AN5544

Application note

Integration guide of SBSFU on STM32CubeWL (including KMS)

Introduction

The SBSFU (Secure Boot and Secure Firmware Update) solution allows the update of the STM32 microcontroller built-in program with new firmware versions, adding new features and correcting potential issues. The update process is performed in a secure way to prevent unauthorized updates and access to confidential on-device data.

The Secure Boot (made of Root of Trust services) is an immutable code, always executed after a system reset.

In a single-core configuration, the Secure Boot checks STM32 static protections, activates STM32 runtime protections, and then verifies the authenticity and integrity of the user application code before every execution, to make sure that invalid or malicious code cannot be run.

In a dual-core configuration, the Secure Boot is made of two parts (one per core):

- Cortex®-M4 side: The Secure Boot checks static protections, checks the Cortex®-M0+ boot configuration, activates Cortex®-M4 runtime protections and boots the Cortex®-M0+.
- Cortex®-M0+ side: The Secure Boot checks static protections,activates Cortex®-M0+ runtime protections, verifies the authenticity and integrity of the user application code before every execution to make sure that invalid or malicious code cannot be run, and then signals to both cores that the user applications are valid.

The Secure Firmware Update application receives the firmware image via a UART interface with the Ymodem protocol. The Secure Firmware Update checks the image authenticity, and the integrity of the code before installing it. The firmware update is done on the complete firmware image, or only on a portion of the firmware image.

Examples can be configured to use asymmetric or symmetric cryptographic schemes with or without firmware encryption:

- to maximize firmware image size, for single-slot configuration
- to ensure safe image installation and enable over-the-air firmware update capability commonly used in IoT devices, for dual-slot configuration

For a complex system with multiple firmware such as protocol stack, middleware, and user application, the firmware image configuration can be extended up to three firmware images. In the applications detailed in this document, one firmware image is used for the single-core configuration, while two firmware images are available for the dual-core configuration.

In the dual-core configuration, the secure key management services (KMS) provide cryptographic services to the user application through the PKCS #11 APIs (KEY ID-based APIs), that are executed inside a protected and isolated environment. User application keys are stored in the protected and isolated environment for their secured update: authenticity check, data decryption, and data integrity check.

In the single-core configuration, the same services are offered but there are not executed inside a protected and isolated environment.

This application note describes how to adapt the STM32CubeWL SBSFU and to integrate it with the user application.

Note:
- *In this document, the EWARM IDE (integrated development environment) is used as an example to provide guidelines for project configuration. Secure Boot and Secure Firmware Update applications are referred to as SBSFU. Boot and Firmware update (with only attack surface reduction) applications are referred to as BFU.*
- *The single-core single-slot BFU configuration is demonstrated in an example named BFU_1_Slot. The single-core dual-slot BFU configuration is demonstrated in an example named BFU_2_Slots. The dual-core single-slot SBSFU configuration is demonstrated in an example named SBSFU_1_Slot_DualCore The dual-core dual-slot SBSFU configuration is demonstrated in an example named SBSFU_2_Slots_DualCore.*

1 General information

This document applies to the STM32CubeWL SBSFU, running on STM32WL Series Arm®-based microcontrollers.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

The table below lists acronyms and terms that are relevant for a better understanding of this document.

Table 1. Acronyms and terms

Acronym or term	Definition
AES	Advanced encryption standard
BFU	Boot and Firmware Update
DAP	Debug access port
ECDSA	Elliptic curve digital signature algorithm
GCM	AES Galois/counter mode
GTZC	Global security controller
HAL	Hardware abstraction layer
HDP	Hide protection
IDE	Integrated development environment
KMS	Key management services
MPU	Memory protection unit
RDP	Readout protection
SB	Secure Boot
SE	Secure Engine
SFU	Secure Firmware Update
SBSFU	Secure Boot and Secure Firmware Update
TZIC	Security illegal access controller
TZSC	Security controller
UART	Universal asynchronous receiver/transmitter
WRP	Write protection
Firmware image	Binary image (executable) run by the device as a user application
Firmware header	Bundle of meta-data describing the firmware image to be installed (contains firmware information and cryptographic information)
mbed-crypto	mbed implementation of the cryptographic algorithms
sfb file	Binary file packing the firmware header and the firmware image

Reference documents

- User manual *Getting started with STM32CubeWL for STM32WL Series* (UM2643)
- User manual *Getting started with the SBSFU of STM32CubeWL* (UM2767)
- User manual *STM32CubeProgrammer software description* (UM2237)
- *STM32 Cortex-M4 MCUs and MPUs programming manual* (PM0214)
- *Cortex-M0+ programming manual for STM32L0, STM32G0, STM32WL and STM32WB Series* (PM0223)

2 Port the STM32CubeWL SBSFU onto another board

The STM32CubeWL SBSFU supplements the STM32Cube software technology, making portability across different STM32WL MCUs easy. The STM32CubeWL SBSFU comes with four examples that are useful starting points to port it onto another STM32WL board. The NUCLEO-WL55JC board is used as example in this document.

