32cube.ide.mcu.\
externaltools.cubeprogrammer.macos64_2.0.200.202202231230/tools/\
bin -e -r 1 -p 3333
```

2. Start GDB Client from wolfBoot root:

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

3.14 NXP LPC54xxx

3.14.1 Build Options

The LPC54xxx build can be obtained by specifying the CPU type and the MCUXpresso SDK path at compile time.

The following configuration has been tested against LPC54606J512BD208:

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

3.14.2 Loading the firmware

Loading with JLink (example: LPC54606J512)

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

```
r
h
```

3.14.3 Debugging with JLink

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

Then, from another console:

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

3.15 Cortex-A53 / Raspberry PI 3 (experimental)

Tested using `https://github.com/raspberrypi/linux` on Ubuntu 20

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

3.15.1 Compiling the kernel

- Get raspberry-pi linux kernel:

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

- Build kernel image:

```
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-
```

- Copy Image and .dtb to the wolfboot directory

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

3.15.2 Testing with qemu-system-aarch64

- Build wolfboot using the example configuration (RSA4096, SHA3)

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

- Sign Linux kernel image

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

- Compose the image

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

- Test boot using qemu

```
qemu-system-aarch64 -M raspi3b -m 1024 -serial stdio -kernel
    wolfboot_linux_raspi.bin -cpu cortex-a53
```

3.15.3 Testing with kernel encryption

The raspberry pi target is used to demonstrate the end-to-end encryption when booting images from RAM. The image is encrypted after being signed. The bootloader uses the same symmetric key to decrypt the image to RAM before performing the validity checks. Here are the steps to enable this feature:

- Build wolfboot using the example configuration (RSA4096, SHA3, ENCRYPT=1)

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

- Create the decrypt key + nonce

```
printf "0123456789abcdef0123456789abcdef0123456789ab" > /tmp/enc_key.der
```

- Sign and encrypt Linux kernel image

```
make keytools
./tools/keytools/sign --aes256 --encrypt /tmp/enc_key.der --rsa4096 --sha3
    Image wolfboot_signing_private_key.der 1
```

- Compose the image

```
tools/bin-assemble/bin-assemble wolfboot_linux_raspi.bin 0x0 wolfboot.bin \
                                0xc0000 Image_v1_signed_and_encrypted.bin
dd if=bcm2710-rpi-3-b.dtb of=wolfboot_linux_raspi.bin bs=1 seek=128K conv=
    notrunc
```

- Test boot using qemu

```
qemu-system-aarch64 -M raspi3b -m 1024 -serial stdio -kernel
    wolfboot_linux_raspi.bin -cpu cortex-a53
```

3.16 Xilinx Zynq UltraScale

Xilinx UltraScale+ ZCU102 (Aarch64)

Build configuration options (`.config`):

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

3.16.1 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-
```

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

3.16.1.2 Signing `tools/keytools/sign --rsa4096 --sha3 /srv/linux-rpi4/vmlinux.bin wolfboot_signing_private_key.der 1`

3.17 Cypress PSoC-6

The Cypress PSoC 62S2 is a dual-core Cortex-M4 & Cortex-M0+ MCU. The secure boot process is managed by the M0+. WolfBoot can be compiled as second stage flash bootloader to manage application verification and firmware updates.

3.17.1 Building

The following configuration has been tested using PSoC 62S2 Wi-Fi BT Pioneer Kit (CY8CKIT-052S2-43012).

3.17.1.1 Target specific requirements wolfBoot uses the following components to access peripherals on the PSoC:

- Cypress Core Library
- PSoC 6 Peripheral Driver Library
- CY8CKIT-062S2-43012 BSP

Cypress provides a customized OpenOCD for programming the flash and debugging.

3.17.2 Clock settings

wolfBoot configures PLL1 to run at 100 MHz and is driving `CLK_FAST`, `CLK_PERI`, and `CLK_SLOW` at that frequency.

3.17.2.1 Build configuration The following configuration has been tested on the PSoC CY8CKIT-62S2-43012:

```
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
```

Note: A reference `.config` can be found in `/config/examples/cypsoc6.config`.

Hardware acceleration is enable by default using psoc6 crypto hw support.

To compile with hardware acceleration disabled, use the option

`PSOC6_CRYPTO=0`

in your wolfBoot configuration.

3.17.2.2 OpenOCD installation

Compile and install the customized OpenOCD.

Use the following configuration file when running `openocd` to connect to the PSoC6 board:

```
### 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
```

3.17.3 Loading the firmware

To upload `factory.bin` to the device with OpenOCD, connect the device, run OpenOCD with the configuration from the previous section, then connect to the local openOCD server running on TCP port 4444 using `telnet localhost 4444`.

From the telnet console, type:

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

When the transfer is finished, you can either close openOCD or start a debugging session.

3.17.4 Debugging

Debugging with OpenOCD:

Use the OpenOCD configuration from the previous sections to run OpenOCD.

From another console, connect using gdb, e.g.:

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

To reset the board to start from the M0+ flash bootloader position (wolfBoot reset handler), use the monitor command sequence below:

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

3.18 Microchip SAME51

SAME51 is a Cortex-M4 microcontroller with a dual-bank, 1MB flash memory divided in blocks of 8KB.

3.18.1 Toolchain

Although it is possible to build wolfBoot with xc32 compilers, we recommend to use gcc for building wolfBoot for best results in terms of footprint and performance, due to some assembly optimizations in wolfCrypt, being available for gcc only. There is no limitation however on the toolchain used to compile the application firmware or RTOS as the two binary files are independent.

3.18.2 Building using gcc/makefile

The following configurations have been tested using ATSAME51J20A development kit.

- `config/examples/same51.config` - example configuration with swap partition (dual-bank disabled)
- `config/examples/same51-dualbank.config` - configuration with two banks (no swap partition)

To build wolfBoot, copy the selected configuration into `.config` and run `make`.

3.18.3 Building using MPLAB IDE

Example projects are provided to build wolfBoot and a test application using MPLAB. These projects are configured to build both stages using xc32-gcc, and have been tested with MpLab IDE v. 6.20.

The example application can be used to update the firmware over USB.

More details about building the example projects can be found in the `IDE/MPLAB` directory in this repository.

3.18.4 Uploading the bootloader and the firmware image

Secure boot and updates have been tested on the SAM E51 Curiosity Nano evaluation board, connecting to a Pro debugger to the D0/D1 pads.

The two firmware images can be uploaded separately using the JLinkExe utility:

```
$ JLinkExe -if swd -speed 1000 -Device ATSAME51J20

J-Link> loadbin wolfboot.bin 0x0

J-Link> loadbin test-app/image_v1_signed.bin 0x8000
```

The above is assuming the default configuration where the BOOT partition starts at address `0x8000`.

3.19 NXP iMX-RT

The NXP iMX-RT10xx family of devices contain a Cortex-M7 with a DCP coprocessor for SHA256 acceleration.

WolfBoot currently supports the NXP RT1040, RT1050, RT1060/1061/1062, and RT1064 devices.

3.19.1 Building wolfBoot

MCUXpresso SDK is required by wolfBoot to access device drivers on this platform. A package can be obtained from the MCUXpresso SDK Builder, by selecting a target and keeping the default choice of components.

- For the RT1040 use `EVKB-IMXRT1040`. See configuration example in `config/examples/imx-rt1040.config`.
- For the RT1050 use `EVKB-IMXRT1050`. See configuration example in `config/examples/imx-rt1050.config`.
- For the RT1060 use `EVKB-IMXRT1060`. See configuration example in `config/examples/imx-rt1060.config`.
- For the RT1064 use `EVK-IMXRT1064`. See configuration example in `config/examples/imx-rt1064.config`.

Set the wolfBoot MCUXPRESSO configuration variable to the path where the SDK package is extracted, then build wolfBoot normally by running make.

wolfBoot support for iMX-RT1060/iMX-RT1050 has been tested using MCUXpresso SDK version 2.14.0. Support for the iMX-RT1064 has been tested using MCUXpresso SDK version 2.13.0

DCP support (hardware acceleration for SHA256 operations) can be enabled by using PKA=1 in the configuration file.

You can also get the SDK and CMSIS bundles using these repositories: * https://github.com/nxp-mcuxpresso/mcux-sdk * https://github.com/nxp-mcuxpresso/CMSIS_5 Use MCUXSDK=1 with this option, since the pack paths are different.

Example:

```
MCUXSDK?=1
MCUXPRESSO?=$(PWD)/../mcux-sdk
MCUXPRESSO_DRIVERS?=$(MCUXPRESSO)/devices/MIMXRT1062
MCUXPRESSO_CMSIS?="$(PWD)/../CMSIS_5/CMSIS"
```

3.19.2 Custom Device Configuration Data (DCD)

On iMX-RT10xx it is possible to load a custom DCD section from an external source file. A customized DCD section should be declared within the .dcd_data section, e.g.:

```
const uint8_t __attribute__((section(".dcd_data"))) dcd_data[] = { /* ... */ };
```

If an external .dcd_data section is provided, the option NXP_CUSTOM_DCD=1 must be added to the configuration.

3.19.3 Building wolfBoot for HAB (High Assurance Boot)

The imx_rt target supports building without a flash configuration, IVT, Boot Data and DCD. This is needed when wanting to use HAB through NXP's *Secure Provisioning Tool* to sign wolfBoot to enable secure boot. To build wolfBoot this way TARGET_IMX_HAB needs to be set to 1 in the configuration file (see config/examples/imx-rt1060 _hab.config for an example). When built with TARGET_IMX_HAB=1 wolfBoot must be written to flash using NXP's *Secure Provisioning Tool*.

3.19.4 Flashing

Firmware can be directly uploaded to the target by copying factory.bin to the virtual USB drive associated to the device, or by loading the image directly into flash using a JTAG/SWD debugger.

The RT1050 EVKB board comes wired to use the 64MB HyperFlash. If you'd like to use QSPI there is a rework that can be performed (see AN12183). The default onboard QSPI 8MB ISSI IS25WP064A (CONFIG_FLASH_IS25WP064A). To use a 64Mbit Winbond W25Q64JV define CONFIG_FLASH_W25Q64JV (16Mbit, 32Mbit, 128Mbit, 256Mbit and 512Mbit versions are also available). These options are also available for the RT1042 and RT1061 target.

If you have updated the MCULink to use JLink then you can connect to the board with JLinkExe using one of the following commands:

```
# HyperFlash
JLinkExe -if swd -speed 5000 -Device "MIMXRT1042xxxxB"
JLinkExe -if swd -speed 5000 -Device "MIMXRT1052XXX6A"
JLinkExe -if swd -speed 5000 -Device "MIMXRT1062XXX6B"
# QSPI
JLinkExe -if swd -speed 5000 -Device
 ↪  "MIMXRT1042xxxxB?BankAddr=0x60000000&Loader=QSPI"
```

```
JLinkExe -if swd -speed 5000 -Device
 ↪  "MIMXRT1052XXX6A?BankAddr=0x60000000&Loader=QSPI"
JLinkExe -if swd -speed 5000 -Device
 ↪  "MIMXRT1062XXX6B?BankAddr=0x60000000&Loader=QSPI"
```

Flash using:

```
loadbin factory.bin 0x60000000
```

3.19.5 Testing Update

First make the update partition, pre-triggered for update:

```
./tools/scripts/prepare_update.sh
```

Run the "loadbin" commands to flash the update:

```
loadbin update.bin 0x60030000
```

Reboot device. Expected output:

```
wolfBoot Test app, version = 1
wolfBoot Test app, version = 8
```

3.19.6 NXP iMX-RT Debugging JTAG / JLINK

```
# Start JLink GDB server for your device
JLinkGDBServer -Device MIMXRT1042xxxxB -speed 5000 -if swd -port 3333
JLinkGDBServer -Device MIMXRT1052xxx6A -speed 5000 -if swd -port 3333
JLinkGDBServer -Device MIMXRT1062xxx6B -speed 5000 -if swd -port 3333

# From wolfBoot directory
arm-none-eabi-gdb
add-symbol-file test-app/image.elf 0x60010100
mon reset init
b main
c
```

3.20 NXP Kinetis

Supports K64 and K82 with crypto hardware acceleration.

3.20.1 Buld options

See `/config/examples/kinetis-k82f.config` for example configuration.

The TARGET is `kinetis`. For LTC PKA support set PKA=.

Set `MCUXPRESSO`, `MCUXPRESSO_CPU`, `MCUXPRESSO_DRIVERS` and `MCUXPRESSO_CMSIS` for MCUXpresso configuration.

3.20.2 Example partitioning for K82

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

3.21 NXP QorIQ P1021 PPC

The NXP QorIQ P1021 is a PPC e500v2 based processor (two cores). This has been tested with a NAND boot source.

3.21.1 Boot ROM NXP P1021

wolfBoot supports loading from external flash using the eLBC FMC (Flash Machine) with NAND.

When each e500 core comes out of reset, its MMU has one 4-Kbyte page defined at `0x0_FFFF_Fnnn`. For NAND boot the first 4KB is loaded to this region with the first offset jump instruction at `0x0_FFFF_FFFC`. The 4KB is mapped to the eLBC FCM buffers.

This device defines the default boot ROM address range to be 8 Mbytes at address `0x0_FF80_0000` to `0x0_FFFF_FFFF`.

These pin determine if the boot ROM will use small or large flash page: * `cfg_rom_loc[0:3]` = 1000 Local bus FCM-8-bit NAND flash small page * `cfg_rom_loc[0:3]` = 1010 Local bus FCM-8-bit NAND flash large page

If the boot sequencer is not enabled, the processor cores exit reset and fetches boot code in default configurations.

A loader must reside in the 4KB page to handle early startup including DDR and then load wolfBoot into DDR for execution.

3.21.2 Design for NXP P1021

1) First stage loader (4KB) resides in first block of NAND flash.
2) Boot ROM loads this into eLBC FCM RAM and maps it to 0xFFFF0000 and sets PC to 0xFFFFFFFC
3) wolfBoot boot assembly configures TLB MMU, LAW, DDR3 and UART (same for all boot stages)
4) First stage loader relocates itself to DDR (to free FCM to allow reading NAND)
5) First stage loader reads entire wolfBoot from NAND flash to DDR and jumps to it
6) wolfBoot loads and parses the header for application partition
7) wolfBoot performs SHA2-384 hash of the application
8) wolfBoot performs a signature verification of the hash
9) wolfBoot loads the application into DDR and jumps to it

3.21.3 First Stage Loader (stage 1) for NXP P1021 PPC

A first stage loader is required to load the wolfBoot image into DDR for execution. This is because only 4KB of code space is available on boot. The stage 1 loader must also copy iteslf from the FCM buffer to DDR (or L2SRAM) to allow using of the eLBC to read NAND blocks.

3.21.3.1 Flash Layout for NXP P1021 PPC (default)

File	NAND offset
stage1/loader_stage1.bin	0x00000000
wolfboot.bin	0x00008000
test-app/image_v1_signed.bin	0x00200000
update	0x01200000
fsl_qe_ucode_1021_10_A.bin	0x01F00000
swap block	0x02200000

3.21.4 Building wolfBoot for NXP P1021 PPC

By default wolfBoot will use `powerpc-linux-gnu-` cross-compiler prefix. These tools can be installed with the Debian package `gcc-powerpc-linux-gnu` (`sudo apt install gcc-powerpc-linux-gnu`).

The `make` creates a `factory_wstage1.bin` image that can be programmed at `0x00000000`, that include the first stage loader, wolfBoot and a signed test application.

To build the first stage load, wolfBoot, sign a custom application and assembly a single factory image use:

```
cp config/examples/nxp-p1021.config .config

# build the key tools
make keytools

make clean
make stage1

# Build wolfBoot (with or without DEBUG)
make DEBUG=1 wolfboot.bin
# OR
make wolfboot.bin

# Sign application
# 1=version (can be any 32-bit value)
./tools/keytools/sign \
    --ecc384 \
    --sha384 \
    test-app/image.bin \
    wolfboot_signing_private_key.der \
    1

./tools/bin-assemble/bin-assemble \
  factory.bin \
    0x0        hal/nxp_p1021_stage1.bin \
    0x8000     wolfboot.bin \
    0x200000   test-app/image.bin \
    0x01F00000 fsl_qe_ucode_1021_10_A.bin
```

3.21.5 Debugging NXP P1021 PPC

Use `V=1` to show verbose output for build steps. Use `DEBUG=1` to enable debug symbols.

The first stage loader must fit into 4KB. To build this in release and assemble a debug version of wolfBoot use the following steps:

```
make clean
make stage1
make DEBUG=1 wolfboot.bin
make DEBUG=1 test-app/image_v1_signed.bin
make factory_wstage1.bin
```

3.22 NXP QorIQ T1024 PPC

The NXP QorIQ T1024 is a two core 64-bit PPC e5500 based processor at 1400MHz. Each core has 256KB L2 cache.

Board: T1024RDB Board rev: 0x3031 CPLD ver: 0x42

T1024E, Version: 1.0, (0x8548_0010) e5500, Version: 2.1, (0x8024_1021)

Reset Configuration Word (RCW): 00000000: 0810000e 00000000 00000000 00000000 00000010: 2d800003 40408812 fc027000 21000000 00000020: 00000000 00000000 60000000 00036800 00000030: 00000100 484a5808 00000000 00000006

Flash is NOR on IFC CS0 (0x0_EC00_0000) 64MB (default).

Default NOR Flash Memory Layout (64MB) (128KB block, 1K page)

Description	Address	Size
RCW	0xEC000000	0x00020000 (128 KB)
Free	0xEC020000	0x000D0000 (832 KB)
Swap Sector	0xEC0F0000	0x00010000 (64 KB)
Free	0xEC100000	0x00700000 (7 MB)
FDT (Primary)	0xEC800000	0x00020000 (128 KB)
FDT (Update)	0xEC820000	0x00020000 (128 KB)
Free	0xEC840000	0x008A0000 (8MB)
Ethenet Config	0xED0E0000	0x00000400 (1 KB)
Free	0xED100000	0x00F00000 (15 MB)
Application (OS)	0xEE000000	0x00F00000 (15 MB)
Update (OS)	0xEEF00000	0x00F00000 (15 MB)
QUICC	0xEFE00000	0x00100000 (1 MB)
DPAA (FMAN)	0xEFF00000	0x00020000 (128 KB)
wolfBoot	0xEFF40000	0x000BC000 (752 KB)
wolfBoot Stage 1	0xEFFFC000	0x00004000 (16 KB)

QE: uploading microcode ‘Microcode for T1024 r1.0’ version 0.0.1

DDR4 2GB

3.22.1 Building wolfBoot for NXP T1024 PPC

By default wolfBoot will use `powerpc-linux-gnu-` cross-compiler prefix. These tools can be installed with the Debian package `gcc-powerpc-linux-gnu` (`sudo apt install gcc-powerpc-linux-gnu`).

The `make` creates a `factory_stage1.bin` image that can be programmed at `0xEC000000`

```
cp ./config/examples/nxp-t1024.config .config
make clean
make keytools
make
```

Or each `make` component can be manually built using:

```
make stage1
make wolfboot.elf
make test-app/image_v1_signed.bin
```

If getting errors with keystore then you can reset things using `make distclean`.

3.22.2 Signing Custom application

```
./tools/keytools/sign --ecc384 --sha384 custom.elf
   wolfboot_signing_private_key.der 1
```

3.22.3 Assembly of custom firmware image

```
./tools/bin-assemble/bin-assemble factory_custom.bin \
    0xEC000000 RCW_CTS.bin \
    0xEC020000 custom.dtb \
    0xEE000000 custom_v1_signed.bin \
    0xEFE00000 iram_Type_A_T1024_r1.0.bin \
    0xEFF00000 fsl_fman_ucode_t1024_r1.0_108_4_5.bin \
    0xEFF40000 wolfboot.bin \
    0xEFFFC000 stage1/loader_stage1.bin
```

Flash factory_custom.bin to NOR base 0xEC00_0000

3.23 NXP QorIQ T2080 PPC

The NXP QorIQ T2080 is a PPC e6500 based processor (four cores). Support has been tested with the NAII 68PPC2.

Example configurations for this target are provided in: * NXP T2080: `/config/examples/nxp-t2080.config`. * NAII 68PPC2: `/config/examples/nxp-t2080-68ppc2.config`.

3.23.1 Design NXP T2080 PPC

The QorIQ requires a Reset Configuration Word (RCW) to define the boot parameters, which resides at the start of the flash (0xE8000000).

The flash boot entry point is `0xEFFFFFFC`, which is an offset jump to wolfBoot initialization boot code. Initially the PowerPC core enables only a 4KB region to execute from. The initialization code (`src/boot_ppc_start.S`) sets the required CCSR and TLB for memory addressing and jumps to wolfBoot `main()`.

RM 4.3.3 Boot Space Translation

“When each core comes out of reset, its MMU has one 4 KB page defined at 0x0_FFFF_Fnnn. Each core begins execution with the instruction at effective address 0x0_FFFF_FFFC. To get this instruction, the core’s first instruction fetch is a burst read of boot code from effective address 0x0_FFFF_FFC0.”

3.23.2 Building wolfBoot for NXP T2080 PPC

By default wolfBoot will use `powerpc-linux-gnu-` cross-compiler prefix. These tools can be installed with the Debian package `gcc-powerpc-linux-gnu` (`sudo apt install gcc-powerpc-linux-gnu`).

The `make` creates a `factory.bin` image that can be programmed at `0xE8080000`

```
cp ./config/examples/nxp-t2080-68ppc2.config .config
make clean
make keytools
make
```

Or each `make` component can be manually built using:

```
make wolfboot.elf
make test-app/image_v1_signed.bin
```

If getting errors with keystore then you can reset things using `make distclean`.

3.23.2.1 Building QorIQ Linux SDK fsl-toolchain

To use the NXP cross-compiler:

Find "QorIQ Linux SDK v2.0 PPCE6500 IMAGE.iso" on nxp.com and extract the "fsl-toolchain". Then run the script to install to default location `/opt/fsl-qoriq/2.0/`.

Then add the following lines to your `.config`:

```
CROSS_COMPILE?=/opt/fsl-qoriq/2.0/sysroots/x86_64-fslsdk-linux/usr/bin/powerpc
    -fsl-linux/powerpc-fsl-linux-
CROSS_COMPILE_PATH=/opt/fsl-qoriq/2.0/sysroots/ppce6500-fsl-linux/usr
```

3.23.3 Programming NXP T2080 PPC

NOR Flash Region: `0xE8000000 - 0xEFFFFFFF` (128 MB)

Flash Layout (with files):

Description	File	Address
Reset Configuration Word (RCW)	`68PPC2_RCW_v0p7.bin`	`0xE8000000`
Frame Manager Microcode	`fsl_fman_ucode_t2080_r1.0.bin`	`0xE8020000`
Signed Application	`test-app/image_v1_signed.bin`	`0xE8080000`
wolfBoot	`wolfboot.bin`	`0xEFF40000`
Boot Entry Point [1]		`0xEFFFFFFC`

Or program the `factory.bin` to `0xE8080000`

Example Boot Debug Output:

```
wolfBoot Init
Part: Active 0, Address E8080000
Image size 1028
Firmware Valid
Loading 1028 bytes to RAM at 19000
Failed parsing DTB to load.
Booting at 19000
Test App

0x00000001
0x00000002
0x00000003
0x00000004
0x00000005
0x00000006
0x00000007
...
```

3.23.3.1 Flash Programming with Lauterbach

See these TRACE32 demo script files: * `./demo/powerpc64bit/hardware/qoriq_t2/t2080rdb/flash_cfi.cmm` * `./demo/powerpc64bit/hardware/qoriq_t2/t2080rdb/demo_set_rcw.cmm`

[1] with offset jump to init code

```
DO flash_cfi.cmm

FLASH.ReProgram 0xEFF40000--0xEFFFFFFF /Erase
Data.LOAD.binary wolfboot.bin 0xEFF40000
FLASH.ReProgram.off

Data.LOAD.binary wolfboot.bin 0xEFF40000 /Verify
```

Note: To disable the flash protection bits use:

```
;enter Non-volatile protection mode (C0h)
Data.Set 0xE8000000+0xAAA %W 0xAAAA
Data.Set 0xE8000000+0x554 %W 0x5555
Data.Set 0xE8000000+0xAAA %W 0xC0C0
;clear all protection bit (80h/30h)
Data.Set 0xE8000000 %W 0x8080
Data.Set 0xE8000000 %W 0x3030
;exit Non-volatile protection mode (90h/00h)
Data.Set 0xE8000000 %W 0x9090
Data.Set 0xE8000000 %W 0x0000
```

3.23.3.2 Flash Programming with CodeWarrior TAP In CodeWarrior use the `Flash Programmer` tool (see under Commander View -> Miscellaneous) * Connection: “CodeWarrior TAP Connection” * Flash Configuration File: “T2080QDS_NOR_FLASH.xml” * Unprotect flash memory before erase: Check * Choose file and set offset address.

```
tftp 1000000 wolfboot.bin
protect off eff40000 +C0000
erase eff40000 +C0000
cp.b 1000000 eff40000 C0000
protect on eff40000 +C0000
cmp.b 1000000 eff40000 C0000
```

3.23.4 Debugging NXP T2080 PPC

3.23.3.3 Flash Programming from U-Boot

```
SYStem.RESet
SYStem.BdmClock 15.MHz
SYStem.CPU T2080
SYStem.DETECT CPU
CORE.ASSIGN 1.
SYStem.Option.FREEZE OFF
SYStem.Up

Data.LOAD.Elf wolfboot.elf /NoCODE

Break main
List.auto
Go
```

If cross-compiling on a different machine you can use the /StripPART option:

```
sYmbol.SourcePATH.SetBaseDir ~/wolfBoot
Data.LOAD.Elf wolfboot.elf /NoCODE /StripPART "/home/username/wolfBoot/"
```

3.23.4.2 CodeWarrior TAP This is an example for debugging the T2080 with CodeWarrior TAP, however we were not successful using it. The Lauterbach is what we ended up using to debug.