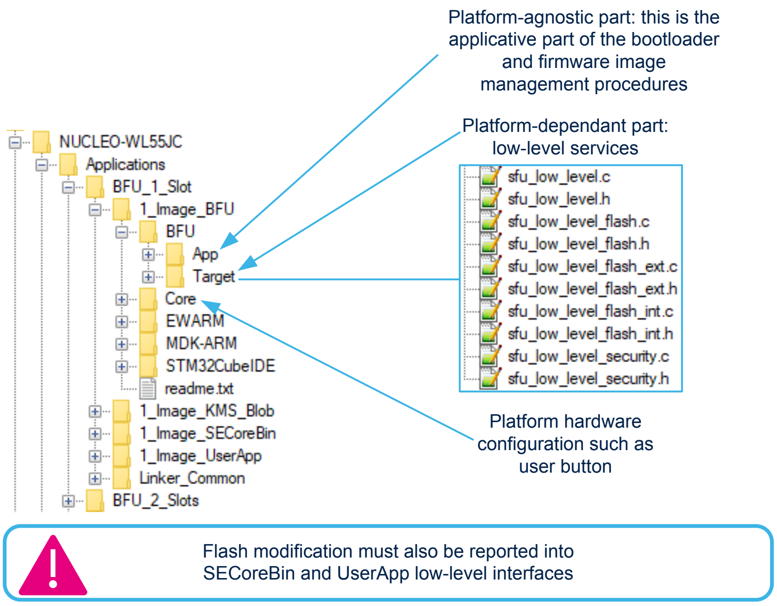
2.1 Hardware adaptation

A few changes are needed to adapt the STM32CubeWL SBSFU to another board:

- GPIO configuration for UART communication with the host PC (in the `sfu_low_level.h` file)
- Flash configuration (in the `sfu_low_level_flash.c`, `sfu_low_level_flash_int.c` and `sfu_low_level_flash_ext.c` files)
- Button configuration (in the `app_hw.h` file)
- Tamper GPIO pin configuration (in the `sfu_low_level_security.h` file)
- DAP (debug access port) configuration (in the `sfu_low_level_security.h` file)

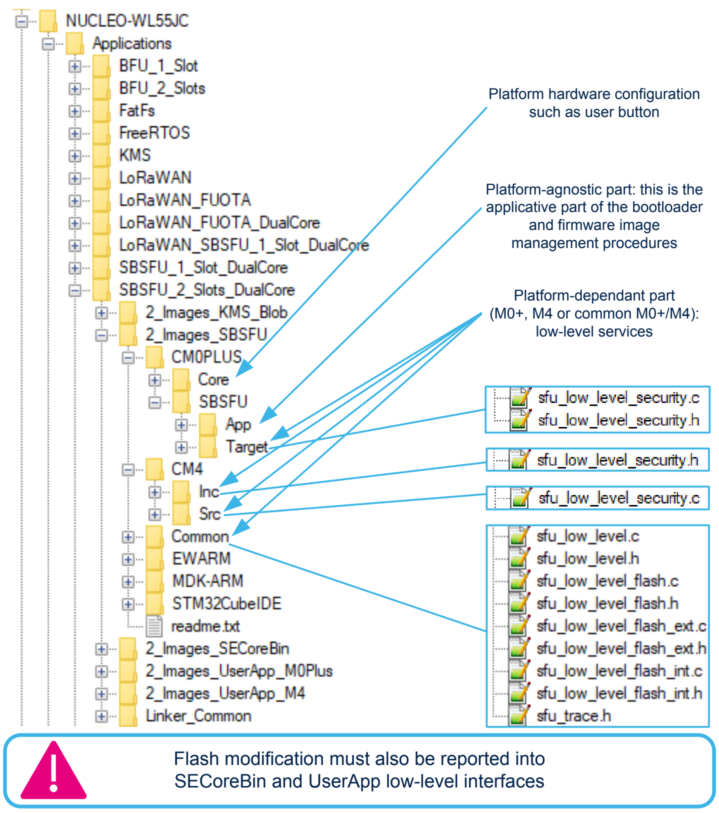
The figure below presents the BFU single-core project structure, together with the location of the files where porting changes are expected.

Figure 1. BFU project structure (single core, attack surface reduction)



The figure below presents the SBSFU dual-core project structure, together with the location of the files where porting changes are expected.

Figure 2. SBSFU project structure (dual core)



2.2 Memory mapping definition

As already highlighted in the STM32CubeWL SBSFU user manual (UM2767), a key aspect is the placement of all elements inside the Flash memory of the device:

- Secure Engine: protected environment to manage all critical data and operations
- Cortex-M4 Secure Boot
- Cortex-M0+ SBSFU
- Active slots: contain active firmwares (firmware header if contiguous as in the BFU single-core configuration + firmware)
- Firmware header (if not contiguous as in the SBSFU dual-core configuration): Flash memory area where is stored the not-contiguous firmware header
- Download slot: stores the downloaded firmware (firmware header + encrypted firmware) to be installed at next reboot
- Swap area: Flash memory area used to swap the content of active and download slots during the installation process

The figures below present the Flash memory mappings illustrated by the NUCLEO-WL55JC examples.

Figure 3. **Memory mapping example (BFU single-core single-slot configuration)**

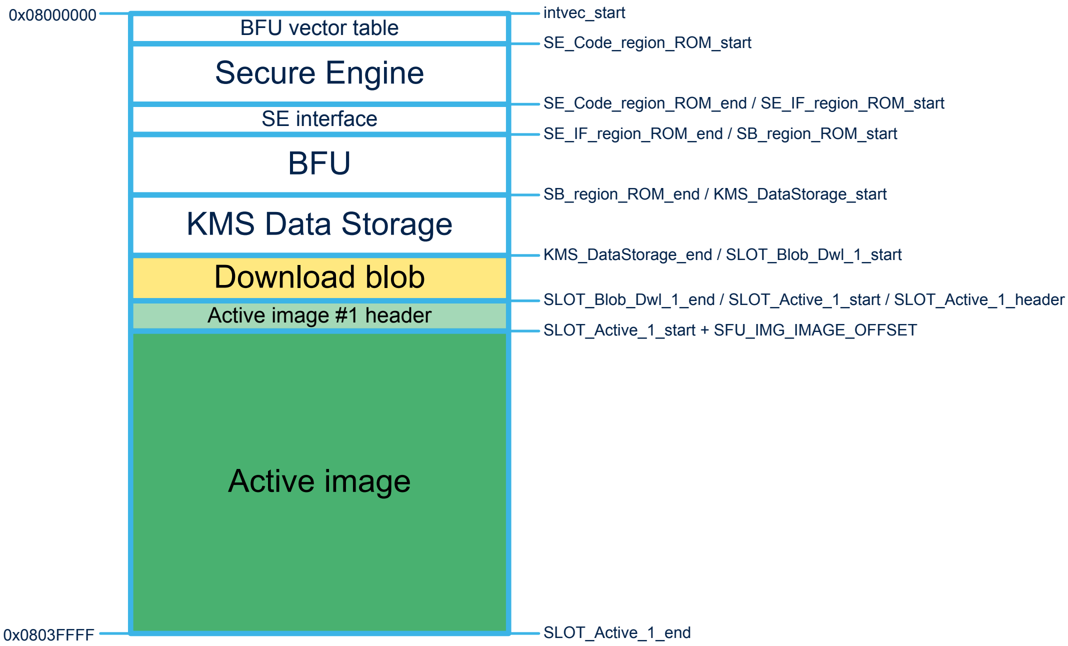


Figure 4. **Memory mapping example (BFU single-core dual-slot configuration)**

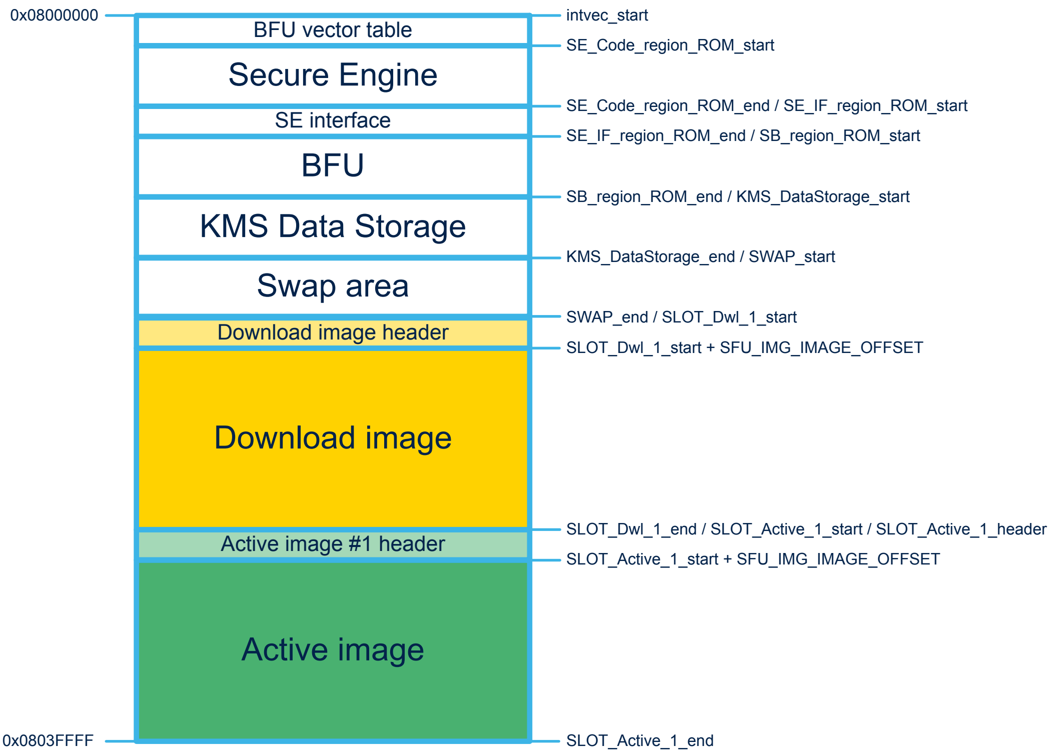


Figure 5. Memory mapping example (SBSFU dual-core single-slot configuration)