Start GDB Proxy:

- Linux: /opt/Freescale/CW_PA_v10.5.1/PA/ccs/bin/gdbproxy
- Windows: C:\Freescale\CW_PA_v10.5.1\PA\ccs\bin\gdbproxy.exe

```
set logging on
set debug remote 10
set remotetimeout 20
set tdesc filename ../xml/e6500.xml
set remote hardware-breakpoint-limit 10
target remote t2080-tap-01:2345
mon probe fpga
mon ccs_host t2080-tap-01
mon ccs_path /opt/Freescale/CodeWarrior_PA_10.5.1/PA/ccs/bin/ccs
mon jtag_speed 12500
mon jtag_chain t4amp
mon connect
Remote debugging using t2080-tap-01:2345
0x00000000 in ?? ()
(gdb) mon get_probe_status
Connected to gdbserver t2080-tap-01:2345

Executing Initialization File: /opt/Freescale/CodeWarrior_PA_10.5.1/PA/
   PA_Support/Initialization_Files/QorIQ_T2/68PPC2_init_sram.tcl
thread break: Stopped, 0x0, 0x0, cpuPowerPCBig,  Connected (state, tid, pid,
   cpu, target)
```

3.24 NXP MCXA153

NXP MCXA153 is a Cortex-M33 microcontroller running at 96MHz. The support has been tested using FRDM-MCXA153 with the onboard MCU-Link configured in JLink mode.

This requires the MCXA SDK from the NXP MCUXpresso SDK Builder. We tested using SDK_2.14.2_MCXA153 and placed into ../NXP/MCXA153 by default (see .config or set with MCUXPRESSO). MCUXpresso SDK Builder

3.24.1 MCX A: Configuring and compiling

Copy the example configuration file and build with make:

```
cp config/examples/mcxa.config .config`
make
```

3.24.2 MCX A: Loading the firmware

The NXP Freedom MCX A board debugger comes loaded with MCU Link, but it can be updated to JLink. See https://docs.nxp.com/bundle/UM12012/page/topics/Updating_MCU_Link_firmware.html

Use JLinkExe tool to upload the initial firmware: `JLinkExe -if swd -Device MCXA153`

At the Jlink prompt, type:

```
loadbin factory.bin 0
Downloading file [factory.bin]...
J-Link: Flash download: Bank 0 @ 0x00000000: Skipped. Contents already match
O.K.
```

Reset or power cycle board.

Once wolfBoot has performed validation of the partition and booted the D15 Green LED on P3_13 will illuminate.

3.24.3 MCX A: Testing firmware update

1) Sign the test-app with version 2:

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

2) Create a bin footer with wolfBoot trailer "BOOT" and "p" (ASCII for 0x70 == IMG_STATE_UPDATING):

```
echo -n "pBOOT" > trigger_magic.bin
```

3) Assembly new factory update.bin:

```
./tools/bin-assemble/bin-assemble \
  update.bin \
    0x0    test-app/image_v2_signed.bin \
    0xAFFB trigger_magic.bin
```

4) Flash update.bin to 0x13000 (`loadbin update.bin 0x13000`). The D15 RGB LED Blue P3_0 will show if version is > 1.

Note: For alternate larger scheme flash `update.bin` to `0x14000` and place trigger_magic.bin at `0x9FFB`.

3.24.4 MCX A: Debugging

Debugging with JLink:

Note: We include a `.gdbinit` in the wolfBoot root that loads the wolfboot and test-app elf files.

In one terminal: `JLinkGDBServer -if swd -Device MCXA153 -port 3333`

In another terminal use gdb:

```
b main
mon reset
c
```

3.25 TI Hercules TMS570LC435

See `/config/examples/ti-tms570lc435.config` for example configuration.

3.26 Nordic nRF52840

We have full Nordic nRF5280 examples for Contiki and RIOT-OS in our wolfBoot-examples repo

Examples for nRF52: * 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

Example of flash memory layout and configuration on the nRF52:

- 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
```

3.27 Simulated

You can create a simulated target that uses files to mimic an internal and optionally an external flash. The build will produce an executable ELF file `wolfBoot.elf`. You can provide another executable ELF as firmware image and it will be executed. The command-line arguments of `wolfBoot.elf` are forwarded to the application. The example application `test-app\app_sim.c` uses the arguments to interact with `libwolfboot.c` and automate functional testing. You can find an example configuration in `config/examples/sim.config`.

An example of using the `test-app/sim.c` to test firmware update:

```
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
```

Note: This also works on Mac OS, but `objcopy` does not exist. Install with `brew install binutils` and make using `OBJCOPY=/usr/local/Cellar//binutils/2.41/bin/objcopy make`.

3.28 Renesas RX65N

Tested on the: * RX65N-2MB-Starter-Kit-Plus * RX65N Target Board (RTK5RX65N0C00000BR) (includes onboard E2 Lite emulator)

Both include an E2 Lite Emulator.

3.28.1 Renesas Console

Console output is supported with DEBUG_UART=1.

RSK+: This board includes a USB to Serial port that uses SCI8 and PJ1/PJ2. This is the wolfBoot HAL default for RX65N.

RX65N target board:

Can route UART Serial output to PC3 via PMOD1-IO0 at Pin 9. This requires an external TTL UART to USB adapter. You will need to set CFLAGS_EXTRA+="-DDEBUG_UART_SCI=3" in .config. In the renesas-rx.c uart_init these port mode and port function select settings are needed:

```
/* Configure PC3/PC2 for UART */
PORT_PMR(0xC) |= ((1 << 2) | (1 << 3));
/* SCI Function Select = 0xA (UART) */
MPC_PFS(0xC2) = 0xA; /* PC2-RXD5 */
MPC_PFS(0xC3) = 0xA; /* PC3-TXD5 */
```

Example Boot Output (with DEBUG_UART=1):

```
wolfBoot HAL Init
Boot partition: 0xFFE00000
Image size 25932

| ------------------------------------------------------------------ |
| Renesas RX User Application in BOOT partition started by wolfBoot  |
| ------------------------------------------------------------------ |

wolfBoot HAL Init

=== Boot Partition[ffe00000] ===
Magic:    WOLF
Version:  01
Status:   ff (New)
Trailer Magic: ˇˇˇˇ

=== Update Partition[ffef0000] ===
Magic: ˇˇˇˇ
Version:  00
Status:   ff (New)
Trailer Magic: ˇˇˇˇ

Current Firmware Version: 1
Hit any key to call wolfBoot_success the firmware.
```

3.28.2 Renesas Flash Layout

Default Onboard Flash Memory Layout (2MB) (32KB sector):

Description	Address	Size
OFSM Option Mem	0xFE7F5D00	0x00000080 (128 B)
Application	0xFFE00000	0x000F0000 (960 KB)
Update	0xFFEF0000	0x000F0000 (960 KB)
Swap	0xFFFE0000	0x00010000 (64 KB)

Description	Address	Size
wolfBoot	0xFFFF0000	0x00010000 (64 KB)

3.28.3 Renesas Data Endianess

To switch RX parts to big endian data use:

```
# Big Endian
rfp-cli -if fine -t e2l -device RX65x -auth id FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 ↪  -write32 0xFE7F5D00 0xFFFFFFF8
OR
# Little Endian
rfp-cli -if fine -t e2l -device RX65x -auth id FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 ↪  -write32 0xFE7F5D00 0xFFFFFFFF
```

3.28.4 Building Renesas RX65N

Building RX wolfBoot requires the RX-ELF compiler. Please Download and install the Renesas RX GCC toolchain: https://llvm-gcc-renesas.com/rx-download-toolchains/

Default installation path (Linux): `~/toolchains/gcc_8.3.0.202311_rx_elf` Default installation path (Windows): `C:\ProgramData\GCC for Renesas RX 8.3.0.202305-GNURX-ELF\rx-elf\rx-elf`

Configuration: Use `./config/examples/renesas-rx65n.config` as a starting point by copying it to the wolfBoot root as `.config`.

```
cp ./config/examples/renesas-rx65n.config .config
make
```

With RX GCC path or or custom cross compiler directly: `make CROSS_COMPILE="~/toolchains/gcc_8.3.0.202311_rx_elf/rx-elf-"` OR `make RX_GCC_PATH="~/toolchains/gcc_8.3.0.202311_rx_elf"`

TSIP: To enable TSIP use `make PKA=1`. See Appendix J for details.

3.28.5 Flashing Renesas RX65N

Download the Renesas Flashing Tool Download the Renesas E2 Lite Linux Driver

Default Flash ID Code: FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Flash Using:

```
rfp-cli -if fine -t e2l -device RX65x -auto -auth id
    FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF \
     -bin FFFF0000 wolfboot.bin \
     -bin FFE00000 test-app/image_v1_signed.bin \
     -run
```

Note: Endianess: if using big endian add `-endian big`

Note: Linux Install E2 Lite USB Driver:

```
sudo cp 99-renesas-emu.rules /etc/udev/rules.d/
sudo udevadm control --reload-rules
```

3.28.6 Debugging Renesas RX65N

Create a new “Renesas Debug” project. Choose the “E2 Lite” emulator and the built `wolfboot.elf`. After project is created open the “Debug Configuration” and change the debugger interface from “JTAG” to “FINE”. Run debug and it will stop in the “reset” code in `boot_renesas_start.S`. If using Big Endian change endianess mode in “Debugger -> Debug Tool Settings -> Memory Endian -> Big Endian”.

3.29 Renesas RX72N

Tested on the RX72N ENVISION KIT (HMI development kit for IoT systems). This includes an onboard E2 Lite emulator.

The Renesas RX72N is supported either natively with “make” or through e2Studio. If using e2Studio see `/IDE/Renesas/e2studio/RX72N/Readme.md`.

Default UART Serial on SCI2 at P12-RXD2 P13-TXD2. Use USB on CN8 to attach a Virtual USB COM port. This feaure is enabled with `DEBUG_UART=1`.

Example Boot Output (with DEBUG_UART=1):

```
wolfBoot HAL Init
Boot partition: 0xFFC00000
Image size 27772

| ------------------------------------------------------------------ |
| Renesas RX User Application in BOOT partition started by wolfBoot  |
| ------------------------------------------------------------------ |

wolfBoot HAL Init

=== Boot Partition[ffc00000] ===
Magic:    WOLF
Version:  01
Status:   ff (New)
Trailer Magic: ˇˇˇˇ

=== Update Partition[ffdf0000] ===
Magic: ˇˇˇˇ
Version:  00
Status:   ff (New)
Trailer Magic: ˇˇˇˇ

Current Firmware Version: 1
Hit any key to call wolfBoot_success the firmware.
```

Default Onboard Flash Memory Layout (4MB) (32KB sector):

Description	Address	Size
OFSM Option Mem	0xFE7F5D00	0x00000080 (128 B)
Application	0xFFC00000	0x001F0000 (1984 KB)
Update	0xFFDF0000	0x001F0000 (1984 KB)
Swap	0xFFFE0000	0x00010000 (64 KB)
wolfBoot	0xFFFF0000	0x00010000 (64 KB)

To switch RX parts to big endian data use:

```
# Big Endian
rfp-cli -if fine -t e2l -device RX72x -auth id FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 ↪  -write32 0xFE7F5D00 0xFFFFFFF8
OR
# Little Endian
rfp-cli -if fine -t e2l -device RX72x -auth id FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 ↪  -write32 0xFE7F5D00 0xFFFFFFFF
```

3.29.1 Building Renesas RX72N

Building RX wolfBoot requires the RX-ELF compiler. Please Download and install the Renesas RX GCC toolchain: https://llvm-gcc-renesas.com/rx-download-toolchains/

Default installation path (Linux): `~/toolchains/gcc_8.3.0.202311_rx_elf` Default installation path (Windows): `C:\ProgramData\GCC for Renesas RX 8.3.0.202305-GNURX-ELF\rx-elf\rx-elf`

Configuration: Use `./config/examples/renesas-rx72n.config` as a starting point by copying it to the wolfBoot root as `.config`.

```
cp ./config/examples/renesas-rx72n.config .config
make
```

With RX GCC path or or custom cross compiler directly:

```
make CROSS_COMPILE="~/toolchains/gcc_8.3.0.202311_rx_elf/bin/rx-elf-"
```

OR

```
make RX_GCC_PATH="~/toolchains/gcc_8.3.0.202311_rx_elf"
```

TSIP: To enable TSIP use `make PKA=1`. See Appendix J for details.

3.29.2 Flashing Renesas RX72N

Download the Renesas Flashing Tool Download the Renesas E2 Lite Linux Driver

Default Flash ID Code: FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Flash Using:

```
rfp-cli -if fine -t e2l -device RX72x -auto -auth id
   FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF \
    -bin FFFF0000 wolfboot.bin \
    -bin FFC00000 test-app/image_v1_signed.bin \
    -run
```

Note: Endianess: if using big endian add `-endian big`

Note: Linux Install E2 Lite USB Driver:

```
sudo cp 99-renesas-emu.rules /etc/udev/rules.d/
sudo udevadm control --reload-rules
```

3.30 Renesas RA6M4

This example for Renesas RA6M4 demonstrates a simple secure firmware update by wolfBoot. A sample application v1 is securely updated to v2. Both versions behave the same except displaying its version of v1 or v2. They are compiled by e2Studio and running on the target board.

In this demo, you may download two versions of application binary file by Renesas Flash Programmer. You can download and execute wolfBoot by e2Studio debugger. Use a USB connection between PC and the board for the debugger and flash programmer.

Flash Allocation:

```
+--------------------------+------------------------+-----+
| B |H|                    |H|                      |     |
| o |e|   Primary          |e|   Update             |Swap |
| o |a|   Partition        |a|   Partition          |Sect |
| t |d|                    |d|                      |     |
+--------------------------+------------------------+-----+
0x00000000: wolfBoot
0x00010000: Primary partition (Header)
0x00010200: Primary partition (Application image)
0x00080000: Update  partition (Header)
0x00080200: Update  partition (Application image)
0x000F0000: Swap sector
```

Detailed steps can be found at `/IDE/Renesas/e2studio/RA6M4/Readme.md`.

3.31 Renesas RZN2L

This example demonstrates simple secure firmware boot from external flash by wolfBoot. A sample application v1 is securely loaded into internal RAM if there is not higher version in update region. A sample application v2 will be loaded when it is in update region.Both versions behave the same except blinking LED Red(v1) or Yellow(v2). They are compiled by e2Studio and running on the target board.

The example uses SPI boot mode with external flash on the evaluation board. On this boot mode, the loader program, which is wolfBoot, is copied to the internal RAM(B-TCM). wolfBoot copies the application program from external flash memory to RAM(System RAM). As final step of wolfBoot the entry point of the copied application program is called if its integrity and authenticity are OK.

Detailed steps can be found at `/IDE/Renesas/e2studio/RA6M4/Readme.md`.

3.32 Qemu x86-64 UEFI

The simplest option to compile wolfBoot as a bootloader for x86-64bit machines is the UEFI mode. This mechanism requires an UEFI bios, which stages wolfBoot by running the binary as an EFI application.

The following instructions describe the procedure to configure wolfBoot as EFI application and run it on qemu using tianocore as main firmware. A GNU/Linux system built via buildroot is then authenticated and staged by wolfBoot.

3.32.1 Prerequisites:

- qemu-system-x86_64
- [GNU-EFI] (https://sourceforge.net/projects/gnu-efi/)
- Open Virtual Machine firmware bios images (OVMF) by Tianocore

On a debian-like system it is sufficient to install the packages as follows:

```
# 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
```

3.32.2 Configuration

An example configuration is provided in `config/examples/x86_64_efi.config`

3.32.3 Building and running on qemu

The bootloader and the initialization script `startup.nsh` for execution in the EFI environment are stored in a loopback FAT partition.

The script `tools/efi/prepare_uefi_partition.sh` creates a new empty FAT loopback partitions and adds `startup.nsh`.

A kernel with an embedded rootfs partition can be now created and added to the image, via the script `tools/efi/compile_efi_linux.sh`. The script actually adds two instances of the target systems: `kernel.img` and `update.img`, both signed for authentication, and tagged with version 1 and 2 respectively.

Compiling with `make` will produce the bootloader image in `wolfboot.efi`.

The script `tools/efi/run_efi.sh` will add `wolfboot.efi` to the bootloader loopback partition, and run the system on qemu. If both kernel images are present and valid, wolfBoot will choose the image with the higher version number, so `update.img` will be staged as it's tagged with version 2.

The sequence is summarized below:

```
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
```

You can `Ctrl-C` or login as `root` and power off qemu with `poweroff`

3.33 Intel x86_64 with Intel FSP support

This setup is more complex than the UEFI approach described earlier, but allows for complete control of the machine since the very first stage after poweron.

In other words, wolfBoot can run as a secure replacement of the system BIOS, thanks to the integration with the Intel Firmware Support Package (FSP). FSP provides services for target-specific initial configuration (memory and silicon initialization, power management, etc.). These services are designed to be accessed and invoked by the bootloader.

If wolfBoot is compiled with FSP support, it invokes the necessary machine-dependent binary code, which that can be obtained from the chip manufacturer.

The following variables must be set in your `.config` file when using this feature:

- `ARCH` = x86_64
- `TARGET` = A useful name for the target you want to support. You can refer to x86_fsp_qemu or kontron_vx3060_s2 for reference
- `FSP_T_BASE`: the base address where the FSP-T binary blob will be loaded.
- `FSP_M_BASE`: the base address where the FSP-M binary blob will be loaded.
- `FSP_S_BASE`: the base address where the FSP-S binary blob will be loaded.
- `FSP_T_BIN`: path to the FSP-T binary blob
- `FSP_M_BIN`: path to the FSP-M binary blob
- `FSP_S_BIN`: path to the FSP-S binary blob
- `WOLFBOOT_ORIGIN`: the start address of wolfBoot inside the flash (flash is mapped so that it ends at the 4GB boundary)
- `BOOTLOADER_PARTITION_SIZE`: the size of the partition that stores wolfBoot in the flash
- `WOLFBOOT_LOAD_BASE`: the address where wolfboot will be loaded in RAM after the first initialization phase

While Intel FSP aims to abstract away specific machine details, you still need some machine-specific code. Current supported targets are QEMU and the TigerLake based Kontron VX3060-S2 board. Refer to the Intel Integration Guide of the selected silicon for more information.

Note:

- This feature requires `NASM` to be installed on the machine building wolfBoot.

3.33.1 Running on 64-bit QEMU

Two example configuration files are available: `config/examples/x86_fsp_qemu.config` and `config/examples/x86_fsp_qemu_seal.config`. Both will try to load a 64bit ELF/Multiboot2 payload from the emulated sata drive. The second one is an example of configuration that also do measure boot and seal/unseal secrets using a TPM.

A test ELF/Multiboot2 image is provided as well. To test `config/examples/x86_fsp_qemu.config` use the following steps:

```
# Copy the example configuration for this target
cp config/examples/x86_fsp_qemu.config .config

# Create necessary Intel FSP binaries from edk2 repo
./tools/scripts/x86_fsp/qemu/qemu_build_fsp.sh

# build wolfboot
make

# make test-app
make test-app/image.elf
```

```
# make_hd.sh sign the image, creates a file-based hard disk image with GPT
    table and raw partitions and then copies the signed images into the
    partitions.
IMAGE=test-app/image.elf tools/scripts/x86_fsp/qemu/make_hd.sh

# run wolfBoot + test-image
./tools/scripts/x86_fsp/qemu/qemu.sh


Cache-as-RAM initialized
FSP-T:0.0.10 build 0
FSP-M:0.0.10 build 0
no microcode for QEMU target
calling FspMemInit...

============= FSP Spec v2.0 Header Revision v3 ($QEMFSP$ v0.0.10.0)
    =============
Fsp BootFirmwareVolumeBase - 0xFFE30000
Fsp BootFirmwareVolumeSize - 0x22000
Fsp TemporaryRamBase       - 0x4
Fsp TemporaryRamSize       - 0x50000
Fsp PeiTemporaryRamBase    - 0x4
Fsp PeiTemporaryRamSize    - 0x34000
Fsp StackBase              - 0x34004
Fsp StackSize              - 0x1C000
Register PPI Notify: DCD0BE23-9586-40F4-B643-06522CED4EDE
Install PPI: 8C8CE578-8A3D-4F1C-9935-896185C32DD3
Install PPI: 5473C07A-3DCB-4DCA-BD6F-1E9689E7349A
The 0th FV start address is 0x000FFE30000, size is 0x00022000, handle is 0
    xFFE30000
Register PPI Notify: 49EDB1C1-BF21-4761-BB12-EB0031AABB39
Register PPI Notify: EA7CA24B-DED5-4DAD-A389-BF827E8F9B38
Install PPI: B9E0ABFE-5979-4914-977F-6DEE78C278A6
Install PPI: A1EEAB87-C859-479D-89B5-1461F4061A3E
Install PPI: DBE23AA9-A345-4B97-85B6-B226F1617389
DiscoverPeimsAndOrderWithApriori(): Found 0x2 PEI FFS files in the 0th FV
Loading PEIM 9B3ADA4F-AE56-4C24-8DEA-F03B7558AE50
Loading PEIM at 0x000FFE3D8C8 EntryPoint=0x000FFE3EC4C PcdPeim.efi
Install PPI: 06E81C58-4AD7-44BC-8390-F10265F72480
Install PPI: 01F34D25-4DE2-23AD-3FF3-36353FF323F1
Install PPI: 4D8B155B-C059-4C8F-8926-06FD4331DB8A
Install PPI: A60C6B59-E459-425D-9C69-0BCC9CB27D81
Register PPI Notify: 605EA650-C65C-42E1-BA80-91A52AB618C6
Loading PEIM 9E1CC850-6731-4848-8752-6673C7005EEE
Loading PEIM at 0x000FFE3F114 EntryPoint=0x000FFE411DF FspmInit.efi
FspmInitPoint() - Begin
BootMode : 0x0
Install PPI: 7408D748-FC8C-4EE6-9288-C4BEC092A410
Register PPI Notify: F894643D-C449-42D1-8EA8-85BDD8C65BDE
PeiInstallPeiMemory MemoryBegin 0x3EF00000, MemoryLength 0x100000
FspmInitPoint() - End
Temp Stack : BaseAddress=0x34004 Length=0x1C000
Temp Heap  : BaseAddress=0x4 Length=0x34000
Total temporary memory:    327680 bytes.
```