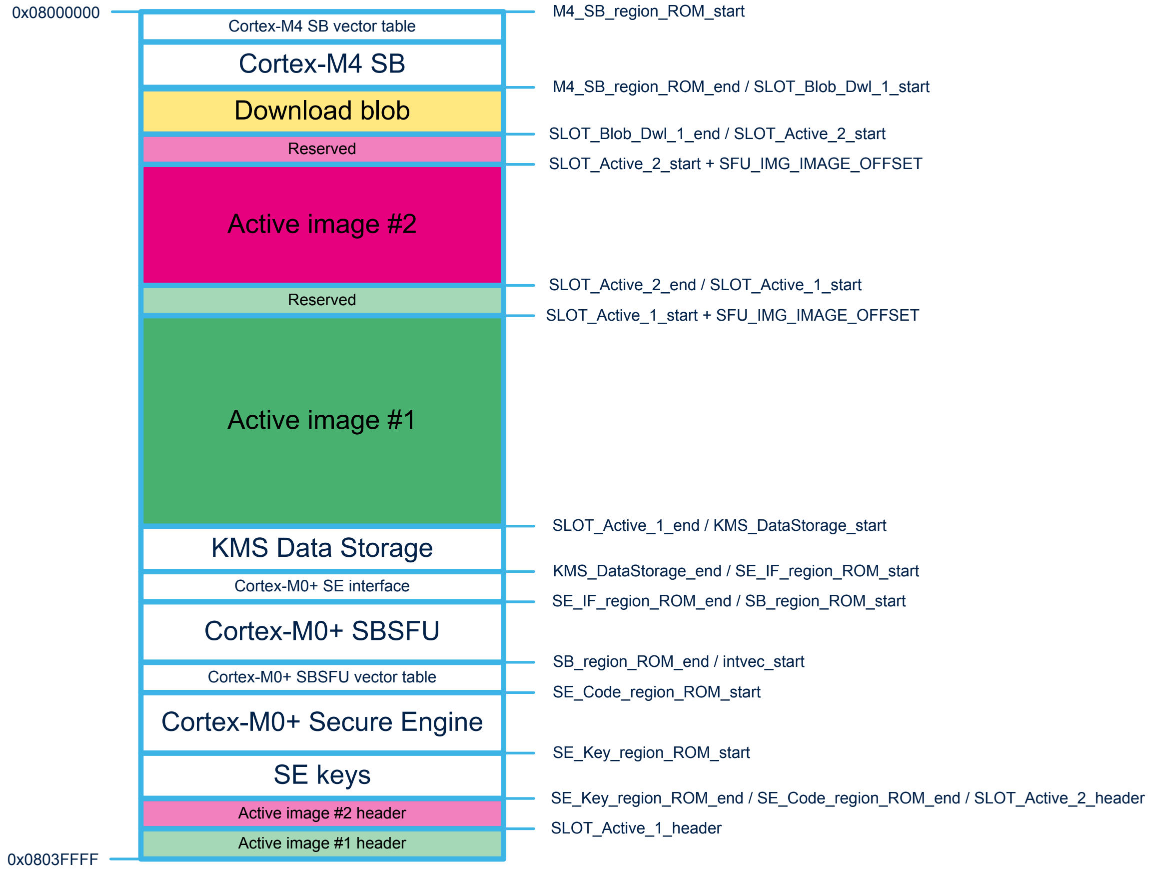
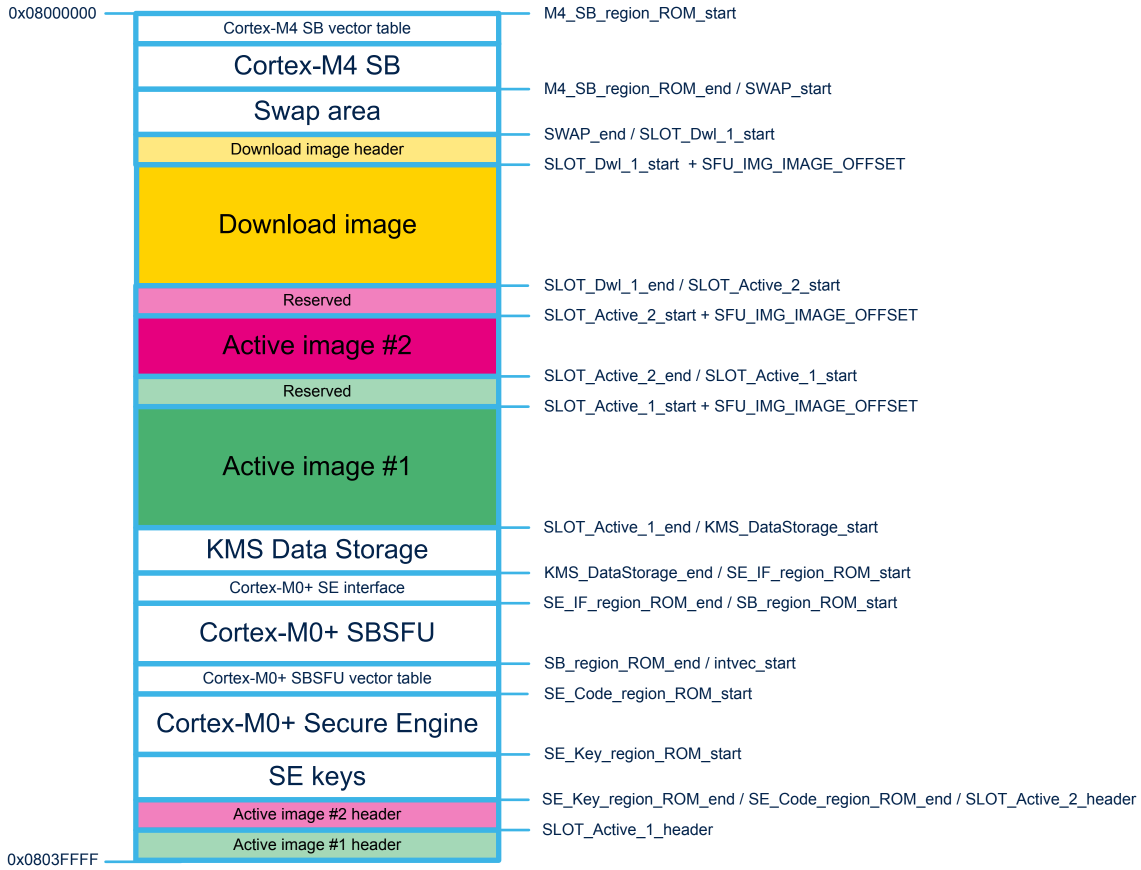