```
  temporary memory stack ever used:       3360 bytes.
  temporary memory heap used for HobList: 2104 bytes.
  temporary memory heap occupied by memory pages: 0 bytes.
Old Stack size 114688, New stack size 131072
Stack Hob: BaseAddress=0x3EF00000 Length=0x20000
Heap Offset = 0x3EF1FFFC Stack Offset = 0x3EECFFFC
Loading PEIM 52C05B14-0B98-496C-BC3B-04B50211D680
Loading PEIM at 0x0003EFF5150 EntryPoint=0x0003EFFBBC6 PeiCore.efi
Reinstall PPI: 8C8CE578-8A3D-4F1C-9935-896185C32DD3
Reinstall PPI: 5473C07A-3DCB-4DCA-BD6F-1E9689E7349A
Reinstall PPI: B9E0ABFE-5979-4914-977F-6DEE78C278A6
Install PPI: F894643D-C449-42D1-8EA8-85BDD8C65BDE
Notify: PPI Guid: F894643D-C449-42D1-8EA8-85BDD8C65BDE, Peim notify entry
   point: FFE40AB2
Memory Discovered Notify invoked ...
FSP TOLM = 0x3F000000
Migrate FSP-M UPD from 7F540 to 3EFF4000
FspMemoryInitApi() - [Status: 0x00000000] - End
success
top reserved 0_3EF00000h
mem: [ 0x3EEF0000, 0x3EF00000 ] - stack (0x10000)
mem: [ 0x3EEEFFF4, 0x3EEF0000 ] - stage2 parameter (0xC)
hoblist@0x3EF20000
mem: [ 0x3EEE8000, 0x3EEEFFF4 ] - page tables (0x7FF4)
page table @ 0x3EEE8000 [length: 7000]
mem: [ 0x3EEE7FF8, 0x3EEE8000 ] - stage2 ptr holder (0x8)
TOLUM: 0x3EEE7FF8
TempRamExitApi() - Begin
Memory Discovered Notify completed ...
TempRamExitApi() - [Status: 0x00000000] - End
mem: [ 0x800000, 0x800084 ] - stage1 .data (0x84)
mem: [ 0x8000A0, 0x801A80 ] - stage1 .bss (0x19E0)
mem: [ 0xFED5E00, 0xFEEAF00 ] - FSPS (0x15100)
Authenticating FSP_S at FED5E00...
Image size 86016
verify_payload: image open successfully.
verify_payload: integrity OK. Checking signature.
FSP_S: verified OK.
FSP-S:0.0.10 build 0
call silicon...
SiliconInitApi() - Begin
Install PPI: 49EDB1C1-BF21-4761-BB12-EB0031AABB39
Notify: PPI Guid: 49EDB1C1-BF21-4761-BB12-EB0031AABB39, Peim notify entry
   point: FFE370A2
The 1th FV start address is 0x0000FED5F00, size is 0x00015000, handle is 0
   xFED5F00
DiscoverPeimsAndOrderWithApriori(): Found 0x4 PEI FFS files in the 1th FV
Loading PEIM 86D70125-BAA3-4296-A62F-602BEBBB9081
Loading PEIM at 0x0003EFEE150 EntryPoint=0x0003EFF15B9 DxeIpl.efi
Install PPI: 1A36E4E7-FAB6-476A-8E75-695A0576FDD7
Install PPI: 0AE8CE5D-E448-4437-A8D7-EBF5F194F731
Loading PEIM 131B73AC-C033-4DE1-8794-6DAB08E731CF
Loading PEIM at 0x0003EFE6000 EntryPoint=0x0003EFE702B FspsInit.efi
FspInitEntryPoint() - start
```

```
Register PPI Notify: 605EA650-C65C-42E1-BA80-91A52AB618C6
Register PPI Notify: BD44F629-EAE7-4198-87F1-39FAB0FD717E
Register PPI Notify: 7CE88FB3-4BD7-4679-87A8-A8D8DEE50D2B
Register PPI Notify: 6ECD1463-4A4A-461B-AF5F-5A33E3B2162B
Register PPI Notify: 30CFE3E7-3DE1-4586-BE20-DEABA1B3B793
FspInitEntryPoint() - end
Loading PEIM BA37F2C5-B0F3-4A95-B55F-F25F4F6F8452
Loading PEIM at 0x0003EFDC000 EntryPoint=0x0003EFDDA67 QemuVideo.efi
NO valid graphics config data found!
Loading PEIM 29CBB005-C972-49F3-960F-292E2202CECD
Loading PEIM at 0x0003EFD2000 EntryPoint=0x0003EFD3265 FspNotifyPhasePeim.efi
The entry of FspNotificationPeim
Reinstall PPI: 0AE8CE5D-E448-4437-A8D7-EBF5F194F731
DXE IPL Entry
FSP HOB is located at 0x3EF20000
Install PPI: 605EA650-C65C-42E1-BA80-91A52AB618C6
Notify: PPI Guid: 605EA650-C65C-42E1-BA80-91A52AB618C6, Peim notify entry
   point: FFE3EB9A
Notify: PPI Guid: 605EA650-C65C-42E1-BA80-91A52AB618C6, Peim notify entry
   point: 3EFE6EE0
FspInitEndOfPeiCallback++
FspInitEndOfPeiCallback--
FSP is waiting for NOTIFY
FspSiliconInitApi() - [Status: 0x00000000] - End
success
pcie retraining failed FFFFFFFF
cap a 0
ddt disabled 0
device enable: 0
device enable: 128
NotifyPhaseApi() - Begin  [Phase: 00000020]
FSP Post PCI Enumeration ...
Install PPI: 30CFE3E7-3DE1-4586-BE20-DEABA1B3B793
Notify: PPI Guid: 30CFE3E7-3DE1-4586-BE20-DEABA1B3B793, Peim notify entry
   point: 3EFE6F12
FspInitAfterPciEnumerationCallback++
FspInitAfterPciEnumerationCallback--
NotifyPhaseApi() - End  [Status: 0x00000000]
NotifyPhaseApi() - Begin  [Phase: 00000040]
FSP Ready To Boot ...
Install PPI: 7CE88FB3-4BD7-4679-87A8-A8D8DEE50D2B
Notify: PPI Guid: 7CE88FB3-4BD7-4679-87A8-A8D8DEE50D2B, Peim notify entry
   point: 3EFE6F44
FspReadyToBootCallback++
FspReadyToBootCallback--
NotifyPhaseApi() - End  [Status: 0x00000000]
NotifyPhaseApi() - Begin  [Phase: 000000F0]
FSP End of Firmware ...
Install PPI: BD44F629-EAE7-4198-87F1-39FAB0FD717E
Notify: PPI Guid: BD44F629-EAE7-4198-87F1-39FAB0FD717E, Peim notify entry
   point: 3EFE6F76
FspEndOfFirmwareCallback++
FspEndOfFirmwareCallback--
NotifyPhaseApi() - End  [Status: 0x00000000]
```

```
CPUID(0):D 68747541 444D4163
mem: [ 0x1FFFF00, 0x200CC70 ] - wolfboot (0xCD70)
mem: [ 0x200CC70, 0x222FA00 ] - wolfboot .bss (0x222D90)
load wolfboot end
Authenticating wolfboot at 2000000...
Image size 52336
verify_payload: image open successfully.
verify_payload: integrity OK. Checking signature.
wolfBoot: verified OK.
starting wolfboot 64bit
AHCI port 0: No disk detected
AHCI port 1: No disk detected
AHCI port 2: No disk detected
AHCI port 3: No disk detected
AHCI port 4: No disk detected
AHCI port 5: Disk detected (det: 3 ipm: 1)
SATA disk drive detected on AHCI port 5
Reading MBR...
Found GPT PTE at sector 1
Found valid boot signature in MBR
Valid GPT partition table
Current LBA: 0x1
Backup LBA: 0x1FFFF
Max number of partitions: 128
Software limited: only allowing up to 16 partitions per disk.
Disk size: 66043392
disk0.p0 (0_1000000h@ 0_100000)
disk0.p1 (0_1000000h@ 0_1100000)
Total partitions on disk0: 2
Checking primary OS image in 0,0...
Checking secondary OS image in 0,1...
Versions, A:1 B:2
Load address 0x222FA00
Attempting boot from partition B
mem: [ 0x222FA00, 0x2241DC8 ] - ELF (0x123C8)
Loading image from disk...done.
Image size 74696
Checking image integrity...done.
Verifying image signature...done.
Firmware Valid.
Booting at 222FB00
mem: [ 0x100, 0x1E0 ] - MPTABLE (0xE0)
Loading elf at 0x222FB00
Found valid elf64 (little endian)
Program Headers 7 (size 56)
Load 504 bytes (offset 0x0) to 0x400000 (p 0x400000)
Load 3999 bytes (offset 0x1000) to 0x401000 (p 0x401000)
Load 1952 bytes (offset 0x2000) to 0x402000 (p 0x402000)
Load 32 bytes (offset 0x3000) to 0x403000 (p 0x403000)
Entry point 0x401000
Elf loaded (ret 0), entry 0x0_401000
mb2 header found at 2232B00
booting...
wolfBoot QEMU x86 FSP test app
```

3.33.2 Running on QEMU with swtpm (TPM emulator)

3.33.1.1 Sample boot output using config/examples/x86_fsp_qemu.config First step: clone and install swtpm, a TPM emulator that can be connected to qemu guest VMs. This TPM emulator will create a memory-mapped I/O device.

A small note is that `config/examples/x86_fsp_qemu_seal.config` showcases two different key ecc size of 384 and 256 of authentication for image verification and TPM sealing respectively.

The correct steps to run the example:

```
# copy the example configuration for this target
cp config/examples/x86_fsp_qemu_seal.config .config

# create necessary Intel FSP binaries from edk2 repo
tools/scripts/x86_fsp/qemu/qemu_build_fsp.sh

# make keytools and tpmtools
make keytools
make tpmtools

# create two keys, one for signing the images (ecc384) and one to seal/unseal
    secret into the TPM (ecc256)
./tools/keytools/keygen --force --ecc384 -g wolfboot_signing_private_key.der
    --ecc256 -g tpm_seal_key.key

# build wolfboot, manually add ECC256 for TPM
make CFLAGS_EXTRA="-DHAVE_ECC256"

# compute the value of PCR0 to sign with TPM key
PCR0=$(python ./tools/scripts/x86_fsp/compute_pcr.py --target qemu
    wolfboot_stage1.bin | tail -n 1)

# sign the policy
./tools/tpm/policy_sign -ecc256 -key=tpm_seal_key.key  -pcr=0 -pcrdigest=$PCR0

# install the policy
./tools/scripts/x86_fsp/tpm_install_policy.sh policy.bin.sig

# make test-app
make test-app/image.elf

# make_hd.sh sign the image, creates a file-based hard disk image with GPT
    table and raw partitions and then copy the signed images into the
    partitions.
IMAGE=test-app/image.elf SIGN=--ecc384 tools/scripts/x86_fsp/qemu/make_hd.sh

# run wolfBoot + test-image, use -t to emulate a TPM (requires swtpm)
./tools/scripts/x86_fsp/qemu/qemu.sh -t
```

For more advanced uses of TPM, please check Appendix M to configure wolfBoot according to your secure boot strategy.

3.33.3 Running on Kontron VX3060-S2

A reference configuration and helper scripts are provided to run wolfBoot on Kontron VX3060-S2 board. A flash dump of the original Flash BIOS is needed. To compile a flashable image run the following steps:

```
cp config/examples/kontron_vx3060_s2.config .config
./tools/scripts/x86_fsp/tgl/tgl_download_fsp.sh
make tpmtools
./tools/scripts/x86_fsp/tgl/assemble_image.sh -k
make CFLAGS_EXTRA="-DHAVE_ECC256"
./tools/scripts/x86_fsp/tgl/assemble_image.sh -n /path/to/original/flash/dump
```

they produce a file named `final_image.bin` inside the root folder of the repository that can be directly flashed into the BIOS flash of the board.

4 Hardware abstraction layer

In order to run wolfBoot on a target microcontroller, an implementation of the HAL must be provided.

The HAL's purpose is to allow write/erase operations from the bootloader and the application initiating the firmware upgrade through the application library, and ensuring that the MCU is running at full speed during boot (to optimize the verification of the signatures).

The implementation of the hardware-specific calls for each platform are grouped in a single c file in the `hal` directory.

The directory also contains a platform-specific linker script for each supported MCU, with the same name and the `.ld` extension. This is used to link the bootloader's firmware on the specific hardware, exporting all the necessary symbols for flash and RAM boundaries.

4.1 Supported platforms

Please see Chapter 3

4.2 API

The Hardware Abstraction Layer (HAL) consists of six function calls be implemented for each supported target:

```
void hal_init(void)
```

This function is called by the bootloader at the very beginning of the execution. Ideally, the implementation provided configures the clock settings for the target microcontroller, to ensure that it runs at at the required speed to shorten the time required for the cryptography primitives to verify the firmware images.

```
void hal_flash_unlock(void)
```

If the IAP interface of the flash memory of the target requires it, this function is called before every write and erase operations to unlock write access to the flash. On some targets, this function may be empty.

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

This function provides an implementation of the flash write function, using the target's IAP interface. `address` is the offset from the beginning of the flash area, `data` is the payload to be stored in the flash using the IAP interface, and `len` is the size of the payload. Implementations of this function must be able to handle writes of any size and alignment. Targets with a minimum programmable size > 1 byte must implement the appropriate read-modify-write logic in order to enable wolfBoot to perform unaligned single-byte writes. `hal_flash_write` should return 0 upon success, or a negative value in case of failure.

```
void hal_flash_lock(void)
```

If the IAP interface of the flash memory requires locking/unlocking, this function restores the flash write protection by excluding write accesses. This function is called by the bootloader at the end of every write and erase operations.

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

Called by the bootloader to erase part of the flash memory to allow subsequent boots. Erase operations must be performed via the specific IAP interface of the target microcontroller. `address` marks the start of the area that the bootloader wants to erase, and `len` specifies the size of the area to be erased. `address` is guaranteed to be aligned to `WOLFBOOT_SECTOR_SIZE`, and `len` is guaranteed to

be a multiple of WOLFBOOT_SECTOR_SIZE. This function must take into account the geometry of the flash sectors, and erase all the sectors in between.

```
void hal_prepare_boot(void)
```

This function is called by the bootloader at a very late stage, before chain-loading the firmware in the next stage. This can be used to revert all the changes made to the clock settings, to ensure that the state of the microcontroller is restored to its original settings.

4.2.1 Optional support for external flash memory

WolfBoot can be compiled with the makefile option EXT_FLASH=1. When the external flash support is enabled, update and swap partitions can be associated to an external memory, and will use alternative HAL function for read/write/erase access. It can also be used in any scenario where flash reads require special handling and must be redirected to a custom implementation. Note that EXT_FLASH=1 is incompatible with the NVM_FLASH_WRITEONCE option.

To associate the update or the swap partition to an external memory, define PART_UPDATE_EXT and/or PART_SWAP_EXT, respectively.

The following functions are used to access the external memory, and must be defined when EXT_FLASH is on:

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

This function provides an implementation of the flash write function, using the external memory's specific interface. `address` is the offset from the beginning of the addressable space in the device, `data` is the payload to be stored, and `len` is the size of the payload. The function is subject to the same restrictions as `hal_flash_write()`. `ext_flash_write` should return 0 upon success, or a negative value in case of failure.

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

This function provides an indirect read of the external memory, using the driver's specific interface. `address` is the offset from the beginning of the addressable space in the device, `data` is a pointer where payload is stored upon a successful call, and `len` is the maximum size allowed for the payload. This function must be able to handle reads of any size and alignment. `ext_flash_read` should return 0 upon success, or a negative value in case of failure.

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

Called by the bootloader to erase part of the external memory. Erase operations must be performed via the specific interface of the target driver (e.g. SPI flash). `address` marks the start of the area relative to the device, that the bootloader wants to erase, and `len` specifies the size of the area to be erased. This function is subject to the same restrictions as `hal_flash_erase()` and must take into account the geometry of the sectors, and erase all the sectors in between.

```
void ext_flash_lock(void)
```

If the interface of the external flash memory requires locking/unlocking, this function may be used to restore the flash write protection or exclude write accesses. This function is called by the bootloader at the end of every write and erase operations on the external device.

```
void ext_flash_unlock(void)
```

If the IAP interface of the external memory requires it, this function is called before every write and erase operations to unlock write access to the device. On some drivers, this function may be empty.

4.2.2 Additional functions required by `DUALBANK_SWAP` option

If the target device supports hardware-assisted bank swapping, it is appropriate to provide two additional functions in the port:

`void hal_flash_dualbank_swap(void)`

Called by the bootloader when the two banks must be swapped. On some architectures this operation implies a reboot, so this function may also never return.

`void fork_bootloader(void)`

This function is called to provide a second copy of the bootloader. Wolfboot will clone itself if the content does not already match. `fork_bootloader()` implementation in new ports must return immediately without performing any actions if the content of the bootloader partition in the two banks already match.

5 Flash partitions

5.1 Flash memory partitions

To integrate wolfBoot you need to partition the flash into separate areas (partitions), taking into account the geometry of the flash memory.

Images boundaries **must** be aligned to physical sectors, because the bootloader erases all the flash sectors before storing a new firmware image, and swaps the content of the two partitions, one sector at a time.

For this reason, before proceeding with partitioning on a target system, the following aspects must be considered:

- BOOT partition and UPDATE partition must have the same size, and be able to contain the running system
- SWAP partition must be as big as the largest sector in both BOOT and UPDATE partition.

The flash memory of the target is partitioned into the following areas:

- Bootloader partition, at the beginning of the flash, generally very small (16-32KB)
- Primary slot (BOOT partition) starting at address `WOLFBOOT_PARTITION_BOOT_ADDRESS`
- Secondary slot (UPDATE partition) starting at address `WOLFBOOT_PARTITION_UPDATE_ADDRESS`
  - both partitions share the same size, defined as `WOLFBOOT_PARTITION_SIZE`
- Swapping space (SWAP partition) starting at address `WOLFBOOT_PARTITION_SWAP_ADDRESS`
  - the swap space size is defined as `WOLFBOOT_SECTOR_SIZE` and must be as big as the largest sector used in either BOOT/UPDATE partitions.

A proper partitioning configuration must be set up for the specific use, by setting the values for offsets and sizes in `include/target.h`.

5.1.1 Bootloader partition

This partition is usually very small, and only contains the bootloader code and data. Public keys pre-authorized during factory image creations are automatically stored as part of the firmware image.

5.1.2 BOOT partition

This is the only partition from where it is possible to chain-load and execute a firmware image. The firmware image must be linked so that its entry-point is at address `WOLFBOOT_PARTITION_BOOT_ADDRESS + 256`.

5.1.3 UPDATE partition

The running firmware is responsible for transferring a new firmware image through a secure channel, and store it in the secondary slot. If an update is initiated, the bootloader will replace or swap the firmware in the boot partition at the next reboot.

5.2 Partition status and sector flags

Partitions are used to store firmware images currently in use (BOOT) or ready to swap in (UPDATE). In order to track the status of the firmware in each partition, a 1-Byte state field is stored at the end of each partition space. This byte is initialized when the partition is erased and accessed for the first time.

Possible states are: - `IMG_STATE_NEW` (0xFF): The image was never staged for boot, or triggered for an update. If an image is present, no flags are active. - `IMG_STATE_UPDATING` (0x70): Only valid in

the UPDATE partition. The image is marked for update and should replace the current image in BOOT. - IMG_STATE_TESTING (0x10): Only valid in the BOOT partition. The image has been just updated, and never completed its boot. If present after reboot, it means that the updated image failed to boot, despite being correctly verified. This particular situation triggers a rollback. - IMG_STATE_SUCCESS (0x00): Only valid in the BOOT partition. The image stored in BOOT has been successfully staged at least once, and the update is now complete.

Starting from the State byte and growing backwards, the bootloader keeps track of the state of each sector, using 4-bits per sector at the end of the UPDATE partition. Whenever an update is initiated, the firmware is transferred from UPDATE to BOOT one sector at a time, and storing a backup of the original firmware from BOOT to UPDATE. Each flash access operation correspond to a different value of the flags for the sector in the sector flags area, so that if the operation is interrupted, it can be resumed upon reboot.

End of flash layout: * 4-bits flag (sector 1) * 4-bits flag (sector 0) * 1-byte partition state * 4-byte trailer “BOOT”

If the FLAGS_HOME build option is used then all flags are placed at the end of the boot partition:

```
                        / -12    /-8       /-4     / END
  |Sn| ... |S2|S1|S0|PU|  MAGIC  |X|X|X|PB| MAGIC |
   ^--sectors   --^  ^--update           ^---boot partition
      flags            partition            flag
                       flag
```

You can use the CUSTOM_PARTITION_TRAILER option to implement your own functions for: get_trailer_at, set_trailer_at and set_partition_magic.

To enable: 1) Add the CUSTOM_PARTITION_TRAILER build option to your .config: CFLAGS_EXTRA+=--DCUSTOM_PARTITION_TRAILER 2) Add your own .c file using OBJS_EXTRA. For example for your own src/custom_trailer.c add this to your .config: OBJS_EXTRA=src/custom_trailer.o.

5.3 Overview of the content of the FLASH partitions

6 wolfBoot Features

6.1 Signing

6.1.1 wolfBoot key tools installation

Instructions for setting up Python, wolfCrypt-py module and wolfBoot for firmware signing and key generation.

Note: There is a pure C version of the key tool available as well. See C Key Tools below.

6.1.2 Install Python3

1. Download latest Python 3.x and run installer: https://www.python.org/downloads
2. Check the box that says Add Python 3.x to PATH

6.1.3 Install wolfCrypt

```
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"
```

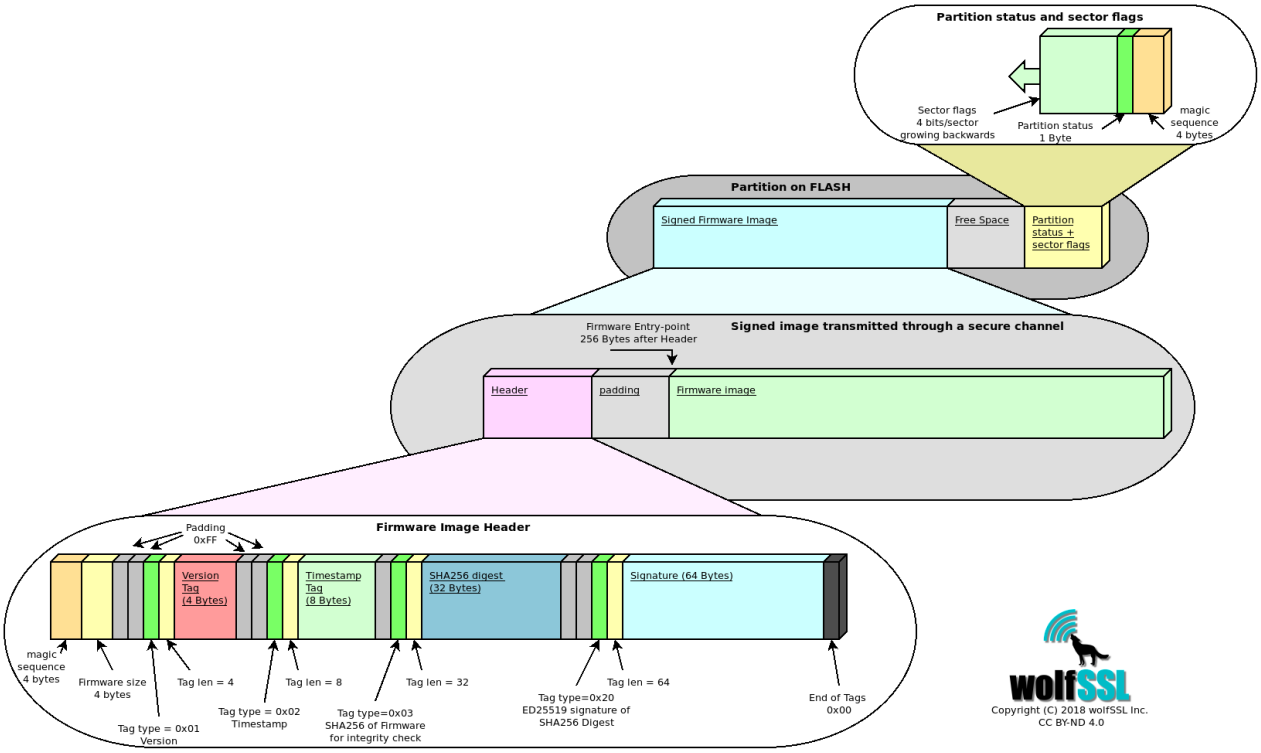

Figure 2: wolfBoot partition

```
make
sudo make install
```

6.1.4 Install wolfcrypt-py

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

6.1.5 Install wolfBoot

```
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 Key Tools

A standalone C version of the keygen tools is available in: `./tools/keytools`.

These can be built in `tools/keytools` using `make` or from the wolfBoot root using `make keytools`.

If the C version of the key tools exists they will be used by wolfBoot (the default is the Python scripts).

6.1.6.1 Windows Visual Studio Use the `wolfBootSignTool.vcxproj` Visual Studio project to build the `sign.exe` and `keygen.exe` tools for use on Windows.

6.1.7 Command Line Usage

6.1.7.1 Keygen tool Usage: `keygen[.py] [OPTIONS] [-g new-keypair.der] [-i existing-pubkey.der] [...]`

keygen is used to populate a keystore with existing and new public keys. Two options are supported:

- `-g privkey.der` to generate a new keypair, add the public key to the keystore and save the private key in a new file `privkey.der`
- `-i existing.der` to import an existing public key from `existing.der`

Arguments are not exclusive, and can be repeated more than once to populate a keystore with multiple keys.

One option must be specified to select the algorithm enabled in the keystore (e.g. `--ed25519` or `--rsa3072`. See the section "Public key signature options" for the sign tool for the available options.

The files generate by the keygen tool is the following:

- A C file `src/keystore.c`, which is normally linked with the wolfBoot image, when the keys are provisioned through generated C code.
- A binary file `keystore.img` that can be used to provision the public keys through an alternative storage
- The private key, for each `-g` option provided from command line

6.1.7.2 Sign tool `sign` and `sign.py` produce a signed firmware image by creating a manifest header in the format supported by wolfBoot.

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

`IMAGE.BIN`: A file containing the binary firmware/software to sign `KEY.DER`: Private key file, in DER format, to sign the binary image `VERSION`: The version associated with this signed software `OPTIONS`: Zero or more options, described below

6.1.7.3 Public key signature options If none of the following arguments is given, the tool will try to guess the key size from the format and key length detected in KEY.DER.

- `--ed25519` Use ED25519 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--ed448` Use ED448 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--ecc256` Use ecc256 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--ecc384` Use ecc384 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--rsa2048` Use rsa2048 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--rsa3072` Use rsa3072 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--rsa4096` Use rsa4096 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--no-sign` Disable secure boot signature verification. No signature verification is performed in the bootloader, and the KEY.DER argument is ignored.

6.1.8 Key generation and management

KeyStore is the name of the mechanism used by wolfBoot to store all the public keys used for authenticating the signature of current firmware and updates.

wolfBoot's key generation tool can be used to generate one or more keys. By default, when running `make` for the first time, a single key `wolfboot_signing_private_key.der` is created, and added to the keystore module. This key should be used to sign any firmware running on the target, as well as firmware update binaries.

Additionally, the keygen tool creates additional files with different representations of the keystore - A .c file (src/keystore.c) which can be used to deploy public keys as part of the bootloader itself, by linking the keystore in `wolfboot.elf` - A .bin file (keystore.bin) which contains the keystore that can be hosted on a custom memory support. In order to access the keystore, a small driver is required (see section "Interface API" below).

By default, the keystore object in `src/keystore.c` is accessed by wolfboot by including its symbols in the build. Once generated, this file contains an array of structures describing each public key that will be available to wolfBoot on the target system. Additionally, there are a few functions that connect to the wolfBoot keystore API to access the details and the content of the public key slots.

The public key is described by the following structure:

```
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` is the incremental identifier for the key slot, starting from 0.
- `key_type` describes the algorithm of the key, e.g. `AUTH_KEY_ECC256` or `AUTH_KEY_RSA3072`
- `mask` describes the permissions for the key. It's a bitmap of the partition ids for which this key can be used for verification
- `pubkey_size` the size of the public key buffer
- `pubkey` the actual buffer containing the public key in its raw format

When booting, wolfBoot will automatically select the public key associated to the signed firmware image, check that it matches the permission mask for the partition id where the verification is running and then attempts to authenticate the signature of the image using the selected public key slot.

6.1.8.1 Creating multiple keys

keygen accepts multiple filenames for private keys.

Two arguments:

- `-g priv.der` generate new keypair, store the private key in priv.der, add the public key to the keystore
- `-i pub.der` import an existing public key and add it to the keystore

Example of creation of a keystore with two ED25519 keys:

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

will create the following files:

- `first.der` first private key
- `second.der` second private key

- src/keystore.c C keystore containing both public keys associated with first.der and second.der.

The keystore.c generated should look similar to this:

```
#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 Public keys and permissions By default, when a new keystore is created, the permissions mask is set to KEY_VERIFY_ALL, which means that the key can be used to verify a firmware targeting any partition id.

To restrict the permissions for single keys, it would be sufficient to change the value of their part_id_mask attributes.

The part_id_mask value is a bitmask, where each bit represent a different partition. The bit '0' is reserved for wolfBoot self-update, while typically the main firmware partition is associated to id 1, so it requires a key with the bit '1' set. In other words, signing a partition with --id 3 would require turning on bit '3' in the mask, i.e. adding (1U « 3) to it.

Beside KEY_VERIFY_ALL, pre-defined mask values can also be used here:

- KEY_VERIFY_APP_ONLY only verifies the main application, with partition id 1
- KEY_VERIFY_SELF_ONLY this key can only be used to authenticate wolfBoot self-updates (id = 0)

- `KEY_VERIFY_ONLY_ID(N)` macro that can be used to restrict the usage of the key to a specific partition id `N`

6.1.9 Signing Firmware

1. Load the private key to use for signing into `./wolfboot_signing_private_key.der`
2. Run the signing tool with asymmetric algorithm, hash algorithm, file to sign, key and version.

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

Note: The last argument is the "version" number.

6.1.10 Signing Firmware with External Private Key (HSM)

Steps for manually signing firmware using an external key source.

```
# Create file with Public Key
openssl rsa -inform DER -outform DER -in my_key.der -out rsa2048_pub.der
 ↪  -pubout

# Add the public key to the wolfBoot keystore using `keygen -i`
./tools/keytools/keygen --rsa2048 -i rsa2048_pub.der
# OR
python3 ./tools/keytools/keygen.py --rsa2048 -i rsa4096_pub.der

# Generate Hash to Sign
./tools/keytools/sign            --rsa2048 --sha-only --sha256
 ↪  test-app/image.bin rsa2048_pub.der 1
# OR
python3 ./tools/keytools/sign.py --rsa2048 --sha-only --sha256
 ↪  test-app/image.bin rsa4096_pub.der 1

# Sign hash Example (here is where you would use an HSM)
openssl pkeyutl -sign -keyform der -inkey my_key.der -in
 ↪  test-app/image_v1_digest.bin > test-app/image_v1.sig

# Generate final signed binary
./tools/keytools/sign            --rsa2048 --sha256 --manual-sign
 ↪  test-app/image.bin rsa2048_pub.der 1 test-app/image_v1.sig
# OR
python3 ./tools/keytools/sign.py --rsa2048 --sha256 --manual-sign
 ↪  test-app/image.bin rsa4096_pub.der 1 test-app/image_v1.sig

# Combine into factory image (0xc0000 is the WOLFBOOT_PARTITION_BOOT_ADDRESS)
tools/bin-assemble/bin-assemble factory.bin 0x0 wolfboot.bin \
                                0xc0000 test-app/image_v1_signed.bin
```

6.2 Measured Boot using wolfBoot

wolfBoot offers a simplified measured boot implementation, a way to record and track the state of the system boot process using a Trusted Platform Module(TPM).

This record is tamper-proofed by special registers in the TPM called Platform Configuration Register. Then, the firmware application, RTOS or rich OS(Linux), can access that log of information by reading the PCRs of the TPM.

wolfBoot can interact with TPM2.0 chips thanks to its integration with wolfTPM. wolfTPM has native support for Microsoft Windows and Linux, and can be used standalone or together with wolfBoot. The combination of wolfBoot with wolfTPM gives the developer a tamper-proof secure storage for protecting the system during and after boot.

6.2.1 Concept

Typically, systems use Secure Boot to guarantee that the correct and geniune firmware is booted by verifying its signature. Afterwards, this knowledge is unknown to the sytem. The application does not know if the system started in a good known state. Sometimes, this guarantee is needed by the firmware itself. To provide such mechanism the concept of Measured Boot exist.

Measured Boot can be used to check every start-up component, including settings and user information(user partition). The result of the checks is then stored into special registers called PCR. This process is called PCR Extend and is refered to as a TPM measurement. PCR registers can be reset only on TPM power-on.

Having TPM measurements provide a way for the firmware or Operating System(OS), like Windows or Linux, to know that the software loaded before it gained control over system, is trustworthy and not modified.

In wolfBoot the concept is simplified to measuring a single component, the main firmware image. However, this can easily be extended by using more PCR registers.

6.2.2 Configuration

To enable measured boot add `MEASURED_BOOT=1` setting in your wolfBoot config.

It is also necessary to select the PCR (index) where the measurement will be stored.

Selection is made using the `MEASURED_BOOT_PCR_A=[index]` setting. Add this setting in your wolfBoot config and replace `[index]` with a number between 0 and 23. Below you will find guidelines for selecting a PCR index.

Any TPM has a minimum of 24 PCR registers. Their typical use is as follows:

Index	Typical use	Recommended to use with
0	Core Root of Trust and/or BIOS measurement	bare-metal, RTOS
1	measurement of Platform Configuration Data	bare-metal, RTOS
2-3	Option ROM Code measurement	bare-metal, RTOS
4-5	Master Boot Record measurement	bare-metal, RTOS
6	State Transitions	bare-metal, RTOS
7	Vendor specific	bare-metal, RTOS
8-9	Partition measurements	bera-metal, RTOS
10	measurement of the Boot Manager	bare-metal, RTOS
11	Typically used by Microsoft Bitlocker	bare-metal, RTOS
12-15	Available for any use	bare-metal, RTOS, Linux, Windows
16	DEBUG	Use only for test purposes
17	DRTM	Trusted Bootloader
18-22	Trusted OS	Trusted Execution Environment(TEE)
23	Application	Use only for temporary measurements

Recommendations for choosing a PCR index:

- During development it is recommended to use PCR16 that is intented for testing.
- In production, if you are running a bare-metal firmware or RTOS, you could use almost all PCRs(PCR0-15), except the one for DRTM and Trusted OS(PCR17-23).
- If you are running Linux or Windows, PCR12-15 can be chosen for production ready firmware, in order to avoid conflict with other software that might be using PCRs from within Linux, like the Linux IMA or Microsoft Bitlocker.

Here is an example part of a wolfBoot .config during development:

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

6.2.2.1 Code wolfBoot offers out-of-the-box solution. There is zero need of the developer to touch wolfBoot code in order to use measured boot. If you would want to check the code, then look in `src/image.c` and more specifically the `measure_boot()` function. There you would find several TPM2 native API calls to wolfTPM. For more information about wolfTPM you can check its GitHub repository.

6.3 Firmware image

6.3.1 Firmware entry point

WolfBoot can only chain-load and execute firmware images from a specific entry point in memory, which must be specified as the origin of the FLASH memory in the linker script of the embedded application. This corresponds to the first partition in the flash memory.

Multiple firmware images can be created this way, and stored in two different partitions. The bootloader will take care of moving the selected firmware to the first (BOOT) partition before chain-loading the image.

Due to the presence of an image header, the entry point of the application has a fixed additional offset of 256B from the beginning of the flash partition.

6.3.2 Firmware image header

Each (signed) firmware image is prepended with a fixed-size **image header**, containing useful information about the firmware. The exact size of the **image header** depends on the size of the image digest and signature, which depend on the algorithms/key sizes used. Larger key sizes will result in a larger image header. The size of the image header is determined by the build system and provided to the application code in the `IMAGE_HEADER_SIZE` macro. The size of the generated image header is also output by the keytools during the signing operation. The **image header** data is padded out to the next multiple of 256B, in order to guarantee that the entry point of the actual firmware is stored on the flash starting from a 256-Bytes aligned address. This ensures that the bootloader can relocate the vector table before chain-loading the firmware so interrupts continue to work properly after the boot is complete. When porting wolfBoot to a platform that doesn't use wolfBoot's Makefile-based build system, extra care should be taken to ensure `IMAGE_HEADER_SIZE` is set to a value that matches the output of the wolfBoot `sign` key tool.

The image header is stored at the beginning of the slot and the actual firmware image starts `IMAGE_HEADER_SIZE` *Bytes after it*

6.3.2.1 Image header: Tags The **image header** is prepended with a single 4-byte magic number, followed by a 4-byte field indicating the firmware image size (excluding the header). All numbers in the header are stored in Little-endian format.

The two fixed fields are followed by one or more tags. Each TAG is structured as follows:

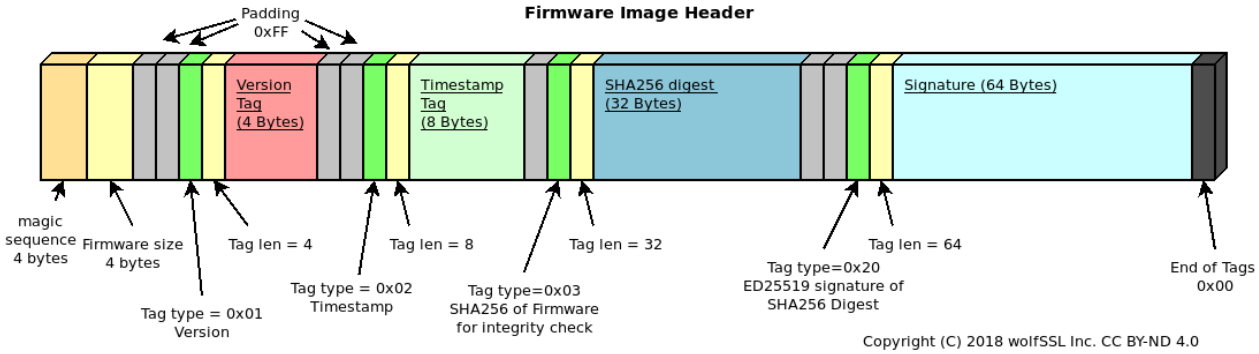

Figure 3: Image header

- 2 bytes indicating the **Type**
- 2 bytes indicating the **size** of the tag, excluding the type and size bytes
- ***N*** bytes of tag content

With the following exception: - A '0xFF' in the Type field indicate a simple padding byte. The 'padding' byte has no **size** field, and the next byte should be processed as **Type** again.

Each **Type** has a different meaning, and integrate information about the firmware. The following Tags are mandatory for validating the firmware image: - A 'version' Tag (type: 0x0001, size: 4 Bytes) indicating the version number for the firmware stored in the image - A 'timestamp' Tag (type: 0x0002, size 8 Bytes) indicating the timestamp in unix seconds for the creation of the firmware - A 'sha digest' Tag (type: 0x0003, size: digest size (32 Bytes for SHA256)) used for integrity check of the firmware - A 'firmware signature' Tag (type: 0x0020, size: 64 Bytes) used to validate the signature stored with the firmware against a known public key - A 'firmware type' Tag (type: 0x0030, size: 2 Bytes) used to identify the type of firmware, and the authentication mechanism in use.

A 'public key hint digest' tag is transmitted in the header (type: 0x10, size:32 Bytes). This tag contains the SHA digest of the public key used by the signing tool. The bootloader may use this field to locate the correct public key in case of multiple keys available.

wolfBoot will, in all cases, refuse to boot an image that cannot be verified and authenticated using the built-in digital signature authentication mechanism.

6.3.2.2 Adding custom fields to the manifest header

It is possible to add custom fields to the manifest header, by using the `--custom-tlv` option in the signing tool.

In order for the fields to be secured (checked by wolfBoot for integrity and authenticity), their value is placed in the manifest header before the signature is calculated. The signing tool takes care of the alignment and padding of the fields.

The custom fields are identified by a 16-bit tag, and their size is indicated by a 16-bit length field. The tag and length fields are stored in little-endian format.

At runtime, the values stored in the manifest header can be accessed using the `wolfBoot_find_header` function.

The syntax for `--custom-tlv` option is also documented in Appendix H.

6.3.2.3 Image header: Example

This example adds a custom field when the signing tool is used to sign the firmware image:

```
./tools/keytools/sign --ed25519 --custom-tlv 0x34 4 0xAABBCCDD
 ↪  test-app/image.bin wolfboot_signing_private_key.der 4
```

The output image `test-app/image_v4_signed.bin` will contain the custom field with tag `0x34` with length 4 and value `0xAABBCCDD`.

From the bootloader code, we can then retrieve the value of the custom field using the `wolfBoot_find_header` function:

```
uint32_t value;
uint8_t* ptr = NULL;
uint16_t tlv = 0x34;
uint8_t* imageHdr = (uint8_t*)WOLFBOOT_PARTITION_BOOT_ADDRESS +
 ↪  IMAGE_HEADER_OFFSET;
uint16_t size = wolfBoot_find_header(imageHdr, tlv, &ptr);
if (size > 0 && ptr != NULL) {
  /* Found field and ptr points to value 0xAABBCCDD */
  memcpy(&value, ptr, size);
  printf("TLV 0x%x=0x%x\n", tlv, value);
}
else {
    /* Error: the field is not found */
}
```

6.3.3 Image signing tool

The image signing tool generates the header with all the required Tags for the compiled image, and add them to the output file that can be then stored on the primary slot on the device, or transmitted later to the device through a secure channel to initiate an update.

6.3.3.1 Storing firmware image Firmware images are stored with their full header at the beginning of any of the partitions on the system. wolfBoot can only boot images from the BOOT partition, while keeping a second firmware image in the UPDATE partition.

In order to boot a different image, wolfBoot will have to swap the content of the two images.

For more information on how firmware images are stored and managed within the two partitions, see Flash partitions

6.4 Firmware update

This section documents the complete firmware update procedure, enabling secure boot for an existing embedded application.

6.4.1 Updating Microcontroller FLASH

The steps to complete a firmware update with wolfBoot are: - Compile the firmware with the correct entry point - Sign the firmware - Transfer the image using a secure connection, and store it to the secondary firmware slot - Trigger the image swap - Reboot to let the bootloader begin the image swap

At any given time, an application or OS running on a wolfBoot system can receive an updated version of itself, and store the updated image in the second partition in the FLASH memory.

Applications or OS threads can be linked to the libwolfboot library, which exports the API to trigger the update at the next reboot, and some helper functions to access the flash partition for erase/write through the target specific HAL.

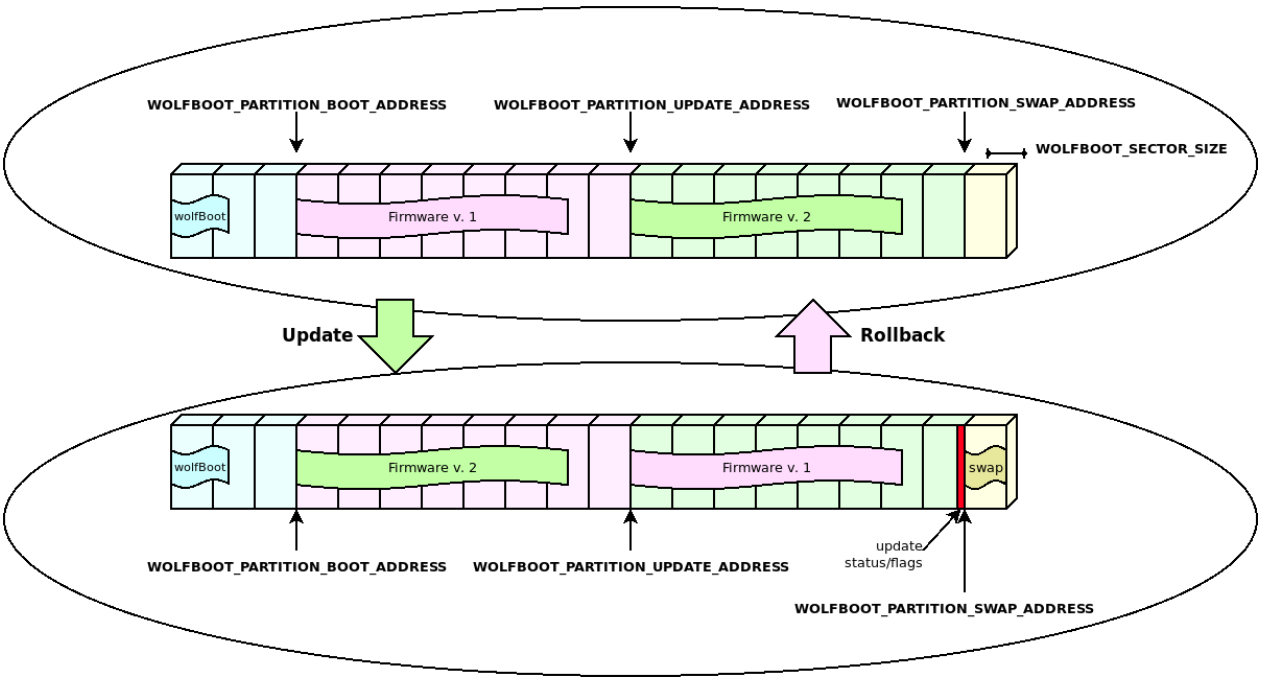

Figure 4: Update and Rollback

6.4.2 Update procedure description

Using the API provided to the application, wolfBoot offers the possibility to initiate, confirm or rollback an update.

After storing the new firmware image in the UPDATE partition, the application should initiate the update by calling `wolfBoot_update_trigger()`. By doing so, the UPDATE partition is marked for update. Upon the next reboot, wolfBoot will: - Validate the new firmware image stored in the UPDATE partition - Verify the signature attached against a known public key stored in the bootloader image - Swap the content of the BOOT and the UPDATE partitions - Mark the new firmware in the BOOT partition as in state `STATE_TESTING` - Boot into the newly received firmware

If the system is interrupted during the swap operation and reboots, wolfBoot will pick up where it left off and continue the update procedure.

6.4.2.1 Successful boot Upon a successful boot, the application should inform the bootloader by calling `wolfBoot_success()`, after verifying that the system is up and running again. This operation confirms the update to a new firmware.

Failing to set the BOOT partition to `STATE_SUCCESS` before the next reboot triggers a roll-back operation. Roll-back is initiated by the bootloader by triggering a new update, this time starting from the backup copy of the original (pre-update) firmware, which is now stored in the UPDATE partition due to the swap occurring earlier.

6.4.2.2 Building a new firmware image Firmware images are position-dependent, and can only boot from the origin of the **BOOT** partition in FLASH. This design constraint implies that the chosen firmware is always stored in the **BOOT** partition, and wolfBoot is responsible for pre-validating an update image and copy it to the correct address.

All the firmware images must therefore have their entry point set to the address corresponding to the beginning of the **BOOT** partition, plus an offset of 256 Bytes to account for the image header.

Once the firmware is compiled and linked, it must be signed using the `sign` tool. The tool produces a signed image that can be transferred to the target using a secure connection, using the same key corresponding to the public key currently used for verification.

The tool also adds all the required Tags to the image header, containing the signatures and the SHA256 hash of the firmware.

6.4.2.3 Self-update wolfBoot can update itself if `RAM_CODE` is set. This procedure operates almost the same as firmware update with a few key differences. The header of the update is marked as a bootloader update (use `--wolfboot-update` for the sign tools).

The new signed wolfBoot image is loaded into the UPDATE parition and triggered the same as a firmware update. Instead of performing a swap, after the image is validated and signature verified, the bootloader is erased and the new image is written to flash. This operation is *not* safe from interruption. Interruption will prevent the device from rebooting.

wolfBoot can be used to deploy new bootloader versions as well as update keys.

6.4.2.4 Incremental updates (aka: 'delta' updates) wolfBoot supports incremental updates, based on a specific older version. The sign tool can create a small "patch" that only contains the binary difference between the version currently running on the target and the update package. This reduces the size of the image to be transferred to the target, while keeping the same level of security through public key verification, and integrity due to the repeated check (on the patch and the resulting image).

The format of the patch is based on the mechanism suggested by Bentley/McIlroy, which is particularly effective to generate small binary patches. This is useful to minimize time and resources needed to transfer, authenticate and install updates.

6.4.2.4.1 How it works As an alternative to transferring the entire firmware image, the key tools create a binary diff between a base version previously uploaded and the new updated image.

The resulting bundle (delta update) contains the information to derive the content of version '2' of the firmware, starting from the base version, that is currently running on the target (version '1' in this example), and the reverse patch to downgrade version '2' back to version '1' if something goes wrong running the new version.

On the device side, wolfboot will recognize and verify the authenticity of the delta update before applying the patch to the current firmware. The new firmware is rebuilt in place, replacing the content of the BOOT partition according to the indication in the (authenticated) 'delta update' bundle.

6.4.2.4.2 Two-steps verification Binary patches are created by comparing signed firmware images. wolfBoot verifies that the patch is applied correctly by checking for the integrity and the authenticity of the resulting image after the patch.

The delta update bundle itself, containing the patches, is prefixed with a manifest header describing the details for the patch, and signed like a normal full update bundle.

This means that wolfBoot will apply two levels of authentication: the first one when the delta bundle is processed (e.g. when an update is triggered), and the second one every time a patch is applied, or reversed, to validate the firmware image before booting.

These steps are performed automatically by the key tools when using the `--delta` option, as described in the example.