Figure 6. Memory mapping example (SBSFU dual-core dual-slot configuration)

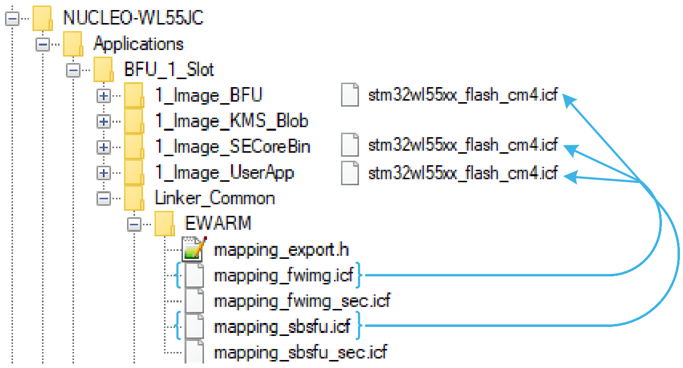


In single-core configuration, the linker file definitions shared between the three projects (SECoreBin, BFU, UserApp) are grouped in the `Linker_Common` folder as shown in the figure below:

- `mapping_fwimg.icf` contains firmware image definitions such as active slots, download slots, and swap area.
- `mapping_sbsfu.icf` contains BFU definitions such as `SE_Code_region`, `SE_Key_region`, and `SE_IF_region`.
- `mapping_export.h` exports the symbols from `mapping_sbsfu.icf` and `mapping_fwimg.icf` to the BFU applications.

Each region can be extended when adding more code is needed, or shifted to another address as long as the resulting security settings satisfy security requirements.

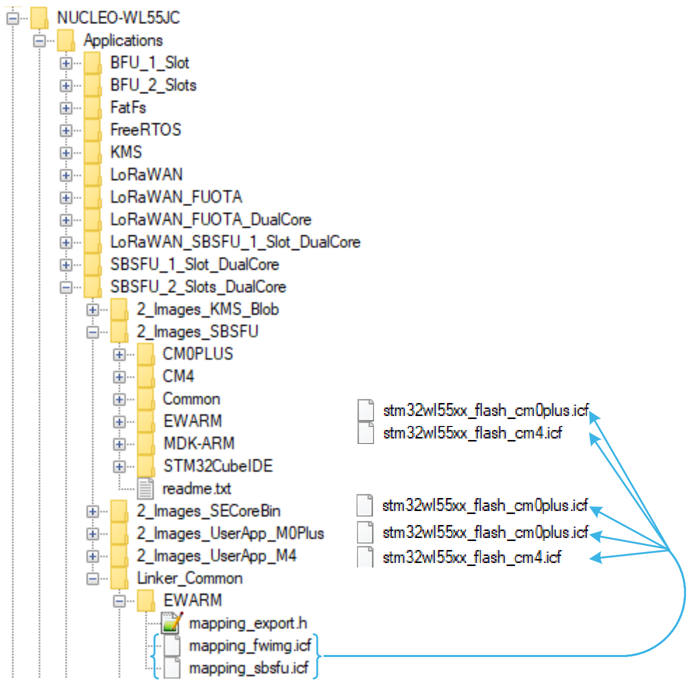
Figure 7. Linker file architecture (BFU single-core configuration)