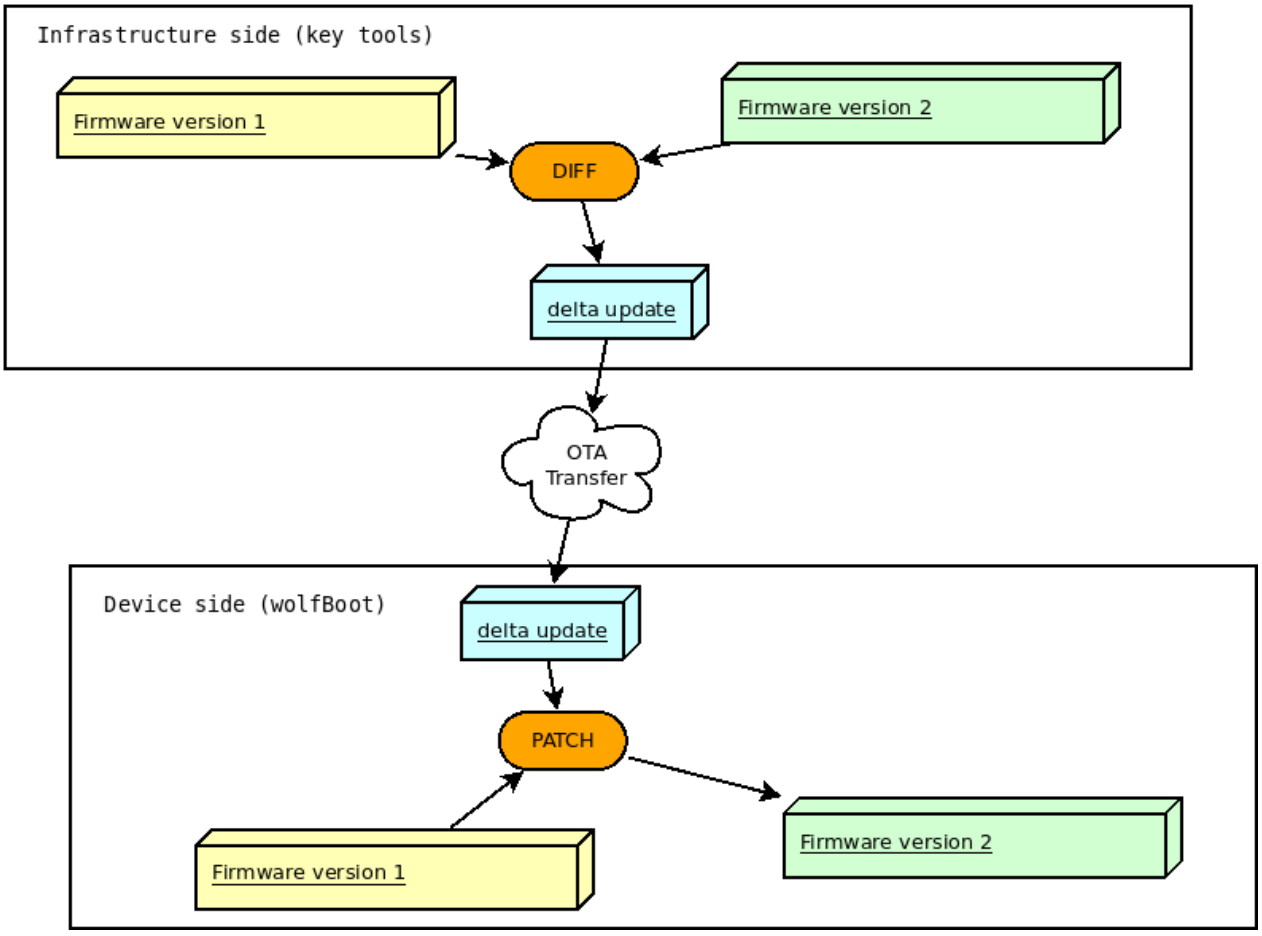

Figure 5: Delta update

6.4.2.4.3 Confirming the update From the application perspective, nothing changes from the normal, 'full' update case. Application must still call `wolfBoot_success()` on the first boot with the updated version to ensure that the update is confirmed.

Failing to confirm the success of the update will cause wolfBoot to revert the patch applied during the update. The 'delta update' bundle also contains a reverse patch, which can revert the update and restore the base version of the firmware.

The diagram below shows the authentication steps and the diff/patch process in both directions (update and roll-back for missed confirmation).

6.4.2.4.4 Incremental update: example Requirement: wolfBoot is compiled with `DELTA_UPDATES=1`

Version "1" is signed as usual, as a standalone image:

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

When updating from version 1 to version 2, you can invoke the sign tool as:

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

Besides the usual output file `image_v2_signed.bin`, the sign tool creates an additional `image_v2_signed_diff.bin` which should be noticeably smaller in size as long as the two binary files contain overlapping areas.

This is the delta update bundle, a signed package containing the patches for updating version 1 to version 2, and to roll back to version 1 if needed, after the first patch has been applied.

The delta bundle `image_v2_signed_diff.bin` can be now transferred to the update partition on the target like a full update image.

At next reboot, wolfBoot recognizes the incremental update, checks the integrity, the authenticity and the versions of the patch. If all checks succeed, the new version is installed by applying the patch on the current firmware image.

If the update is not confirmed, at the next reboot wolfBoot will restore the original base `image_v1_signed.bin`, using the reverse patch contained in the delta update bundle.

6.5 Remote External flash memory support via UART

wolfBoot can emulate external partitions using UART communication with a neighbor system. This feature is particularly useful in those asynchronous multi-process architectures, where updates can be stored with the assistance of an external processing unit.

6.5.1 Bootloader setup

The option to activate this feature is `UART_FLASH=1`. This configuration option depends on the external flash API, which means that the option `EXT_FLASH=1` is also mandatory to compile the bootloader.

The HAL of the target system must be expanded to include a simple UART driver, that will be used by the bootloader to access the content of the remote flash using one of the UART controllers on board.

Example UART drivers for a few of the supported platforms can be found in the `hal/uart` directory.

The API exposed by the UART HAL extension for the supported targets is composed by the following functions:

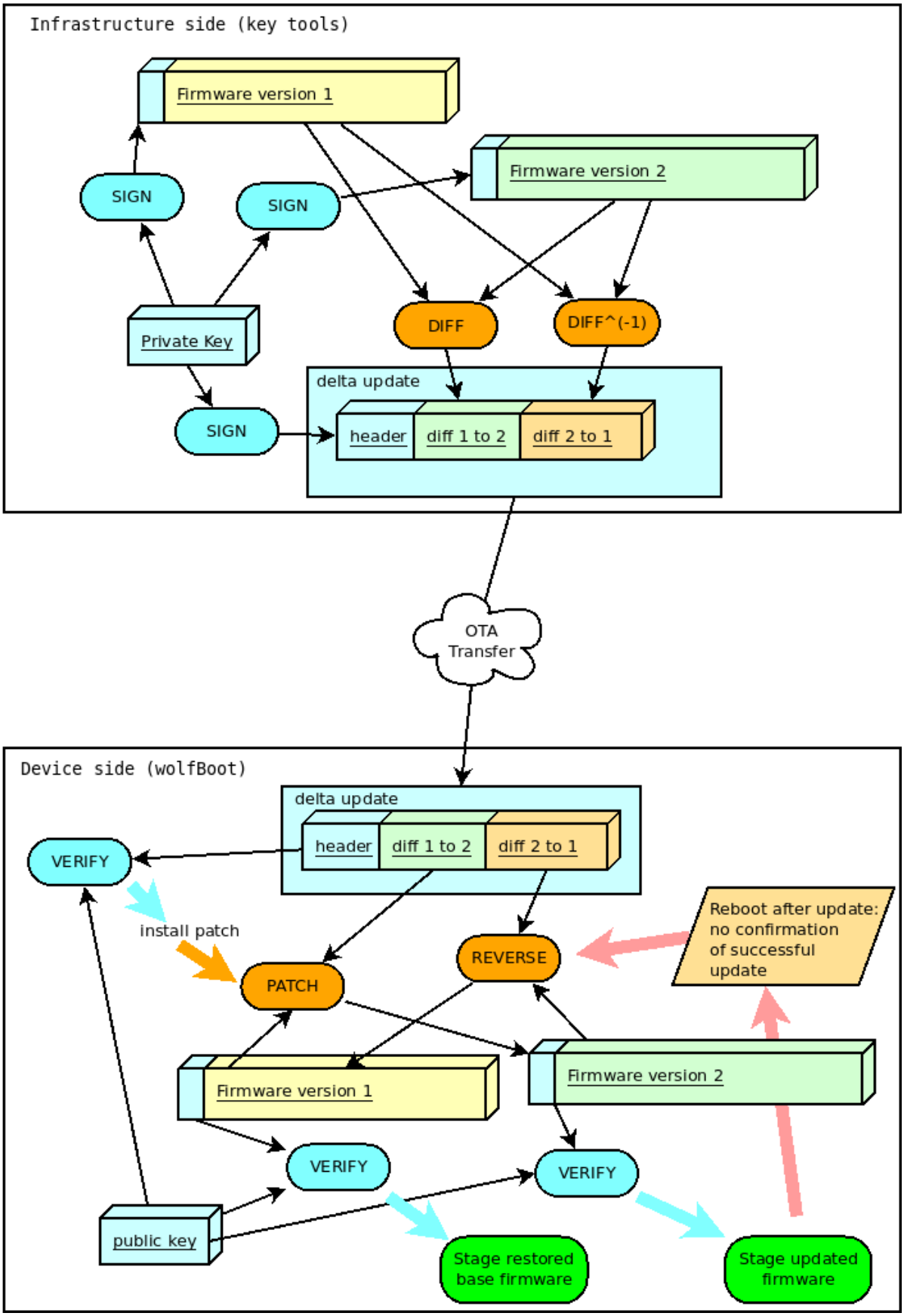

Figure 6: Delta update: details

```
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);
```

Consider implementing these three functions based on the provided examples if you want to use external flash memory support on your platform, if not officially supported yet.

6.5.2 Host side: UART flash server

On the remote system hosting the external partition image for the target, a simple protocol can be implemented on top of UART messages to serve flash-access specific calls.

An example uart-flash-server daemon, designed to run on a GNU/Linux host and emulate the external partition with a local file on the filesystem, is available in `tools/uart-flash-server`.

6.5.3 External flash update mechanism

wolfBoot treats external UPDATE and SWAP partitions in the same way as when they are mapped on a local SPI flash. Read and write operations are simply translated into remote procedure calls via UART, that can be interpreted by the remote application and provide read and write access to actual storage elements which would only be accessible by the host.

This means that after a successful update, a copy of the previous firmware will be stored in the remote partition to provide exactly the same update mechanism that is available in all the other use cases. The only difference consist in the way of accessing the physical storage area, but all the mechanisms at a higher level stay the same.

6.6 Encrypted external partitions

wolfBoot offers the possibility to encrypt the content of the entire UPDATE partition, by using a pre-shared symmetric key which can be temporarily stored in a safer non-volatile memory area.

SWAP partition is also temporarily encrypted using the same key, so a dump of the external flash won't reveal any content of the firmware update packages.

6.6.1 Rationale

Encryption of external partition works at the level of the external flash interface.

All write calls to external partitions from the bootloader perform an additional encryption step to hide the actual content of the external non-volatile memory.

Viceversa, all read operations will decrypt the data stored when the feature is enabled.

An extra option is provided to the `sign` tool to encrypt the firmware update after signing it, so that it can be stored as is in the external memory by the application, and will be decrypted by the bootloader in order to verify the update and begin the installation.

6.6.2 Temporary key storage

By default, wolfBoot will store the pre-shared symmetric key used for encryption in a temporary area on the internal flash. This allows read-out protections to be used to hide the temporary key.

Alternatively, more secure mechanisms are available to store the temporary key in a different key storage (e.g. using a hardware security module or a TPM device).

The temporary key can be set at run time by the application, and will be used exactly once by the bootloader to verify and install the next update. The key can be for example received from a back-end

during the update process using secure communication, and set by the application, using `libwolfboot` API, to be used by wolfBoot upon next boot.

Aside from setting the temporary key, the update mechanism remains the same for distributing, uploading and installing firmware updates through wolfBoot.

6.6.3 Libwolfboot API

The API to communicate with the bootloader from the application is expanded when this feature is enabled, to allow setting a temporary key to process the next update.

The functions

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

can be used to set a temporary encryption key for the external partition, or erase a key previously set, respectively.

Moreover, using `libwolfboot` to access the external flash with wolfboot hal from the application will not use encryption. This way the received update, already encrypted at origin, can be stored in the external memory unchanged, and retrieved in its encrypted format, e.g. to verify that the transfer has been successful before reboot.

6.6.4 Symmetric encryption algorithms

The default algorithm used to encrypt and decrypt data in external partitions is Chacha20-256.

- The key provided to `wolfBoot_set_encrypt_key()` must be exactly 32 Bytes long.
- The nonce argument must be a 96-bit (12 Bytes) randomly generated buffer, to be used as IV for encryption and decryption.

AES-128 and AES-256 are also supported. AES is used in counter mode. AES-128 and AES-256 have a key length of 16 and 32 bytes respectively, and the IV size is 16 bytes long in both cases.

6.6.5 Example usage

To compile wolfBoot with encryption support, use the option `ENCRYPT=1`.

By default, this also selects `ENCRYPT_WITH_CHACHA=1`. To use AES encryption instead, select `ENCRYPT_WITH_AES128=1` or `ENCRYPT_WITH_AES256=1`.

6.6.6 Signing and encrypting the update bundle with ChaCha20-256

The `sign` tool can sign and encrypt the image with a single command. In case of chacha20, the encryption secret is provided in a binary file that should contain a concatenation of a 32B ChaCha-256 key and a 12B nonce.

In the examples provided, the test application uses the following parameters:

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

So it is easy to prepare the encryption secret in the test scripts or from the command line using:

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

The `sign` tool can now be invoked to produce a signed+encrypted image, by using the extra argument `--encrypt` followed by the secret file:

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

which will produce as output the file `test-app/image_v24_signed_and_encrypted.bin`, that can be transferred to the target's external device.

6.6.7 Signing and encrypting the update bundle with AES-256

In case of AES-256, the encryption secret is provided in a binary file that should contain a concatenation of a 32B key and a 16B IV.

In the examples provided, the test application uses the following parameters:

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

So it is easy to prepare the encryption secret in the test scripts or from the command line using:

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

The `sign` tool can now be invoked to produce a signed+encrypted image, by using the extra argument `--encrypt` followed by the secret file. To select AES-256, use the `--aes256` option.

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

which will produce as output the file `test-app/image_v24_signed_and_encrypted.bin`, that can be transferred to the target's external device.

6.6.8 Encryption of incremental (delta) updates

When used in combination with delta updates, encryption works the same way as in full-update mode. The final delta image is encrypted with the selected algorithm.

6.6.9 Encryption of self-updates

When used in combination with bootloader 'self' updates, the encryption algorithm must be configured to run from RAM.

This is done by changing the linker script for the target. At the moment the feature has been successfully tested with the ChaCha algorithm.

The `.text` and `.rodata` segments in FLASH must be updated to not include symbols to be loaded in memory, so the following lines in the .text section:

```
*(.text*)
*(.rodata*)
```

Must be replaced with:

```
*(EXCLUDE_FILE(*chacha.o).text*)
*(EXCLUDE_FILE(*chacha.o).rodata*)
```

Similarly, the .data section loaded in RAM should contain all the .text and .rodata also coming from the symbols of the encryption algorithm. The .data section should have the following added, after `KEEP(*(.ramcode))`:

```
        KEEP(*(.text.wc_Chacha*))
        KEEP(*(.text.rotlFixed*))
        KEEP(*(.rodata.sigma))
        KEEP(*(.rodata.tau))
```

The combination of encryption + self update has been successfully tested on STM32L0. When using makefile based build, a different linker script `hal/$(TARGET)_chacha_ram.ld` is used as template. The file `hal/stm32l0_chacha_ram.ld` contains the changes described above to place all the needed symbols in RAM.

6.6.10 API usage in the application

When transferring the image, the application can still use the libwolfboot API functions to store the encrypted firmware. When called from the application, the function `ext_flash_write` will store the payload unencrypted.

In order to trigger an update, before calling `wolfBoot_update_trigger` it is necessary to set the temporary key used by the bootloader by calling `wolfBoot_set_encrypt_key`.

An example of encrypted update trigger can be found in the stm32wb test application source code (in `../test-app/app_stm32wb.c`).

6.7 Application interface for interactions with the bootloader

wolfBoot offers a small interface to interact with the images stored in the partition, explicitly initiate an update and confirm the success of a previously scheduled update.

6.7.1 Compiling and linking with libwolfboot

An application that requires interactions with wolfBoot must include the header file:

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

This exports the API function declarations, and the predefined values for the flags and tags stored together with the firmware images in the two partitions.

For more information about flash partitions, flags and states see Flash partitions.

6.7.2 API

libwolfboot provides low-level access interface to flash partition states. The state of each partition can be retrieved and altered by the application.

Basic interaction from the application is provided via the following high-level function calls:

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

```
void wolfBoot_update_trigger(void)
```

```
void wolfBoot_success(void)
```

6.7.2.1 Firmware version Current (boot) firmware and update firmware versions can be retrieved from the application using:

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

Or via the shortcut macros:

```
wolfBoot_current_firmware_version()
```

and

`wolfBoot_update_firmware_version()`

6.7.2.2 Trigger an update

- `wolfBoot_update_trigger()` is used to trigger an update upon the next reboot, and it is normally used by an update application that has retrieved a new version of the running firmware, and has stored it in the UPDATE partition on the flash. This function will set the state of the UPDATE partition to `STATE_UPDATING`, instructing the bootloader to perform the update upon the next execution (after reboot).

wolfBoot update process swaps the contents of the UPDATE and the BOOT partitions, using a temporary single-block SWAP space.

6.7.2.3 Confirm current image

- `wolfBoot_success()` indicates a successful boot of a new firmware. This can be called by the application at any time, but it will only be effective to mark the current firmware (in the BOOT partition) with the state `STATE_SUCCESS`, indicating that no roll-back is required. An application should typically call `wolfBoot_success()` only after verifying that the basic system features are up and running, including the possibility to retrieve a new firmware for the next upgrade.

If after an upgrade and reboot wolfBoot detects that the active firmware is still in `STATE_TESTING` state, it means that a successful boot has not been confirmed for the application, and will attempt to revert the update by swapping the two images again.

For more information about the update process, see Firmware Update

For the image format, see Firmware Image

7 Integrating wolfBoot in an existing project

7.1 Required steps

- See the Targets chapter for reference implementation examples.
- Provide a HAL implementation for the target platform (see Hardware Abstraction Layer)
- Decide a flash partition strategy and modify `include/target.h` accordingly (see Flash partitions)
- Change the entry point of the firmware image to account for bootloader presence
- Equip the application with the wolfBoot library to interact with the bootloader
- Configure and compile a bootable image with a single "make" command
- For help signing firmware see wolfBoot Signing
- For enabling measured boot see wolfBoot measured boot

7.2 Examples provided

Additional examples available on our GitHub wolfBoot-examples repository here.

The following steps are automated in the default `Makefile` target, using the baremetal test application as an example to create the factory image. By running `make`, the build system will:

- Create a Ed25519 Key-pair using the keygen tool
- Compile the bootloader. The public key generated in the step above is included in the build
- Compile the firmware image from the test application in the 'test_app' directory
- Re-link the firmware to change the entry-point to the start address of the primary partition
- Sign the firmware image using the `sign` tool
- Create a factory image by concatenating the bootloader and the firmware image

The factory image can be flashed to the target device. It contains the bootloader and the signed initial firmware at the specified address on the flash.

The `sign.py` tool transforms a bootable firmware image to comply with the firmware image format required by the bootloader.

For detailed information about the firmware image format, see Firmware image

For detailed information about the configuration options for the target system, see Compiling wolfBoot

7.3 Upgrading the firmware

- Compile the new firmware image, and link it so that its entry point is at the start address of the primary partition
- Sign the firmware using the `sign.py` tool and the private key generated for the factory image
- Transfer the image using a secure connection, and store it to the secondary firmware slot
- Trigger the image swap using libwolfboot `wolfBoot_update_trigger()` function. See wolfBoot library API for a description of the operation
- Reboot to let the bootloader begin the image swap
- Confirm the success of the update using libwolfboot `wolfBoot_success()` function. See wolfBoot library API for a description of the operation

For more detailed information about firmware update implementation, see Firmware Update

8 Troubleshooting

8.1 Python errors when signing a key

```
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'
```

You need to install the latest wolfcrypt-py here: https://github.com/wolfSSL/wolfcrypt-py

Use `pip3 install wolfcrypt`.

Or to install based on a local wolfSSL installation use:

```
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 Python errors in command line parser running keygen.py

```
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"
```

The version of the python interpreter installed on the system is too old. To run `keygen.py` you need to upgrade python to v.3.8 or greater.

8.3 Contact support

If you run into problems and need help, contact us at *support@wolfssl.com*

A ATA Security

A.1 Introduction

This document provides an overview of how wolfBoot can leaverage the ATA security features to lock or unlock ATA drive. The ATA drive may be locked either by using a hardcoded password or by using a secret that is sealed in the TPM.

A.2 Table of Contents

A.3 Unlocking the Disk with a Hardcoded Password

To unlock the disk using a hardcoded password, use the following options in your .config file:

```
DISK_LOCK=1
DISK_LOCK_PASSWORD=hardcoded_password
```

If the ATA disk has no password set, the disk will be locked with the password provided at the first boot.

A.4 Unlocking the Disk with a TPM-Sealed Secret

wolfBoot allows to seal secret safely in the TPM in a way that it can be unsealed only under specific conditions. Please refer to Appendix M and Appendix Gfor more information. If the options `WOLFBOOT_TPM_SEAL` and `DISK_LOCK` are enabled, wolfBoot will use a TPM sealed secret as the password to unlock the disk. The following options controls the sealing and unsealing of the secret:

Option	Description
WOLFBOOT_TPM_SEAL_KEY_ID	The key ID to use for sign the policy
ATA_UNLOCK_DISK_KEY_NV_INDEX	The NV index to store the sealed secret.
WOLFBOOT_DEBUG_REMOVE_SEALED_ON_ERROR	In case of error, delete the secret and panic()

In case there are no secret sealed at `ATA_UNLOCK_DISK_KEY_NV_INDEX`, a new random secret will be created and sealed at that index. In case the ATA drive is not locked, it will be locked at the first boot with the secret sealed in the TPM.

A.5 Disabling the password

If you need to disable the password, a master password should be already set on the device. Then you can use the following options to compile wolfBoot so that it will disable the password from the drive and panic:

```
WOLFBOOT_ATA_DISABLE_USER_PASSWORD=1
ATA_MASTER_PASSWORD=the_master_password
```

B Signing firmware using Microsoft Azure Key Vault

Microsoft offers secure key management and provisioning tools, using keys stored in HSMs. This mechanisms helps to centralize key management for several purposes, including the support for signing payloads using the managed keys, which can be used in combination with wolfBoot for provisioning public keys in a fleet of devices.

B.1 Preparing the keystore

wolfBoot can import public keys in the keystore using the keygen command line tool provided. keygen supports both raw ECC keys and ASN.1 format (.der).

Azure allows to download the public keys in ASN.1 format to provision the device. To retrieve each public key to use for firmware authentication in wolfBoot, use:

```
az keyvault key download --vault-name <vault-name> -n test-signing-key-1 -e DER
 ↪  -f public-key-1.der
```

A keystore can now be created importing the public keys and with keygen's `-i` (import) option. The option may be repeated multiple times to add more keys to the keystore.

```
./tools/keytools/keygen --ecc256 -i public-key-1.der [-i public-key-2.der ...]
```

B.2 Signing the firmware image for wolfBoot

The signing operation using any external HSM is performed through three-steps, as described in the relevant section in Appendix B. In this section we describe the procedure to sign the firmware image using Azure key vault.

B.2.1 Obtaining the SHA256 digest

Step 1 consists in calling the `./sign` tool with the extra `--sha-only` argument, to generate the digest to sign. The public key associated to the selected signing key in the vault needs to be provided:

```
./tools/keytools/sign --ecc256 --sha-only --sha256 test-app/image.bin
 ↪  public-key-1.der 1
```

To fit in a https REST request, the digest obtained must be encoded using base64:

```
DIGEST=$(cat test-app/image_v1_digest.bin | base64url_encode)
```

The variable `DIGEST` now contains a printable encoding of the key, which can be attached to the request.

B.2.2 HTTPS request for signing the digest with the Key Vault

To prepare the request, first get an access token from the vault and store it in a variable:

```
ACCESS_TOKEN=$(az account get-access-token --resource
 ↪  "https://vault.azure.net" --query "accessToken" -o tsv)
```

Use the URL associated to the selected key vault:

```
KEY_IDENTIFIER="https://<vault-name>.vault.azure.net/keys/test-signing-key"
```

Perform the request using cURL, and store the result in a variable:

```
SIGNING_RESULT=$(curl -X POST \
    -s "${KEY_IDENTIFIER}/sign?api-version=7.4" \
    -H "Authorization: Bearer ${ACCESS_TOKEN}" \
    -H "Content-Type:application/json" \
    -H "Accept:application/json" \
    -d "{\"alg\":\"ES256\",\"value\":\"${DIGEST}\"}")
echo $SIGNING_RESULT
```

The field `.value` in the result contains the (base64 encoded) signature. To extract the signature from the response, you can use a JSON parser:

```
SIGNATURE=$(jq -jn "$SIGNING_RESULT|.value")
```

The signature can now be decoded from base64 into a binary, so the `sign` tool can incorporate the signature into the manifest header.

```
echo $SIGNATURE| base64url_decode > test-app/image_v1_digest.sig
```

B.2.3 Final step: create the signed firmware image

The 'third step' in the HSM three-steps procedure requires the `--manual-sign` option and the signature obtained through the Azure REST API.

```
./tools/keytools/sign --ecc256 --sha256 --manual-sign test-app/image.bin test-
   signin-key_pub.der 1 test-app/image_v1_digest.sig
```

The resulting binary file `image_v1_signed.bin` will now contain a signed firmware image that can be authenticated and staged by wolfBoot.

C Using One-Time Programmable (OTP) flash area for keystore

Some microcontrollers provide a special area in flash memory that can only be written once and cannot be erased.

This feature comes particularly handy when you want to store the public keys required to authenticate the firmware update images, which has exactly the same requirements. A public key is a cryptographic key that can be freely distributed and is used to verify the signature of the firmware update image. By storing the public keys in the OTP area, you can ensure that they are immutable and cannot be tampered with.

C.1 Compiling wolfBoot to access OTP as keystore

To use the OTP area as a keystore, you need to compile wolfBoot with the `FLASH_OTP_KEYSTORE` option enabled. This option is disabled by default, which means that the keystore is incorporated into the wolfBoot binary itself.

When wolfBoot uses the OTP area as a keystore, it reads the public keys from the OTP area at runtime. The public keys are stored in the OTP area, after an initial 16-byte header that contains the number of keys stored, the size of each key, and other information.

In order for wolfBoot to start authenticating the firmware images at boot and upon update, the public keys must be provisioned to the OTP area in a separate step, as described in the next sections.

Depending on the target device, you can either prepare a binary image of the OTP area content, or use `otp-keystore-primer` firmware to directly provision the keys on the target.

C.2 Creating an image of the OTP area content

It is possible to create a binary image of the content for the OTP area. The resulting file (otp.bin) can be manually provisioned using any external tool that allows writing to the target OTP area.

To compile the otp-keystore-gen tool using the current keystore content:

```
make otpgen
```

And then, to create the image file `otp.bin`:

```
./tools/keytools/otp/otp-keystore-gen
```

C.3 Directly provisioning the public keys to the OTP area (primer)

After enabling the `FLASH_OTP_KEYSTORE` option in your `.config` file, when you compile wolfBoot by running “make”, an additional application called `otp-keystore-primer` is generated under `tools/keytools/otp`. This application is used to provision the public keys to the OTP area. By flashing this application to the microcontroller, the public keys contained in your keystore (previously generated by keygen) are written to the OTP area.

The `otp-keystore-primer` application is generated with the public keys embedded in it. The keys are retrieved from the `keystore.c` file, generated by the keygen command. The `otp-keystore-primer` application reads the public keys from the `keystore.c` file and writes them to the OTP area.

After generating a new `keystore.c` with the keygen application, you can generate the `otp-keystore-primer` application again, by running `make otp`.

> [!WARNING] The `otp-keystore-primer` application is a one-time use application. Once the application runs on your target, the public keys are written to the OTP area, and it will be impossible to erase them. Therefore, it is important to ensure that the public keys are

correct before provisioning them to the OTP area, and that the associated private keys are stored securely. Accidentally losing the private keys will render the public keys stored in the OTP area useless.

[!CAUTION] ** Be very careful when using the `otp-keystore-primer` application. Use it at your own risk. **

C.4 Examples

C.4.1 STM32H5 OTP KeyStore

Example for NULCLEO-STM32H563ZI with TrustZone (via PKCS11), DualBank and signing with PQ LMS:

1) Setup the configuration and key tools:

```
cp config/examples/stm32h5-tz-dualbank-otp-lms.config .config
make include/target.h
make keytools
```

2) Generate key(s) to write to OTP

- `./examples/keytools/keygen --lms -g 1.key -g 2.key -g 3.key -g 4.key -g 5.key`

3) Backup the generated keys and `src/keystore.c`

- Save to safe place outside of the wolfBoot tree

4) Set the signing key to use

- Copy one of the generated keys to `wolfboot_signing_private_key.der`
- `cp 1.key wolfboot_signing_private_key.der`

5) Setup the OTP keystore

Flash the OTP keystore primer: - Run `make otp` - Flash `./tools/keytools/otp/otp-keystore-primer.bin` to `0x08000000` - Disconnect the tool and hit reset button - The primer will run and flash keystore.c to OTP and enable write protection on those blocks

OR

Generate OTP (otp.bin) and flash using external tool - Run `make otpgen` - Run `./tools/keytools/otp/otp-keystore-gen` to generate an otp.bin file - Program otp.bin to `0x08FFF000` using external tool like STM32CubeProgrammer

6) Verify OTP keystore

- Read memory at address `0x08FFF000` (should start with ASCII “WOLFBOOT”)
- Typically use STM32CubeProgrammer for this

7) Setup the option bytes

- User Configuration 2 -> TrustZone Enable (TZEN=0xB4)
- Bank1 - Flash Watermark area (SECWM1_START=0x00, SECWM1_END=0x1F)
- Bank2 - Flash Watermark area (SECWM2_START=0x00, SECWM2_END=0x1F)

8) Mass erase the device

- STM32CubeProgrammer -> Full chip erase

9) Build wolfBoot and test application using `make`

10) Flash wolfBoot and test-app

- Flash `wolfboot.bin` at `0x0C000000`
- Flash `test-app/image_v1_signed.bin` at `0x08040000`

11) Disconnect and reboot, the red LED should turn on.
12) Connect to USB UART on NUCLEO board for console

Explore the command line (run help)

```
========================
STM32H5 wolfBoot demo Application
Copyright 2024 wolfSSL Inc
GPL v3
Version : 0x1
========================

cmd> help
help : shows this help message
info : display information about the system and partitions
success : confirm a successful update
pkcs11 : enable and test crypto calls with PKCS11 in secure mode
random : generate a random number
timestamp : print the current timestamp
benchmark : run the wolfCrypt benchmark
test : run the wolfCrypt test
update : update the firmware via XMODEM
reboot : reboot the system
```

13) Test Update

- Sign a new version of the firmware: `./tools/keytools/sign --lms test-app/image.bin wolfboot_signing_private_key.der 2`
- Run "update" command on the shell and wait for xmodem transfer
- Use serial terminal that supports xmodem like "minicom" or "CoolTerm".
  - Run `minicom` on `/dev/ttyACM0` and start file transfer using "CTRL+A; S"
  - Select xmodem then navigate to the new signed firmware file `test-app/image_v2_signed.bin`
- During the transfer, the yellow LED will flash.
- The green LED is dim because it's sync with the UART RX
- At the end of the transfer, the new image will be in the update partition.
- Reset board to install new firmware and confirm new version number.

Example update output:

```
cmd> update
Erasing update partition...Done.
Waiting for XMODEM transfer...
.........................................................................................




End of transfer. ret: 0
New firmware version: 0x2
Triggering update...
Update completed successfully.

cmd> reboot

========================
STM32H5 wolfBoot demo Application
Copyright 2024 wolfSSL Inc
```

```
GPL v3
Version : 0x2
========================

cmd>
```

D KeyStore structure: support for multiple public keys

D.1 What is wolfBoot KeyStore

KeyStore is the mechanism used by wolfBoot to store all the public keys used for authenticating the signature of current firmware and updates.

wolfBoot's key generation tool can be used to generate one or more keys. By default, when running `make` for the first time, a single key `wolfboot_signing_private_key.der` is created, and added to the keystore module. This key should be used to sign any firmware running on the target, as well as firmware update binaries.

Additionally, the keygen tool creates additional files with different representations of the keystore - A .c file (src/keystore.c) which can be used to deploy public keys as part of the bootloader itself, by linking the keystore in `wolfboot.elf` - A .bin file (keystore.bin) which contains the keystore that can be hosted on a custom memory support. In order to access the keystore, a small driver is required (see section "Interface API" below).

D.2 Default usage (built-in keystore)

By default, the keystore object in `src/keystore.c` is accessed by wolfboot by including its symbols in the build. Once generated, this file contains an array of structures describing each public key that will be available to wolfBoot on the target system. Additionally, there are a few functions that connect to the wolfBoot keystore API to access the details and the content of the public key slots.

The public key is described by the following structure:

```
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` is the incremental identifier for the key slot, starting from 0.
- `key_type` describes the algorithm of the key, e.g. `AUTH_KEY_ECC256` or `AUTH_KEY_RSA3072`
- `mask` describes the permissions for the key. It's a bitmap of the partition ids for which this key can be used for verification
- `pubkey_size` the size of the public key buffer
- `pubkey` the actual buffer containing the public key in its raw format

When booting, wolfBoot will automatically select the public key associated to the signed firmware image, check that it matches the permission mask for the partition id where the verification is running and then attempts to authenticate the signature of the image using the selected public key slot.

D.2.1 Creating multiple keys

keygen accepts multiple filenames for private keys.

Two arguments:

- `-g priv.der` generate new keypair, store the private key in priv.der, add the public key to the keystore
- `-i pub.der` import an existing public key and add it to the keystore

Example of creation of a keystore with two ED25519 keys:

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

will create the following files:

- `first.der` first private key
- `second.der` second private key
- `src/keystore.c` C keystore containing both public keys associated with `first.der` and `second.der`.

The `keystore.c` generated should look similar to this:

```
#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
        },
    },


};
```

D.2.2 Permissions

By default, when a new keystore is created, the permissions mask is set to `KEY_VERIFY_ALL`, which means that the key can be used to verify a firmware targeting any partition id.

The `part_id_mask` value is a bitmask, where each bit represent a different partition. The bit '0' is reserved for wolfBoot self-update, while typically the main firmware partition is associated to id 1, so it requires a key with the bit '1' set. In other words, signing a partition with `--id 3` would require turning on bit '3' in the mask, i.e. adding (1U « 3) to it.

To restrict the permissions for single keys, it would be sufficient to change the value of each key `part_id_mask`. This is done via the `--id` command line option for keygen. Each generated or imported key can be associated with a number of partition by passing the partition IDs in a comma-separated list, e.g.:

```
keygen --ecc256 -g generic.key --id 1,2,3 -g restricted.key
```

Generates two keypairs, `generic.key` and `restricted.key`. The former assumes the default mask `KEY_VERIFY_ALL`, which makes it possible to use it to authenticate any of the system components. The latter instead, will carry a mask with only the bits '1', '2', and '3' set (mask = b00001110 =0x000e), allowing the usage only with the assigned partition IDs.

D.2.3 Importing public keys

The "-i" option is used to import existing public keys into the keyvault. The usage is identical to the '-g' option, except that the file provided must exist and contain a valid public key of the given algorithm and key size.

D.2.4 Generating and importing keys of different types

By default, wolfBoot hardcodes the type of key used for all the signature verification operations into the keystore format.

Alternatively, wolfBoot can be compiled with the option `WOLFBOOT_UNIVERSAL_KEYSTORE=1`, which disables the check at compile time and allows adding keys of different types to the keystore. For example, if we want to create two keypairs with different ECC curves, and additionally store a pre-existing RSA2048 public key file `rsa-pub.der`, we could run the following:

```
keygen --ecc256 -g a.key  --ecc384 -g b.key --rsa2048 -i rsa-pub.der
```

The command above generates a keystore with three public keys that are accessible by the bootloader at runtime.

Please note that by default wolfBoot does not include any public key algorithm implementations besides the one selected via the option `SIGN=`, so usually this feature is reserved to specific use cases where other policies or components in the chain-of-trust require to store different key types for different purposes.

D.3 Using KeyStore with external Key Vaults

It is possible to use an external NVM, a Key Vault or any generic support to access the KeyStore. In this case, wolfBoot should not link the generated keystore.c directly, but rather rely on an external interface, that exports the same API which would be implemented by `keystore.c`.

The API consists of a few functions described below.

D.3.1 Interface API

D.3.1.1 Number of keys in the keystore `int keystore_num_pubkeys(void)`

Returns the number of slots in the keystore. At least one slot should be populated if you want to authenticate your firmware today. The interface assumes that the slots are numbered sequentially, from zero to `keystore_num_pubkeys() - 1`. Accessing those slots through this API should always return a valid public key.

D.3.1.2 Size of the public key in a slot `int keystore_get_size(int id)`

Returns the size of the public key stored in the slot `id`. In case of error, return a negative value.

D.3.1.3 Actual public key buffer (mapped/copied in memory) `uint8_t *keystore_get_buffer(int id)`

Returns a pointer to an accessible area in memory, containing the buffer with the public key associated to the slot `id`.

D.3.1.4 Permissions mask `uint32_t keystore_get_mask(int id)`

Returns the permissions mask, as a 32-bit word, for the public key stored in the slot `id`.

E Build wolfBoot as Library

Instead of building as standalone repository, wolfBoot can be built as a secure-boot library and integrated in third party bootloaders, custom staging solutions etc.

E.1 Library API

The wolfBoot secure-boot image verification has a very simple interface. The core object describing the image is a `struct wolfBoot_image`, which is initialized when `wolfBoot_open_image_address()` is called. The signature is:

```
int wolfBoot_open_image_address(struct wolfBoot_image* img, uint8_t* image)
```

where `img` is a pointer to a local (uninitialized) structure of type `wolfBoot_image`, and `image` is a pointer to where the signed image is mapped in memory, starting from the beginning of the manifest header.

On success, zero is returned. If the image does not contain a valid 'magic number' at the beginning of the manifest, or if the size of the image is bigger than `WOLFBOOT_PARTITION_SIZE`, -1 is returned.

If the `open_image_address` operation is successful, two other functions can be invoked:

- `int wolfBoot_verify_integrity(struct wolfBoot_image *img)`

This function verifies the integrity of the image, by calculating the SHA hash of the image content, and comparing it with the digest stored in the manifest header. `img` is a pointer to an object of type `wolfBoot_image`, previously initialized by `wolfBoot_open_image_address`.

0 is returned if the image integrity could be successfully verified, -1 otherwise.

- `int wolfBoot_verify_authenticity(struct wolfBoot_image *img)`

This function verifies that the image content has been signed by a trusted counterpart (i.e. that could be verified using one of the public keys available).

`0` is returned in case of successful authentication, `-1` if anything went wrong during the operation, and `-2` if the signature could be found, but was not possible to authenticate against its public key.

E.2 Library mode: example application

An example application is provided in `hal/library.c`.

The application `test-lib` opens a file from a path passed as argument from the command line, and verifies that the file contains a valid, signed image that can be verified for integrity and authenticity using wolfBoot in library mode.

E.3 Configuring and compiling the test-lib application

Step 1: use the provided configuration to compile wolfBoot in library mode:

```
cp config/examples/library.config .config
```

Step 2: create a file `target.h` that only contains the following lines:

```
cat > include/target.h << EOF
#ifndef H_TARGETS_TARGET_
#define H_TARGETS_TARGET_

#define WOLFBOOT_NO_PARTITIONS
```

```
#define WOLFBOOT_SECTOR_SIZE                0x20000
#define WOLFBOOT_PARTITION_SIZE             0x20000

#endif /* !H_TARGETS_TARGET_ */

EOF
```

Change `WOLFBOOT_PARTITION_SIZE` accordingly. `wolfBoot_open_image_address()` will discard images larger than `WOLFBOOT_PARTITION_SIZE - IMAGE_HEADER_SIZE`.

Step 3: compile keytools and create keys.

```
make keytools
./tools/keytools/keygen --ed25519 -g wolfboot_signing_private_key.der
```

Step 4: Create an empty file and sign it using the private key.

```
touch empty
./tools/keytools/sign --ed25519 --sha256 empty wolfboot_signing_private_key.
   der 1
```

Step 5: compile the test-lib application, linked with wolfBoot in library mode, and the public key from the keypair created at step 4.

```
make test-lib
```

Step 6: run the application with the signed image

```
./test-lib empty_v1_signed.bin
```

If everything went right, the output should be similar to:

```
Firmware Valid
booting 0x5609e3526590(actually exiting)
```

F wolfBoot Loaders / Updaters

F.1 loader.c

The default wolfBoot loader entry point that starts the wolfBoot secure boot process and leverages one of the `*_updater.c` implementations.

F.2 loader_stage1.c

A first stage loader whose purpose is to load wolfBoot from flash to ram and jump to it. This is required on platforms where flash is not memory mapped (XIP). For example on PowerPC e500v2 where external NAND flash is used for boot only a small 4KB region is available, so wolfBoot must be loaded to RAM and then run.

Example: `make WOLFBOOT_STAGE1_LOAD_ADDR=0x1000 stage1`

- `WOLFBOOT_STAGE1_SIZE`: Maximum size of wolfBoot stage 1 loader
- `WOLFBOOT_STAGE1_FLASH_ADDR`: Location in Flash for stage 1 loader (XIP from boot ROM)
- `WOLFBOOT_STAGE1_BASE_ADDR`: Address in RAM to load stage 1 loader to
- `WOLFBOOT_STAGE1_LOAD_ADDR`: Address in RAM to load wolfBoot to
- `WOLFBOOT_LOAD_ADDRESS`: Address in RAM to load application partition

F.3 update_ram.c

Implementation for RAM based updater

F.4 update_flash.c

Implementation for Flash based updater

F.5 update_flash_hwswap.c

Implementation for hardware assisted updater

G Measured Boot using wolfBoot

wolfBoot offers a simplified measured boot implementation, a way to record and track the state of the system boot process using a Trusted Platform Module(TPM).

This record is tamper-proofed by special registers in the TPM called Platform Configuration Register. Then, the firmware application, RTOS or rich OS(Linux), can access that log of information by reading the PCRs of the TPM.

wolfBoot can interact with TPM2.0 chips thanks to its integration with wolfTPM. wolfTPM has native support for Microsoft Windows and Linux, and can be used standalone or together with wolfBoot. The combination of wolfBoot with wolfTPM gives the developer a tamper-proof secure storage for protecting the system during and after boot.

G.1 Concept

Typically, systems use Secure Boot to guarantee that the correct and genuine firmware is booted by verifying its signature. Afterwards, this knowledge is unknown to the system. The application does not know if the system started in a good known state. Sometimes, this guarantee is needed by the firmware itself. To provide such mechanism the concept of Measured Boot exist.

Measured Boot can be used to check every start-up component, including settings and user information(user partition). The result of the checks is then stored into special registers called PCR. This process is called PCR Extend and is referred to as a TPM measurement. PCR registers can be reset only on TPM power-on.

Having TPM measurements provide a way for the firmware or Operating System(OS), like Windows or Linux, to know that the software loaded before it gained control over system, is trustworthy and not modified.

In wolfBoot the concept is simplified to measuring a single component, the main firmware image. However, this can easily be extended by using more PCR registers.

G.2 Configuration

To enable measured boot add `MEASURED_BOOT=1` setting in your wolfBoot config.

It is also necessary to select the PCR (index) where the measurement will be stored.

Selection is made using the `MEASURED_BOOT_PCR_A=[index]` setting. Add this setting in your wolfBoot config and replace `[index]` with a number between 0 and 23. Below you will find guidelines for selecting a PCR index.

Any TPM has a minimum of 24 PCR registers. Their typical use is as follows:

Index	Typical use	Recommended to use with
0	Core Root of Trust and/or BIOS measurement	bare-metal, RTOS
1	measurement of Platform Configuration Data	bare-metal, RTOS
2-3	Option ROM Code measurement	bare-metal, RTOS
4-5	Master Boot Record measurement	bare-metal, RTOS
6	State Transitions	bare-metal, RTOS
7	Vendor specific	bare-metal, RTOS
8-9	Partition measurements	bera-metal, RTOS
10	measurement of the Boot Manager	bare-metal, RTOS
11	Typically used by Microsoft Bitlocker	bare-metal, RTOS
12-15	Available for any use	bare-metal, RTOS, Linux, Windows

Index	Typical use	Recommended to use with
16	DEBUG	Use only for test purposes
17	DRTM	Trusted Bootloader
18-22	Trusted OS	Trusted Execution Environment(TEE)
23	Application	Use only for temporary measurements

Recommendations for choosing a PCR index:

- During development it is recommended to use PCR16 that is intended for testing.
- In production, if you are running a bare-metal firmware or RTOS, you could use almost all PCRs(PCR0-15), except the one for DRTM and Trusted OS(PCR17-23).
- If you are running Linux or Windows, PCR12-15 can be chosen for production ready firmware, in order to avoid conflict with other software that might be using PCRs from within Linux, like the Linux IMA or Microsoft Bitlocker.

Here is an example part of a wolfBoot .config during development:

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

G.2.1 Code

wolfBoot offers out-of-the-box solution. There is zero need of the developer to touch wolfBoot code in order to use measured boot. If you would want to check the code, then look in `src/image.c` and more specifically the `measure_boot()` function. There you would find several TPM2 native API calls to wolfTPM. For more information about wolfTPM you can check its GitHub repository.

H Post-Quantum Signatures

wolfBoot is adding support for post-quantum signatures. At present, support for LMS/HSS, and XMSS/XMSS^MT has been added.

LMS/HSS and XMSS/XMSS^MT are both post-quantum stateful hash-based signature (HBS) schemes. They are known for having small public keys, relatively fast signing and verifying operations, but larger signatures. Their signature sizes however are tunable via their different parameters, which affords a space-time tradeoff.

Stateful HBS schemes are based on the security of their underlying hash functions and Merkle trees, which are not expected to be broken by the advent of cryptographically relevant quantum computers. For this reason they have been recommended by both NIST SP 800-208, and the NSA's CNSA 2.0 suite.

See these links for more info on stateful HBS support and wolfSSL/wolfCrypt:

- https://www.wolfssl.com/documentation/manuals/wolfssl/appendix07.html#post-quantum-stateful-hash-based-signatures
- https://github.com/wolfSSL/wolfssl-examples/tree/master/pq/stateful_hash_sig

H.1 Supported PQ Signature Methods

These four PQ signature options are supported:

- LMS: uses wolfcrypt implementation from `wc_lms.c`, and `wc_lms_impl.c`.
- XMSS: uses wolfcrypt implementation from `wc_xmss.c`, and `wc_xmss_impl.c`.
- ext_LMS: uses external integration from `ext_lms.c`.
- ext_XMSS: uses external integration from `ext_xmss.c`.

The wolfcrypt implementations are more performant and are recommended. The external integrations are experimental and for testing interoperability.

H.1.1 LMS/HSS Config

A new LMS sim example has been added here:

`config/examples/sim-lms.config`

The `LMS_LEVELS`, `LMS_HEIGHT`, and `LMS_WINTERNITZ`, `IMAGE_SIGNATURE_SIZE`, and (optionally) `IMAGE_HEADER_SIZE` must be set:

```
SIGN?=LMS
...
LMS_LEVELS=2
LMS_HEIGHT=5
LMS_WINTERNITZ=8
...
IMAGE_SIGNATURE_SIZE=2644
IMAGE_HEADER_SIZE?=5288
```

In LMS the signature size is a function of the parameters. Use the added helper script `tools/lms/lms_siglen.sh` to calculate your signature length given your LMS parameters:

```
$ ./tools/lms/lms_siglen.sh  2 5 8
levels:     2
height:     5
winternitz: 8
signature length: 2644
```

H.1.2 XMSS/XMSS^MT Config

A new XMSS sim example has been added here:

`config/examples/sim-xmss.config`

The `XMSS_PARAMS`, `IMAGE_SIGNATURE_SIZE`, and (optionally) `IMAGE_HEADER_SIZE` must be set:

```
SIGN?=XMSS
...
XMSS_PARAMS='XMSS-SHA2_10_256'
...
IMAGE_SIGNATURE_SIZE=2500
IMAGE_HEADER_SIZE?=5000
```

The `XMSS_PARAMS` may be any SHA256 parameter set string from Tables 10 and 11 from NIST SP 800-208. Use the helper script `tools/xmss/xmss_siglen.sh` to calculate your signature length given your XMSS/XMSS^MT parameter string, e.g.:

```
$ ./tools/xmss/xmss_siglen.sh XMSS-SHA2_10_256
parameter set:    XMSS-SHA2_10_256
signature length: 2500

$ ./tools/xmss/xmss_siglen.sh XMSSMT-SHA2_20/2_256
parameter set:    XMSSMT-SHA2_20/2_256
signature length: 4963
```

H.2 Building the external PQ Integrations

H.2.1 ext_LMS Support

The external LMS/HSS support in wolfCrypt requires the hash-sigs library (https://github.com/cisco/hash-sigs). Use the following procedure to prepare hash-sigs for building with wolfBoot:

```
$ cd lib
$ mkdir hash-sigs
$ ls
 CMakeLists.txt  hash-sigs  wolfssl  wolfTPM
$ cd hash-sigs
$ mkdir lib
$ git clone https://github.com/cisco/hash-sigs.git src
$ cd src
$ git checkout b0631b8891295bf2929e68761205337b7c031726
$ git apply ../../../tools/lms/0001-Patch-to-support-wolfBoot-LMS-build.patch
```

Nothing more is needed, as wolfBoot will automatically produce the required hash-sigs build artifacts.