Linker file definitions shared by all three projects

In dual-core configuration, the principle is the same but there are more projects (SECoreBin, SBSFU including the Cortex-M4 Secure Boot, UserApp_M0PLUS and UserApp_M4), as presented in the figure below.

Figure 8. Linker file architecture (SBSFU dual-core configuration)



Linker file definitions shared by all four projects

The security peripheral configuration (RDP, WRP and, for dual-core configuration only, secure memory, HDP, TZSC, TZIC, Cortex-M0+ boot address, Cortex-M0+ debug) is automatically computed based on the SBSFU/BFU linker symbols, except for the MPU configuration, due to the following constraints:

- Each MPU region base-address must be a multiple of the MPU region size.
- Each MPU region can be divided into eight sub-regions to adjust the size.

The mapping constraints with the MPU configuration are illustrated in Figure 9.

In single-core configuration (attack surface reduction when enabled), the BFU linker symbols used for WRP zone must be 2-Kbyte aligned (SB_region_ROM_end).

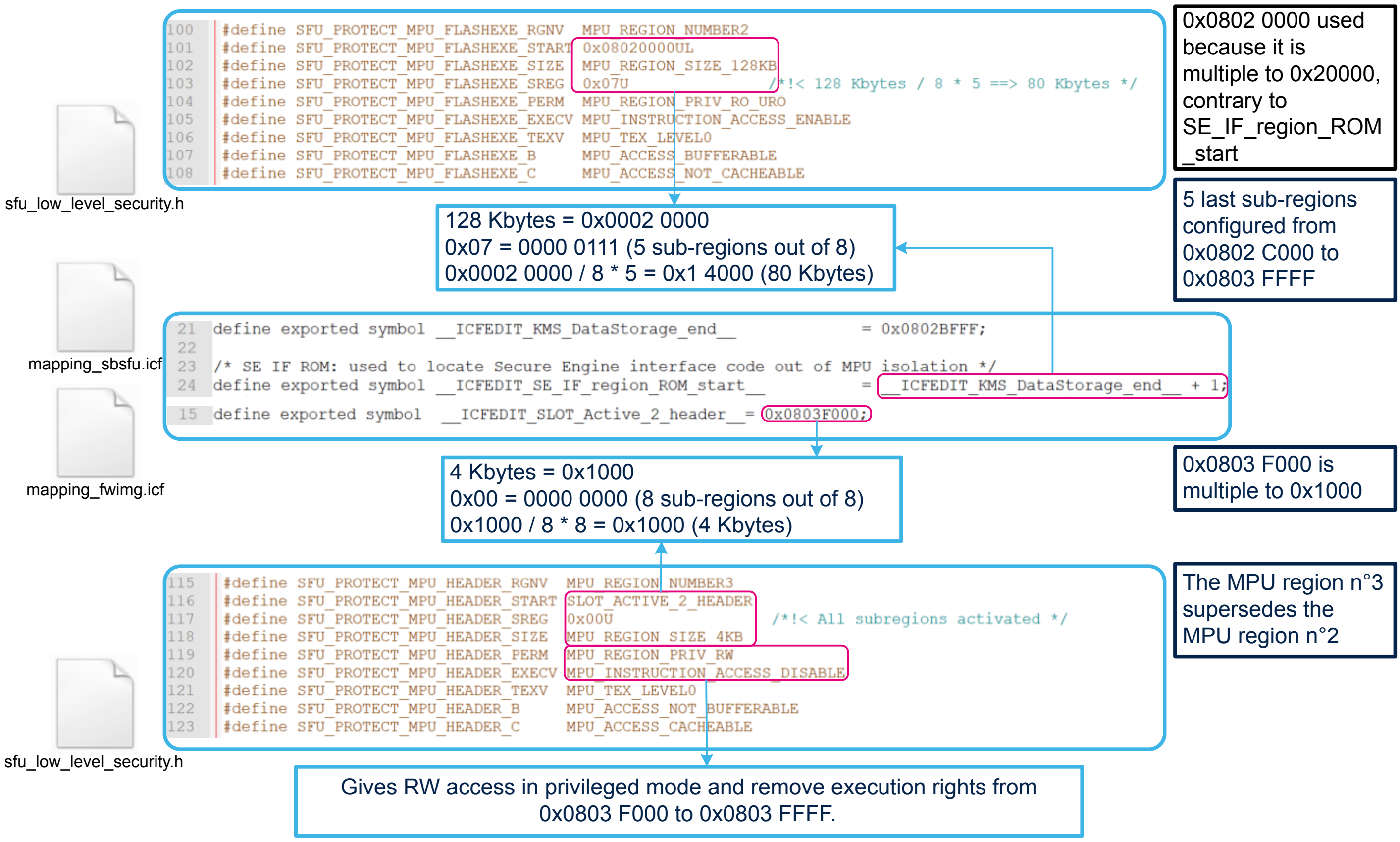
In dual-core configuration, the SBSFU linker symbols must respect the following constraints:

- 2-Kbyte alignment for Flash frontiers (WRP, secure Flash, HDP, TZSC)
  – Cortex-M4 WRP: M4_SB_region_ROM_start and M4_SB_region_ROM_end
  – Cortex-M0+ WRP: SE_IF_region_ROM_start and SE_Key_region_ROM_end
  – Cortex-M0+ secure Flash: SLOT_Active_1_start
- 1-Kbyte alignment for RAM frontiers (secure RAM, TZSC)
  – Cortex-M0+ secure RAM: SRAM2_BASE
  – Cortex-M0+ TZSC (privileged): SE_region_RAM_start

The figure below shows a typical case where the declared MPU region is larger than the concerned memory area to respect the memory constraints:

- The first sub-regions are disabled.
- The end of the MPU region is overwritten by a MPU region with a higher index.

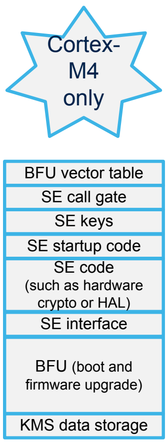
Figure 9. Mapping constraint with MPU configuration



2.2.1 Parameters for SBSFU region definition

The figures below present the parameters in the `mapping_sbsfu.icf` file, that are used for the configuration of the BFU/SBSFU regions.

Figure 10. **BFU regions (mapping_sbsfu.icf from BFU single-core project)**

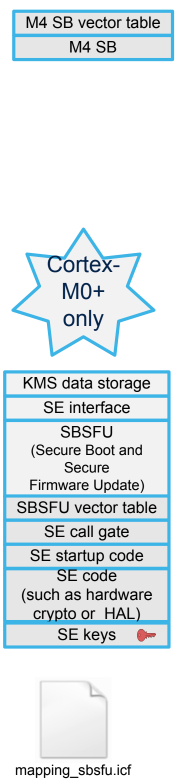


- Offsets to allow auto-adjustment when updating a size : BFU code setting the protections takes it into account.
  ➔ It is user's responsibility to verify the protection during product validation.
- Absolute values used in case of constraints (as for MPU configuration)
- Regions start address must be 256 byte-aligned.
- If the attack surface reduction is enabled, SB_region_ROM_start/end must be 2-Kbyte aligned.
- There is no SE isolation with the BFU project, only an attack surface reduction when the feature is enabled.

```
/* Vector table */
define exported symbol __ICFEDIT_intvec_start__                   = 0x08000000;
define exported symbol __ICFEDIT_Vector_size__                    = 0x200;

/* SE Code region */
define exported symbol __ICFEDIT_SE_Code_region_ROM_start__       = __ICFEDIT_intvec_start__ + __ICFEDIT_Vector_size__;
define exported symbol __ICFEDIT_SE_CallGate_region_ROM_start__   = __ICFEDIT_SE_Code_region_ROM_start__ + 4;
define exported symbol __ICFEDIT_SE_CallGate_Region_ROM_End__     = __ICFEDIT_SE_Code_region_ROM_start__ + 0x1FF;

/* SE key region */
define exported symbol __ICFEDIT_SE_Key_region_ROM_start__        = __ICFEDIT_SE_CallGate_Region_ROM_End__ + 1;
define exported symbol __ICFEDIT_SE_Key_region_ROM_end__          = __ICFEDIT_SE_Key_region_ROM_start__ + 0x2FF;

/* SE Startup */
define exported symbol __ICFEDIT_SE_Startup_region_ROM_start__    = __ICFEDIT_SE_Key_region_ROM_end__ + 1;
define exported symbol __ICFEDIT_SE_Code_nokey_region_ROM_start__ = __ICFEDIT_SE_Startup_region_ROM_start__ + 0x100;
define exported symbol __ICFEDIT_SE_Code_region_ROM_end__         = __ICFEDIT_SE_Startup_region_ROM_start__ + 0x74FF;

/* SE IF ROM: used to locate Secure Engine interface code */
define exported symbol __ICFEDIT_SE_IF_region_ROM_start__         = __ICFEDIT_SE_Code_region_ROM_end__ + 1;
define exported symbol __ICFEDIT_SE_IF_region_ROM_end__           = __ICFEDIT_SE_IF_region_ROM_start__ + 0x7FF;

/* SBSFU Code region */
define exported symbol __ICFEDIT_SB_region_ROM_start__            = __ICFEDIT_SE_IF_region_ROM_end__ + 1;
define exported symbol __ICFEDIT_SB_region_ROM_end__              = 0x0800CFFF;

/* KMS Data Storage (NVMS) region */
/* KMS Data Storage need for 2 images : 4 kbytes * 2 ==> 8 kbytes */
define exported symbol __ICFEDIT_KMS_DataStorage_start__          = 0x0800D000;
define exported symbol __ICFEDIT_KMS_DataStorage_end__            = 0x0800EFFF;
```