Note: the hash-sigs project only builds static libraries: - hss_verify.a: a single-threaded verify-only static lib. - hss_lib.a: a single-threaded static lib. - hss_lib_thread.a: a multi-threaded static lib.

The keytools utility links against `hss_lib.a`, as it needs full keygen, signing, and verifying functionality. However wolfBoot links directly with the subset of objects in the `hss_verify.a` build rule, as it only requires verify functionality.

H.2.2 ext_XMSS Support

The external XMSS/XMSS^MT support in wolfCrypt requires a patched version of the xmss-reference library. Use the following procedure to prepare xmss-reference for building with wolfBoot:

```
$ cd lib
$ git clone https://github.com/XMSS/xmss-reference.git xmss
$ ls
CMakeLists.txt  wolfPKCS11  wolfTPM  wolfssl  xmss
$ cd xmss
$ git checkout 171ccbd26f098542a67eb5d2b128281c80bd71a6
$ git apply ../../tools/xmss/0001-Patch-to-support-wolfSSL-xmss-reference-
   integration.patch
```

The patch creates an addendum readme, `patch_readme.md`, with further comments.

Nothing more is needed beyond the patch step, as wolfBoot will handle building the xmss build artifacts it requires.

I Remote External flash memory support via UART

wolfBoot can emulate external partitions using UART communication with a neighbor system. This feature is particularly useful in those asynchronous multi-process architectures, where updates can be stored with the assistance of an external processing unit.

I.1 Bootloader setup

The option to activate this feature is UART_FLASH=1. This configuration option depends on the external flash API, which means that the option EXT_FLASH=1 is also mandatory to compile the bootloader.

The HAL of the target system must be expanded to include a simple UART driver, that will be used by the bootloader to access the content of the remote flash using one of the UART controllers on board.

Example UART drivers for a few of the supported platforms can be found in the `hal/uart` directory.

The API exposed by the UART HAL extension for the supported targets is composed by the following functions:

```
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);
```

Consider implementing these three functions based on the provided examples if you want to use external flash memory support on your platform, if not officially supported yet.

I.2 Host side: UART flash server

On the remote system hosting the external partition image for the target, a simple protocol can be implemented on top of UART messages to serve flash-access specific calls.

An example uart-flash-server daemon, designed to run on a GNU/Linux host and emulate the external partition with a local file on the filesystem, is available in `tools/uart-flash-server`.

I.3 External flash update mechanism

wolfBoot treats external UPDATE and SWAP partitions in the same way as when they are mapped on a local SPI flash. Read and write operations are simply translated into remote procedure calls via UART, that can be interpreted by the remote application and provide read and write access to actual storage elements which would only be accessible by the host.

This means that after a successful update, a copy of the previous firmware will be stored in the remote partition to provide exactly the same update mechanism that is available in all the other use cases. The only difference consist in the way of accessing the physical storage area, but all the mechanisms at a higher level stay the same.

J Renesas wolfBoot

Platforms Supported:

- Renesas RZ (RZN2L) (RSIP)
  - #renesas-rzn2l
  - IDE/Renesas/e2studio/RZN2L/Readme.md
  - IDE/Renesas/e2studio/RZN2L/Readme_wRSIP.md
- Renesas RA (RA6M4) (SCE)
  - #renesas-ra6m4
  - IDE/Renesas/e2studio/RA6M4/Readme.md
  - IDE/Renesas/e2studio/RA6M4/Readme_withSCE.md
- Renesas RX (RX65N/RX72N) (TSIP)
  - #renesas-rx72n
  - IDE/Renesas/e2studio/RX72N/Readme.md
  - IDE/Renesas/e2studio/RX72N/Readme_withTSIP.md

All of the Renesas examples support using e2Studio. The Renesas RX parts support using wolfBoot Makefile's with the rx-elf-gcc cross-compiler and example .config files.

J.1 Security Key Management Tool (SKMT) Key Wrapping

1) Setup a Renesas KeyWrap account and do the PGP key exchange. https://dlm.renesas.com/keywrap You will get a public key from Renesas `keywrap-pub.key` that needs imported to PGP/GPG. Note: You cannot use RSA 4096-bit key, must be RSA-2048 or RSA-3072.
2) Using "Security Key Management Tool" create 32-byte UFPK (User Factory Programming Key). This can be a random 32-byte value. Example: Random 32-bytes `B94A2B96 1C755101 74F0C967 ECFC20B3 77C7FB25 6DB627B1 BFFADEE0 5EE98AC4`
3) Sign and Encrypt the 32-byte binary file with PGP the `sample.key`. Result is `sample.key.gpg`. Use GPG4Win and the Sign/Encrypt option. Sign with your own GPG key and encrypt with the Renesas public key.
4) Use https://dlm.renesas.com/keywrap to wrap `sample.key.gpg`. It will use the Hidden Root Key (HRK) that both Renesas and the RX TSIP have pre-provisioned from Renesas Factory. Result is `sample.key_enc.key`. Example: `00000001 6CCB9A1C 8AA58883 B1CB02DE 6C37DA60 54FB94E2 06EAE720 4D9CCF4C 6EEB288C`

J.2 RX TSIP

1) Build key tools for Renesas

```
# Build keytools for Renesas RX (TSIP)
$ make keytools RENESAS_KEY=2
```

2) wolfBoot public key (create or import existing)

Instructions below for ECDSA P384 (SECP384R1). For SECP256R1 replace "ecc384" with "ecc256" and "secp384r1" with "secp256r1".

Create new signing key:

```
# Create new signing key
$ ./tools/keytools/keygen --ecc384 -g ./pri-ecc384.der
Keytype: ECC384
Generating key (type: ECC384)
Associated key file:   ./pri-ecc384.der
```

```
Partition ids mask:   ffffffff
Key type   :           ECC384
Public key slot:       0
Done.

# Export public portion of key as PEM
$ openssl ec -inform der -in ./pri-ecc384.der -pubout -out ./pub-ecc384.pem
```

OR

Import Public Key:

```
# Export public portion of key as DER
$ openssl ec -inform der -in ./pri-ecc384.der -pubout -outform der -out
 ↪  ./pub-ecc384.der

# Import public key and populate src/keystore.c
$ ./tools/keytools/keygen --ecc384 -i ./pub-ecc384.der
Keytype: ECC384
Associated key file:   ./pub-ecc384.der
Partition ids mask:   ffffffff
Key type   :           ECC384
Public key slot:       0
Done.
```

3) Create wrapped public key (code files)

Use the Security Key Management Tool (SKMT) command line tool (CLI) to create a wrapped public key.

This will use the user encryption key to wrap the public key and output key_data.c / key_data.h files.

```
$ C:\Renesas\SecurityKeyManagementTool\cli\skmt.exe -genkey -ufpk
 ↪  file=./sample.key -wufpk file=./sample.key_enc.key -key
 ↪  file=./pub-ecc384.pem -mcu RX-TSIP -keytype secp384r1-public -output
 ↪  include/key_data.c -filetype csource -keyname enc_pub_key
Output File: include\key_data.h
Output File: include\key_data.c
UFPK: B94A2B961C75510174F0C967ECFC20B377C7FB256DB627B1BFFADEE05EE98AC4
W-UFPK:
 ↪  000000016CCB9A1C8AA58883B1CB02DE6C37DA6054FB94E206EAE7204D9CCF4C6EEB288C
IV: 6C296A040EEF5EDD687E8D3D98D146D0
Encrypted key:
 ↪  5DD8D7E59E6AC85AE340BBA60AA8F8BE56C4C1FE02340C49EB8F36DA79B8D6640961FE9EAECDD6BADF083C5
```

4) Create wrapped public key (flash file)

Generate Motorola HEX file to write wrapped key to flash.

```
$ C:\Renesas\SecurityKeyManagementTool\cli\skmt.exe -genkey -ufpk
 ↪  file=./sample.key -wufpk file=./sample.key_enc.key -key
 ↪  file=./pub-ecc384.pem -mcu RX-TSIP -keytype secp384r1-public -output
 ↪  pub-ecc384.srec -filetype "mot" -address FFFF0000
Output File: Y:\GitHub\wolfboot\pub-ecc384.srec
UFPK: B94A2B961C75510174F0C967ECFC20B377C7FB256DB627B1BFFADEE05EE98AC4
W-UFPK:
 ↪  000000016CCB9A1C8AA58883B1CB02DE6C37DA6054FB94E206EAE7204D9CCF4C6EEB288C
IV: 9C13402DF1AF631DC2A10C2424182601
Encrypted key:
 ↪  C4A0B368552EB921A3AF3427FD7403BBE6CB8EE259D6CC0692AA72D46F7343F5FFE7DA97A1C811B21BF392E
```

The generated file is a Motorola HEX (S-Record) formatted image containing the wrapped public key with instructions to use the 0xFFFF0000 address.

```
S00E00007075622D65636333737265D5
S315FFFF000000000000000000006CCB9A1C8AA58883C5
S315FFFF0010B1CB02DE6C37DA6054FB94E206EAE720E7
S315FFFF00204D9CCF4C6EEB288C9C13402DF1AF631D7F
S315FFFF0030C2A10C2424182601C4A0B368552EB921EA
S315FFFF0040A3AF3427FD7403BBE6CB8EE259D6CC06AE
S315FFFF005092AA72D46F7343F5FFE7DA97A1C811B27D
S315FFFF00601BF392E3834B67C3CE6F84707CCB8923ED
S315FFFF0070D4FBB8DA003EF23C1CD785B6F58E5DB1F0
S315FFFF008061F575F78D646434AC2BFAF207F6FFF66C
S315FFFF0090363C800CFF7E7BFF4857452A70C496B6D9
S311FFFF00A075D08DD6924CAB5ED6FF44C5E3
S705FFFF0000FC
```

The default flash memory address is 0xFFFF0000, but it can be changed. The following two places must be set: a) The user_settings.h build macro RENESAS_TSIP_INSTALLEDKEY_ADDR b) The linker script .rot section (example hal/rx72n.ld or hal/rx65n.ld).

5) Edit .config PKA?=1.

6) Rebuild wolfBoot. make clean && make wolfboot.srec

7) Sign application

Sign application using the created private key above pri-ecc384.der:

```
$ ./tools/keytools/sign --ecc384 --sha256 test-app/image.bin pri-ecc384.der 1
wolfBoot KeyTools (Compiled C version)
wolfBoot version 2010000
Update type:          Firmware
Input image:          test-app/image.bin
Selected cipher:      ECC384
Selected hash  :      SHA256
Public key:           pri-ecc384.der
Output  image:        test-app/image_v1_signed.bin
Target partition id : 1
image header size overridden by config value (1024 bytes)
Calculating SHA256 digest...
Signing the digest...
Output image(s) successfully created.
```

8) Flash wolfboot.srec, pub-ecc384.srec and signed application binary

Download files to flash using Renesas flash programmer.

J.2.1 RX TSIP Benchmarks

Hardware	Clock	Algorithm	RX TSIP	Debug	Release (-Os)	Release (-O2)
RX72N	240MHz	ECDSA Verify P384	17.26 ms	1570 ms	441 ms	313 ms
RX72N	240MHz	ECDSA Verify P256	2.73 ms	469 ms	135 ms	107 ms
RX65N	120MHz	ECDSA Verify P384	18.57 ms	4213 ms	2179 ms	1831 ms
RX65N	120MHz	ECDSA Verify P256	2.95 ms	1208 ms	602 ms	517 ms

K wolfBoot Key Tools

keygen and `sign` are two command line tools to be used on a PC (or automated server) environment to manage wolfBoot private keys and sign the initial firmware and all the updates for the target.

K.1 C or Python

The tools are distributed in two versions, using the same command line syntax, for portability reasons.

By default, C keytools are compiled. The makefiles and scripts in this repository will use the C tools.

K.1.1 C Key Tools

A standalone C version of the key tools is available in: `./tools/keytools`.

These can be built in `tools/keytools` using `make` or from the wolfBoot root using `make keytools`.

If the C version of the key tools exists they will be used by wolfBoot's makefile and scripts.

K.1.1.1 Windows Visual Studio Use the `wolfBootSignTool.vcxproj` Visual Studio project to build the `sign.exe` and `keygen.exe` tools for use on Windows.

If you see any error about missing `target.h` this is a generated file based on your .config using the make process. It is needed for `WOLFBOOT_SECTOR_SIZE` used in delta updates.

K.1.2 Python key tools

Please note that the Python tools are deprecated and will be removed in future versions.

In order to use the python key tools, ensure that the `wolfcrypt` package is installed in your python environment. In most systems it's sufficient to run a command similar to:

```
pip install wolfcrypt
```

to ensure that the dependencies are met.

K.2 Command Line Usage

K.2.1 Keygen tool

Usage: `keygen [OPTIONS] [-g new-keypair.der] [-i existing-pubkey.der] [...]`

keygen is used to populate a keystore with existing and new public keys. Two options are supported:

- `-g privkey.der` to generate a new keypair, add the public key to the keystore and save the private key in a new file `privkey.der`
- `-i existing.der` to import an existing public key from `existing.der`
- `--der` save generated private key in DER format.

Arguments are not exclusive, and can be repeated more than once to populate a keystore with multiple keys.

One option must be specified to select the algorithm enabled in the keystore (e.g. `--ed25519` or `--rsa3072`). See the section "Public key signature options" for the sign tool for the available options.

The files generate by the keygen tool is the following:

- A C file `src/keystore.c`, which is normally linked with the wolfBoot image, when the keys are provisioned through generated C code.

- A binary file `keystore.img` that can be used to provision the public keys through an alternative storage
- The private key, for each `-g` option provided from command line

For more information about the keystore mechanism, see Appendix D.

K.2.2 Sign tool

`sign` produces a signed firmware image by creating a manifest header in the format supported by wolfBoot.

Usage: `sign [OPTIONS] IMAGE.BIN KEY.DER VERSION`

`IMAGE.BIN`: A file containing the binary firmware/software to sign `KEY.DER`: Private key file, in DER format, to sign the binary image `VERSION`: The version associated with this signed software `OPTIONS`: Zero or more options, described below

K.2.2.1 Public key signature options If none of the following arguments is given, the tool will try to guess the key size from the format and key length detected in KEY.DER.

- `--ed25519` Use ED25519 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--ed448` Use ED448 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--ecc256` Use ecc256 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--ecc384` Use ecc384 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--ecc521` Use ecc521 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--rsa2048` Use rsa2048 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--rsa3072` Use rsa3072 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--rsa4096` Use rsa4096 for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--lms` Use LMS/HSS for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--xmss` Use XMSS/XMSS^MT for signing the firmware. Assume that the given KEY.DER file is in this format.
- `--no-sign` Disable secure boot signature verification. No signature verification is performed in the bootloader, and the KEY.DER argument should not be supplied.

K.2.2.2 Hash digest options If none of the following is used, '–sha256' is assumed by default.

- `--sha256` Use sha256 for digest calculation on binary images and public keys.
- `--sha348` Use sha384 for digest calculation on binary images and public keys.
- `--sha3` Use sha3-384 for digest calculation on binary images and public keys.

K.2.2.3 Target partition id (Multiple partition images, "self-update" feature) If none of the following is used, "–id=1" is assumed by default. On systems with a single image to verify (e.g. microcontroller with a single active partition), ID=1 is the default identifier for the firmware image to stage. ID=0 is reserved for wolfBoot 'self-update', and refers to the partition where the bootloader itself is stored.

- `--id N` Set image partition id to "N".
- `--wolfboot-update` Indicate that the image contains a signed self-update package for the bootloader. Equivalent to `--id 0`.

K.2.2.4 Encryption using a symmetric key Although signed to be authenticated, by default the image is not encrypted and it's distributed as plain text. End-to-end encryption from the firmware packaging to the update process can be used if the firmware is stored on external non-volatile memories. Encrypted updates can be produced using a pre-shared, secret symmetric key, by passing the following option:

- `--encrypt SHAREDKEY.BIN` use the file SHAREKEY.BIN to encrypt the image.

The format of the file depends on the algorithm selected for the encryption. If no format is specified, and the `--encrypt SHAREDKEY.BIN` option is present, `--chacha` is assumed by default.

See options below.

- `--chacha` Use ChaCha20 algorithm for encrypting the image. The file SHAREDKEY.BIN is expected to be exactly 44 bytes in size, of which 32 will be used for the key, 12 for the initialization of the IV.
- `--aes128` Use AES-128 algorithm in counter mode for encrypting the image. The file SHAREDKEY.BIN is expected to be exactly 32 bytes in size, of which 16 will be used for the key, 16 for the initialization of the IV.
- `--aes256` Use AES-256 algorithm in counter mode for encrypting the image. The file SHAREDKEY.BIN is expected to be exactly 48 bytes in size, of which 32 will be used for the key, 16 for the initialization of the IV.

K.2.2.5 Delta updates (incremental updates from a known version) An incremental update is created using the sign tool when the following option is provided:

- `--delta BASE_SIGNED_IMG.BIN` This option creates a binary diff file between `BASE_SIGNED_IMG.BIN` and the new image signed starting from `IMAGE.BIN`. The result is stored in a file ending in `_signed_diff.bin`.

The compression scheme used is Bentley–McIlroy.

K.2.2.6 Policy signing (for sealing/unsealing with a TPM) Provides a PCR mask and digest to be signed and included in the header. The signing key is used to sign the digest.

- `--policy policy.bin`: This argument is multi-purpose. By default the file should contain a 4-byte PCR mask and SHA2-256 PCR digest to be signed. If using `--manual-sign` then the file should contain the 4-byte PCR mask and signature. The PCR mask and signature will be included in the `HDR_POLICY_SIGNATURE` header tag. A copy of the final signed policy (including 4 byte PCR mask) will be output to `[inputname].sig`. Note: This may require increasing the `IMAGE_HEADER_SIZE` as two signatures will be stored in the header.

K.2.2.7 Adding custom fields to the manifest header Provides a value to be set with a custom tag

- `--custom-tlv tag len val`: Adds a TLV entry to the manifest header, corresponding to the type identified by `tag`, with length `len` bytes, and assigns the value `val`. Values can be decimal or hex numbers (prefixed by '0x'). The tag is a 16-bit number. Valid tags are in the range between 0x0030 and 0xFEFE.
- `--custom-tlv-buffer tag value`: Adds a TLV entry with arbitrary length to the manifest header, corresponding to the type identified by `tag`, and assigns the value `value`. The tag is a 16-bit number. Valid tags are in the range between 0x0030 and 0xFEFE. The length is implicit, and is the length of the value. Value argument is in the form of a hex string, e.g. `--custom-tlv-buffer 0x0030 AABBCCDDEE` will add a TLV entry with tag 0x0030, length 5 and value 0xAABBCCDDEE.
- `--custom-tlv-string tag ascii-string`: Adds a TLV entry with arbitrary length to the manifest header, corresponding to the type identified by `tag`, and assigns the value of `ascii-string`. The tag is a 16-bit number. Valid tags are in the range between 0x0030 and 0xFEFE. The length is implicit, and is the length of the `ascii-string`. `ascii-string` argument is in the form of a string, e.g. `--custom-tlv-string 0x0030 "Version-1"` will add a TLV entry with tag 0x0030, length 9 and value Version-1.

K.2.2.8 Three-steps signing using external provisioning tools If the private key is not accessible, while it's possible to sign payloads using a third-party tool, the sign mechanism can be split in three phases:

- Phase 1: Only create the sha digest for the image, and prepare an intermediate file that can be signed by third party tool.

This is done using the following option:

- `--sha-only` When this option is selected, the sign tool will create an intermediate image including part of the manifest that must be signed, ending in `_digest.bin`. In this case, KEY.DER contains the public part of the key that will be used to sign the firmware in Phase 2.
- Phase 2: The intermediate image `*_digest.bin` is signed by an external tool, an HSM or a third party signing service. The signature is then exported in its raw format and copied to a file, e.g. IMAGE_SIGNATURE.SIG
- Phase 3: use the following option to build the final authenticated firmware image, including its manifest header in front:
  - `--manual-sign` When this option is provided, the KEY.DER argument contains the public part of the key that was used to sign the firmware in Phase 2. This option requires one extra argument at the end, after VERSION, which should be the filename of the signature that was the output of the previous phase, so `IMAGE_SIGNATURE.SIG`

For a real-life example, see the section below.

K.3 Examples

K.3.1 Signing Firmware

1. Load the private key to use for signing into `./wolfboot_signing_private_key.der`
2. Run the signing tool with asymmetric algorithm, hash algorithm, file to sign, key and version.

```
./tools/keytools/sign --rsa2048 --sha256 test-app/image.bin
↪ wolfboot_signing_private_key.der 1
```

Note: The last argument is the "version" number.

K.3.2 Signing Firmware with External Private Key (HSM)

Steps for manually signing firmware using an external key source.

```
# Create file with Public Key
openssl rsa -inform DER -outform DER -in my_key.der -out rsa2048_pub.der
 ↪  -pubout

# Add the public key to the wolfBoot keystore using `keygen -i`
./tools/keytools/keygen --rsa2048 -i rsa2048_pub.der

# Generate Hash to Sign
./tools/keytools/sign --rsa2048 --sha-only --sha256 test-app/image.bin
 ↪  rsa2048_pub.der 1

# Sign hash Example (here is where you would use an HSM)
openssl pkeyutl -sign -keyform der -inkey my_key.der -in
 ↪  test-app/image_v1_digest.bin > test-app/image_v1.sig

# Generate final signed binary
./tools/keytools/sign --rsa2048 --sha256 --manual-sign test-app/image.bin
 ↪  rsa2048_pub.der 1 test-app/image_v1.sig

# Combine into factory image (0xc0000 is the WOLFBOOT_PARTITION_BOOT_ADDRESS)
tools/bin-assemble/bin-assemble factory.bin 0x0 wolfboot.bin \
                            0xc0000 test-app/image_v1_signed.bin
```

K.3.3 Signing Firmware with Azure Key Vault

See Appendix B.

L wolfCrypt in TrustZone-M secure domain

ARMv8-M microcontrollers support hardware-assisted domain separation for running software. This TEE mechanism provides two separate domains (secure & non-secure), and an additional zone that can be used as interface to call into secure functions from the non-secure domain (non-secure callable).

wolfBoot may optionally export the crypto functions as a non-callable APIs that are accessible from any software staged in non-secure domain.

L.1 Compiling wolfBoot with wolfCrypt in TrustZone-M secure domain

When wolfBoot is compiled with the options `TZEN=1` and `WOLFCRYPT_TZ=1`, a more complete set of components of the wolfCrypt crypto library are built-in the bootloader, and they can be accessed by applications or OSs running in non-secure domain through non-secure callable APIs.

This feature is used to isolate the core crypto operations from the applications.

L.2 PKCS11 API in non-secure world

The `WOLFCRYPT_TZ_PKCS11` option provides a standard PKCS11 interface, including a storage for PKCS11 objects in a dedicated flash area in secure mode.

This means that applications, TLS libraries and operating systems running in non-secure domain can access wolfCrypt through a standard PKCS11 interface and use the crypto library with pre-provisioned keys that are never exposed to the non-secure domain.

L.3 Example using STM32L552

- Copy the example configuration for STM32-L5 with support for wolfCrypt in TrustZone-M and PKCS11 interface: `cp config/examples/stm32l5-wolfcrypt-tz.config .config`
- Run `make`. `wolfboot.elf` and the test applications are built as separate objects. The application is signed and stored as `test-app/image_v1_signed.bin`.
- Ensure that the option bytes on your target device are set as follows:

```
OPTION BYTES BANK: 0

   Read Out Protection:

     RDP          : 0xAA (Level 0, no protection)

   BOR Level:

     BOR_LEV      : 0x0 (BOR Level 0, reset level threshold is around 1.7 V)

   User Configuration:

     nRST_STOP    : 0x1 (No reset generated when entering Stop mode)
     nRST_STDBY   : 0x1 (No reset generated when entering Standby mode)
     nRST_SHDW    : 0x1 (No reset generated when entering the Shutdown mode)
     IWDG_SW      : 0x1 (Software independant watchdog)
     IWDG_STOP    : 0x1 (IWDG counter active in stop mode)
     IWDG_STDBY   : 0x1 (IWDG counter active in standby mode)
     WWDG_SW      : 0x1 (Software window watchdog)
     SWAP_BANK    : 0x0 (Bank 1 and bank 2 address are not swapped)
```

```
    DB256        : 0x1 (256Kb dual-bank Flash with contiguous addresses)
    DBANK        : 0x0 (Single bank mode with 128 bits data read width)
    SRAM2_PE     : 0x1 (SRAM2 parity check disable)
    SRAM2_RST    : 0x1 (SRAM2 is not erased when a system reset occurs)
    nSWBOOT0     : 0x1 (BOOT0 taken from PH3/BOOT0 pin)
    nBOOT0       : 0x1 (nBOOT0 = 1)
    PA15_PUPEN   : 0x1 (USB power delivery dead-battery disabled/ TDI pull-up
        activated)
    TZEN         : 0x1 (Global TrustZone security enabled)
    HDP1EN       : 0x0 (No HDP area 1)
    HDP1_PEND    : 0x0  (0x8000000)
    HDP2EN       : 0x0 (No HDP area 2)
    HDP2_PEND    : 0x0  (0x8000000)
    NSBOOTADD0   : 0x100000  (0x8000000)
    NSBOOTADD1   : 0x17F200  (0xBF90000)
    SECBOOTADD0  : 0x180000  (0xC000000)
    BOOT_LOCK    : 0x0 (Boot based on the pad/option bit configuration)

  Secure Area 1:

    SECWM1_PSTRT : 0x0  (0x8000000)
    SECWM1_PEND  : 0x39  (0x8039000)

  Write Protection 1:

    WRP1A_PSTRT  : 0x7F  (0x807F000)
    WRP1A_PEND   : 0x0  (0x8000000)
    WRP1B_PSTRT  : 0x7F  (0x807F000)
    WRP1B_PEND   : 0x0  (0x8000000)
OPTION BYTES BANK: 1

  Secure Area 2:

    SECWM2_PSTRT : 0x7F  (0x807F000)
    SECWM2_PEND  : 0x0  (0x8000000)

  Write Protection 2:

    WRP2A_PSTRT  : 0x7F  (0x80BF000)
    WRP2A_PEND   : 0x0  (0x8040000)
    WRP2B_PSTRT  : 0x7F  (0x80BF000)
    WRP2B_PEND   : 0x0  (0x8040000)
```

- Upload `wolfboot.bin` and the test application to the two different domains in flash:

```
STM32_Programmer_CLI -c port=swd -d wolfboot.bin 0x0C000000
STM32_Programmer_CLI -c port=swd -d test-app/image_v1_signed.bin 0x08040000
```

- After rebooting, the LED on the board should turn on sequentially:
  - Red LED: Secure boot was successful. Application has started.
  - Blue LED: PKCS11 Token has been initialized and stored
  - Green LED: ECDSA Sign/Verify test successful

L.4 Example using STM32H563

- Copy one of the example configurations for STM32H5 with support for TrustZone and PKCS11 to `.config`: `cp config/examples/stm32h5-tz.config .config` `cp config/examples/stm32h5-tz-dualbank-otp.config .config` (with Dual Bank) `cp config/examples/stm32h5-tz-dualbank-otp-lms.config .config` (with Dual Bank and PQ LMS)
- Run `make`. `wolfboot.elf` and the test applications are built as separate objects. The application is signed and stored as `test-app/image_v1_signed.bin`.
- Ensure that the option bytes on your target device are set as follows:

```
OPTION BYTES BANK: 0

   Product state:

     PRODUCT_STATE: 0xED (Open)

   BOR Level:

     BOR_LEV      : 0x0 (BOR Level 1, the threshold level is low (around 2.1 V
        ))
     BORH_EN      : 0x0  (0x0)

   User Configuration:

     IO_VDD_HSLV  : 0x0  (0x0)
     IO_VDDIO2_HSLV: 0x0  (0x0)
     IWDG_STOP    : 0x1  (0x1)
     IWDG_STDBY   : 0x1  (0x1)
     BOOT_UBE     : 0xB4 (OEM-iRoT (user flash) selected)
     SWAP_BANK    : 0x0  (0x0)
     IWDG_SW      : 0x1  (0x1)
     NRST_STOP    : 0x1  (0x1)
     NRST_STDBY   : 0x1  (0x1)
OPTION BYTES BANK: 1

   User Configuration 2:

     TZEN         : 0xB4 (Trust zone enabled)
     SRAM2_ECC    : 0x1 (SRAM2 ECC check disabled)
     SRAM3_ECC    : 0x1 (SRAM3 ECC check disabled)
     BKPRAM_ECC   : 0x1 (BKPRAM ECC check disabled)
     SRAM2_RST    : 0x1 (SRAM2 not erased when a system reset occurs)
     SRAM1_3_RST  : 0x1 (SRAM1 and SRAM3 not erased when a system reset occurs
        )
OPTION BYTES BANK: 2

   Boot Configuration:

     NSBOOTADD    : 0x80400  (0x8040000)
     NSBOOT_LOCK  : 0xC3 (The SWAP_BANK and NSBOOTADD can still be modified
        following their individual rules.)
     SECBOOT_LOCK : 0xC3 (The BOOT_UBE, SWAP_BANK and SECBOOTADD can still be
        modified following their individual rules.)
     SECBOOTADD   : 0xC0000  (0xC000000)
```

```
OPTION BYTES BANK: 3

   Bank1 - Flash watermark area definition:

     SECWM1_STRT  : 0x0  (0x8000000)
     SECWM1_END   : 0x1F  (0x803E000)

   Write sector group protection 1:

     WRPSGn1      : 0xFFFFFFFF  (0x0)
OPTION BYTES BANK: 4

   Bank2 - Flash watermark area definition:

     SECWM2_STRT  : 0x7F  (0x81FE000)
     SECWM2_END   : 0x0  (0x8100000)

   Write sector group protection 2:

     WRPSGn2      : 0xFFFFFFFF  (0x8000000)
OPTION BYTES BANK: 5

   OTP write protection:

     LOCKBL       : 0x0  (0x0)
OPTION BYTES BANK: 6

   Flash data bank 1 sectors:

     EDATA1_EN    : 0x0 (No Flash high-cycle data area)
     EDATA1_STRT  : 0x0  (0x0)
OPTION BYTES BANK: 7

   Flash data bank 2 sectors :

     EDATA2_EN    : 0x0 (No Flash high-cycle data area)
     EDATA2_STRT  : 0x0  (0x0)
OPTION BYTES BANK: 8

   Flash HDP bank 1:

     HDP1_STRT    : 0x1  (0x2000)
     HDP1_END     : 0x0  (0x0)
OPTION BYTES BANK: 9

   Flash HDP bank 2:

     HDP2_STRT    : 0x1  (0x2000)
     HDP2_END     : 0x0  (0x0)
```

- Upload `wolfboot.bin` and the test application to the two different domains in flash:

```
STM32_Programmer_CLI -c port=swd -d wolfboot.bin 0x0C000000
STM32_Programmer_CLI -c port=swd -d test-app/image_v1_signed.bin 0x08040000
```

- After rebooting, the LED on the board should turn on sequentially:
  - Red LED: Secure boot was successful. Application has started.
  - Blue LED: PKCS11 Token has been initialized and stored
  - Green LED: ECDSA Sign/Verify test successful

M wolfBoot TPM support

In wolfBoot we support TPM based root of trust, sealing/unsealing, cryptographic offloading and measured boot using a TPM.

M.1 Build Options

Config Option	Preprocessor Macro	Description
WOLFTPM=1	WOLFBOOT_TPM	Enables wolfTPM support
WOLFBOOT_TPM_VERIFY=1	WOLFBOOT_TPM_VERIFY	Enables cryptographic offloading for RSA2048 and ECC256/384 to the TPM.
WOLFBOOT_TPM_KEYSTORE=1	WOLFBOOT_TPM_KEYSTORE	Enables TPM based root of trust. NV Index must store a hash of the trusted public key.
WOLFBOOT_TPM_KEYSTORE_NV_BASE=0x	WOLFBOOT_TPM_KEYSTORE_NV_BASE	NV index platform range 0x1400000 - 0x17FFFFF.
WOLFBOOT_TPM_KEYSTORE_AUTH=secret	WOLFBOOT_TPM_KEYSTORE_AUTH	Password for NV access
MEASURED_BOOT=1	WOLFBOOT_MEASURED_BOOT	Enable measured boot. Extend PCR with wolfBoot hash.
MEASURED_PCR_A=16	WOLFBOOT_MEASURED_PCR_A=16	The PCR index to use. See Appendix G.
WOLFBOOT_TPM_SEAL=1	WOLFBOOT_TPM_SEAL	Enables support for sealing/unsealing based on PCR policy signed externally.
WOLFBOOT_TPM_SEAL_NV_BASE=0x1400300	WOLFBOOT_TPM_SEAL_NV_BASE	Override the default sealed blob storage location in the platform hierarchy.
WOLFBOOT_TPM_SEAL_AUTH=secret	WOLFBOOT_TPM_SEAL_AUTH	Password for sealing/unsealing secrets, if omitted the PCR policy will be used

M.2 Root of Trust (ROT)

See wolfTPM Secure Root of Trust (ROT) example here.

The design uses a platform NV handle that has been locked. The NV stores a hash of the public key. It is recommended to supply a derived “authentication” value to prevent TPM tampering. This authentication value is encrypted on the bus.

M.3 Cryptographic offloading

The RSA2048 and ECC256/384 bit verification can be offloaded to a TPM for code size reduction or performance improvement. Enabled using `WOLFBOOT_TPM_VERIFY`. NOTE: The TPM’s RSA verify requires ASN.1 encoding, so use SIGN=RSA2048ENC

M.4 Measured Boot

The wolfBoot image is hashed and extended to the indicated PCR. This can be used later in the application to prove the boot process was not tampered with. Enabled with `WOLFBOOT_MEASURED_BOOT` and exposes API `wolfBoot_tpm2_extend`.

M.5 Sealing and Unsealing a secret

See the wolfTPM Sealing/Unsealing example here

Known PCR values must be signed to seal/unseal a secret. The signature for the authorization policy resides in the signed header using the `--policy` argument. If a signed policy is not in the header then

a value cannot be sealed. Instead the PCR(s) values and a PCR policy digest will be printed to sign. You can use `./tools/keytools/sign` or `./tools/tpm/policy_sign` to sign the policy externally.

This exposes two new wolfBoot API's for sealing and unsealing data with blob stored to NV index:

```
int wolfBoot_seal_auth(const uint8_t* pubkey_hint, const uint8_t* policy,
 ↪  uint16_t policySz,
    int index, const uint8_t* secret, int secret_sz, const byte* auth, int
     ↪  authSz);
int wolfBoot_unseal_auth(const uint8_t* pubkey_hint, const uint8_t* policy,
 ↪  uint16_t policySz,
    int index, uint8_t* secret, int* secret_sz, const byte* auth, int authSz);
```

By default this index will be based on an NV Index at `(0x01400300 + index)`. The default NV base can be overridden with `WOLFBOOT_TPM_SEAL_NV_BASE`.

NOTE: The TPM's RSA verify requires ASN.1 encoding, so use SIGN=RSA2048ENC

M.5.1 Testing seal/unseal with simulator

```
% cp config/examples/sim-tpm-seal.config .config
% make keytools
% make tpmtools
% echo aaa > aaa.bin
% ./tools/tpm/pcr_extend 0 aaa.bin
% ./tools/tpm/policy_create -pcr=0
# if ROT enabled
% ./tools/tpm/rot -write [-auth=TestAuth]
% make clean
$ make POLICY_FILE=policy.bin [WOLFBOOT_TPM_KEYSTORE_AUTH=TestAuth]
 ↪  [WOLFBOOT_TPM_SEAL_AUTH=SealAuth]

% ./wolfboot.elf get_version
Simulator assigned ./internal_flash.dd to base 0x103378000
Mfg IBM  (0), Vendor SW   TPM, Fw 8217.4131 (0x163636), FIPS 140-2 1, CC-EAL4 0
Unlocking disk...
Boot partition: 0x1033f8000
Image size 54400
Error 395 reading blob from NV index 1400300 (error TPM_RC_HANDLE)
Error 395 unsealing secret! (TPM_RC_HANDLE)
Sealed secret does not exist!
Creating new secret (32 bytes)
430dee45553c4a8b75fbc6bcd0890765c48cab760b24b1aa6b633dc0538e0159
Wrote 210 bytes to NV index 0x1400300
Read 210 bytes from NV index 0x1400300
Secret Check 32 bytes
430dee45553c4a8b75fbc6bcd0890765c48cab760b24b1aa6b633dc0538e0159
Secret 32 bytes
430dee45553c4a8b75fbc6bcd0890765c48cab760b24b1aa6b633dc0538e0159
Boot partition: 0x1033f8000
Image size 54400
TPM Root of Trust valid (id 0)
Simulator assigned ./internal_flash.dd to base 0x103543000
1

% ./wolfboot.elf get_version
```

```
Simulator assigned ./internal_flash.dd to base 0x10c01c000
Mfg IBM  (0), Vendor SW   TPM, Fw 8217.4131 (0x163636), FIPS 140-2 1, CC-EAL4 0
Unlocking disk...
Boot partition: 0x10c09c000
Image size 54400
Read 210 bytes from NV index 0x1400300
Secret 32 bytes
430dee45553c4a8b75fbc6bcd0890765c48cab760b24b1aa6b633dc0538e0159
Boot partition: 0x10c09c000
Image size 54400
TPM Root of Trust valid (id 0)
Simulator assigned ./internal_flash.dd to base 0x10c1e7000
1
```

M.5.2 Testing seal/unseal on actual hardware

1) Get the actual PCR digest for policy.
2) Sign policy and include in firmware image header.

M.5.2.1 Getting PCR values If no signed policy exists, then the seal function will generate and display the active PCR's, PCR digest and policy digest (to sign)

```
% make tpmtools
% ./tools/tpm/rot -write
% ./tools/tpm/pcr_reset 16
% ./wolfboot.elf get_version
Simulator assigned ./internal_flash.dd to base 0x101a64000
Mfg IBM  (0), Vendor SW   TPM, Fw 8217.4131 (0x163636), FIPS 140-2 1, CC-EAL4 0
Boot partition: 0x101ae4000
Image size 57192
Policy header not found!
Generating policy based on active PCR's!
Getting active PCR's (0-16)
PCR 16 (counter 20)
8f7ac1d5a5eac58a2305ca459f27c35705a9212c0fb2a9088b1df761f3d5f842
Found 1 active PCR's (mask 0x00010000)
PCR Digest (32 bytes):
f84085631f85333ad0338b06c82f16888b7923abaccffb881d5416e389be256c
PCR Mask (0x00010000) and PCR Policy Digest (36 bytes):
0000010034ba061436aba2e9a167a1ee46af4a9578a8c6b9f71fdece21607a0cb40468ec
Use this policy with the sign tool (--policy arg) or POLICY_FILE config
Image policy signature missing!
Boot partition: 0x101ae4000
Image size 57192
TPM Root of Trust valid (id 0)
Simulator assigned ./internal_flash.dd to base 0x101c2f000
1
```

The `0000010034ba061436aba2e9a167a1ee46af4a9578a8c6b9f71fdece21607a0cb40468ec` above can be directly used by the keytool. The

```
echo "0000010034ba061436aba2e9a167a1ee46af4a9578a8c6b9f71fdece21607a0cb40468ec"
| xxd -r -p > policy.bin
```

OR use the `tools/tpm/policy_create` tool to generate a digest to be signed. The used PCR(s) must be set using "-pcr=#". The PCR digest can be supplied using "-pcrdigest=" or if not supplied will be read

from the TPM directly.

```
% ./tools/tpm/policy_create -pcr=16 -
 ↪  pcrdigest=f84085631f85333ad0338b06c82f16888b7923abaccffb881d5416e389be256c
 ↪  -out=policy.bin
# OR
% ./tools/tpm/policy_create -pcrmask=0x00010000 -
 ↪  pcrdigest=f84085631f85333ad0338b06c82f16888b7923abaccffb881d5416e389be256c
 ↪  -out=policy.bin
Policy Create Tool
PCR Index(s) (SHA256): 16  (mask 0x00010000)
PCR Digest (32 bytes):
    f84085631f85333ad0338b06c82f16888b7923abaccffb881d5416e389be256c
PCR Mask (0x00010000) and PCR Policy Digest (36 bytes):
    0000010034ba061436aba2e9a167a1ee46af4a9578a8c6b9f71fdece21607a0cb40468ec
Wrote 36 bytes to policy.bin
```

M.5.2.2 Signing Policy

Building firmware with the policy digest to sign:

```
% make POLICY_FILE=policy.bin
```

OR manually sign the policy using the `tools/tpm/policy_sign` or `tools/keytools/sign` tools. These tools do not need access to a TPM, they are signing a policy digest. The result is a 32-bit PCR mask + signature.

Sign with policy_sign tool:

```
% ./tools/tpm/policy_sign -pcr=0 -
 ↪  pcrdigest=eca4e8eda468b8667244ae972b8240d3244ea72341b2bf2383e79c66643bbecc
Sign PCR Policy Tool
Signing Algorithm: ECC256
PCR Index(s) (SHA256): 0
Policy Signing Key: wolfboot_signing_private_key.der
PCR Digest (32 bytes):
    eca4e8eda468b8667244ae972b8240d3244ea72341b2bf2383e79c66643bbecc
PCR Policy Digest (32 bytes):
    2d401eb05f45ba2b15c35f628b5896cc7de9745bb6e722363e2dbee804e0500f
PCR Policy Digest (w/PolicyRef) (32 bytes):
    749b3139ece21449a7828f11ee05303b0473ff1a26cf41d6f9ff28b24c717f02
PCR Mask (0x1) and Policy Signature (68 bytes):
    01000000
    5b5f875b3f7ce78b5935abe4fc5a4d8a6e87c4b4ac0836fbab909e232b6d7ca2
    3ecfc6be723b695b951ba2886d3c7b83ab2f8cc0e96d766bc84276eaf3f213ee
Wrote PCR Mask + Signature (68 bytes) to policy.bin.sig
```

Sign using the signing key tool:

```
% ./tools/keytools/sign --ecc256 --policy policy.bin test-app/image.elf
 ↪  wolfboot_signing_private_key.der 1
wolfBoot KeyTools (Compiled C version)
wolfBoot version 1100000
Update type:          Firmware
Input image:          test-app/image.elf
Selected cipher:      ECC256
Selected hash  :      SHA256
Public key:           wolfboot_signing_private_key.der
Output  image:        test-app/image_v1_signed.bin
```

```
Target partition id : 1
image header size calculated at runtime (256 bytes)
Calculating SHA256 digest...
Signing the digest...
Opening policy file policy.bin
Signing the policy digest...
Saving policy signature to policy.bin.sig
Output image(s) successfully created.
```

N wolfBoot Configuration Options

This section shows parameters by running `make config`.

- `ARCH`: Architecture of the target to be used.
  - Default: ARM
  - Possible: x86_64/AARCH64/ARM/RNESAS_RX/RISCV/PPC/ARM_BE
- `HASH`: Selection of hash algorithm to be used.
  - Default: SHA256
  - Possible: SHA3/SHA256/SHA384
- `MCUXSDK`: Enable when using NXP's MCUXpresso SDK.
  - Default: 1
- `MCUXPRESSO`: Setting for MCUXpresso IDE environment.
  - Default: /mnt/c/Users/(User)/(Project)/wolfboot-2.4.0/mcux-sdk
- `MCUXPRESSO_CPU`: CPU-specific settings for MCUXpresso.
  - Default: MK64FN1M0VLL12
- `MCUXPRESSO_DRIVERS`: Enable driver support for MCUXpresso.
  - Default: /mnt/c/Users/(User)/(Project)/wolfboot-2.4.0/mcux-sdk/devices/MK64F12
- `MCUXPRESSO_CMSIS`: Enable CMSIS (Cortex Microcontroller Software Interface Standard) library.
  - Default: /mnt/c/Users/(User)/(Project)/wolfboot-2.4.0/CMSIS_5/CMSIS
- `FREEDOM_E_SDK`: Enable when using SiFive Freedom-E SDK (for RISC-V).
  - Default: /home/(User)/src/freedom-e-sdk
- `STM32CUBE`: Enable STM32Cube HAL (for STM32).
  - Default: /home/(User)/STM32Cube/Repository/STM32Cube_FW_WB_V1.3.0
- `CYPRESS_PDL`: Enable Cypress Peripheral Driver Library (PDL).
  - Default: /home/(User)/src/psoc6pdl
- `CYPRESS_CORE_LIB`: Enable Cypress core library.
  - Default: /home/(User)/src/cypress-core-lib
- `CYPRESS_TARGET_LIB`: Enable Cypress target-specific library.
  - Default: /home/(User)/src/TARGET_CY8CKIT-062S2-43012
- `CORTEX_M7`: Enable when targeting ARM Cortex-M7.
  - Default: 0
- `CORTEX_M33`: Enable when targeting ARM Cortex-M33.
  - Default: 0
- `NO_ASM`: Disable assembly optimizations and implement in C language only.
  - Default: 0
- `NO_XIP`: Disable XIP (Execute in Place) (do not execute code directly from flash memory).
  - Default: 0

- WOLFBOOT_VERSION: Option to specify the version of wolfBoot.
  - Default is set in include/wolfboot/version.h
- V: Enable Verbose build.
  - Default: 0
- NO_MPU: Disable Memory Protection Unit (MPU).
  - Default: 0
- SPMATH: Enable SP Math library (single-precision math library).
  - Default: 1
- SPMATHALL: Enable all SP Math functions.
  - Default: 0
- IMAGE_HEADER_SIZE: Specify the firmware image header size.
  - Default: 256
- PKA: Enable public key cryptography processing (Public Key Accelerator).
  - Default: 1
- TZEN: Enable TrustZone security features.
  - Default: 0
- PSOC6_CRYPTO: Use Cypress PSoC 6 series hardware cryptographic engine.
  - Default: 1
- WOLFBOOT_TPM_VERIFY: Enable firmware verification using TPM (Trusted Platform Module).
  - Default: 0
- WOLFBOOT_TPM_SEAL: Enable function to seal data using TPM.
  - Default: 0
- WOLFBOOT_TPM_KEYSTORE: Enable key storage using TPM.
  - Default: 0
- WOLFCRYPT_TZ: Enable the use of wolfCrypt in TrustZone.
  - Default: 0
- WOLFCRYPT_TZ_PKCS11: Enable PKCS#11 interface in TrustZone.
  - Default: 0
- WOLFBOOT_LOAD_ADDRESS: Specify the load address for wolfBoot.
  - Default: 0x200000
- WOLFBOOT_LOAD_DTS_ADDRESS: Specify the load address for Device Tree Storage (DTS).
  - Default: 0x400000
- WOLFBOOT_DTS_BOOT_ADDRESS: Specify the device tree address during boot.
  - Default: 0x30000
- WOLFBOOT_DTS_UPDATE_ADDRESS: Specify the device tree address for updates.
  - Default: 0x50000

- DELTA_BLOCK_SIZE: Specify the block size for delta updates.
  - Default: 256
- WOLFBOOT_HUGE_STACK: Option to increase stack size.
  - Default: 0
- FORCE_32BIT: Option to force build as a 32-bit system.
  - Default: 0
- ENCRYPT_WITH_CHACHA: Enable firmware encryption using the ChaCha algorithm.
  - Default: 0
- ARMORED: Enable additional mitigations against fault-injection attacks, e.g. voltage and clock glitches, or EMFI.
  - Default: 0
- LMS_LEVELS: Specify the levels for LMS (Leighton-Micali Signature) hash-based signatures.
  - Default: 0
- LMS_HEIGHT: Specify the hash tree height for LMS signatures.
  - Default: 0
- LMS_WINTERNITZ: Set the Winternitz coefficient (LMS signature parameter).
  - Default: 0
- WOLFBOOT_UNIVERSAL_KEYSTORE: Enable storing public keys of different types in the same keystore.
  - Default: 0
- XMSS_PARAMS: Specify parameters for XMSS (eXtended Merkle Signature Scheme).
  - Default: XMSS-SHA2_10_256
  - Possible: XMSS-SHA2_10_256
- ELF: Enable support for ELF format.
  - Default: 0
- BIG_ENDIAN: Support big-endian architecture.
  - Default: 0
- NXP_CUSTOM_DCD: Enable custom DCD (Device Configuration Data) settings for NXP platforms.
  - Default: 0
- NXP_CUSTOM_DCD_OBJS: Enable custom DCD objects for NXP.
- FLASH_OTP_KEYSTORE: Enable flash key storage using OTP (One-Time Programmable) memory.
  - Default: 0
- KEYVAULT_OBJ_SIZE: Specify the size of objects stored in KeyVault.
- KEYVAULT_MAX_ITEMS: Specify the maximum number of items that can be stored in KeyVault.
- NO_ARM_ASM: Disable ARM assembly code and implement in C language only.
  - Default: 0

- SIGN_SECONDARY: Enable a second signature for the images. Used to implement hybrid mode (e.g. ECC + ML_DSA). Set to the secondary algorithm selected for hybrid (classic + PQC) authentication.
- WOLFHSM_CLIENT: Enable wolfHSM client (HSM).
  - Default: 0
- WOLFHSM_CLIENT_LOCAL_KEYS: Option for wolfHSM client to use local keys.
  - Default: 0