mapping_sbsfu.icf

Figure 11. **SBSFU regions (mapping_sbsfu.icf from SBSFU dual-core project)**




```
/* M4 SB Code region */
define exported symbol __ICFEDIT_M4_SB_region_ROM_start__         = 0x08000000;
define exported symbol __ICFEDIT_M4_SB_region_ROM_end__           = 0x080057FF;
```

- Offsets to allow auto-adjustment when updating a size : SBSFU code setting the protections takes it into account.
  ➔ It is user's responsibility to verify the protection during product validation.
- Absolute values used in case of constraints (as for MPU configuration)
- Regions start address must be 256-byte aligned.
- M4_SB_region_ROM_start/end, SE_IF_region_ROM_start/end, SLOT_Active_1_start, SE_Key_region_ROM_start and intvec_start must be 2-Kbyte aligned.
- With the STM32WL, SE isolation is ensured by the MPU (read only, execution rights) and the TZSC (privileged mode) protections.

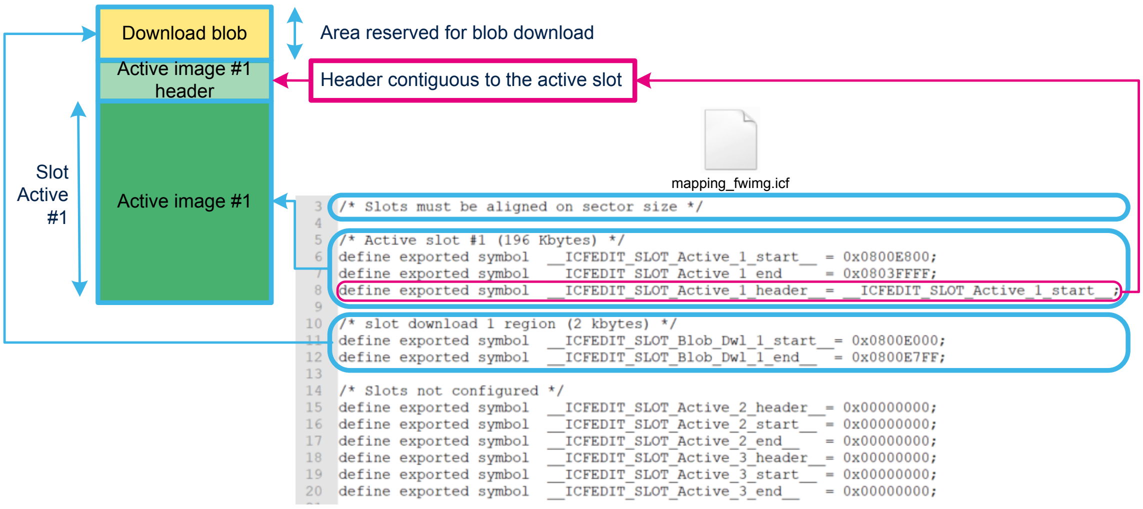
```
/* KMS Data Storage (NVMS) region protected area */
/* KMS Data Storage need for 2 images : 4 kbytes * 2 ==> 8 kbytes */
define exported symbol __ICFEDIT_KMS_DataStorage_start__          = 0x0802A000;
define exported symbol __ICFEDIT_KMS_DataStorage_end__            = 0x0802BFFF;

/* SE IF ROM: used to locate Secure Engine interface code out of MPU isolation */
define exported symbol __ICFEDIT_SE_IF_region_ROM_start__         = __ICFEDIT_KMS_DataStorage_end__ + 1;
define exported symbol __ICFEDIT_SE_IF_region_ROM_end__           = __ICFEDIT_SE_IF_region_ROM_start__ + 0x13FF;

/* SBSFU Code region */
define exported symbol __ICFEDIT_SB_region_ROM_start__            = __ICFEDIT_SE_IF_region_ROM_end__ + 1;
define exported symbol __ICFEDIT_SB_region_ROM_end__              = 0x080367FF;
```

```
/* M0 Vector table with alignment constraint on VECTOR_SIZE */
define exported symbol __ICFEDIT_intvec_start__                   = __ICFEDIT_SB_region_ROM_end__ + 1;
define exported symbol __ICFEDIT_Vector_size__                    = 0x200;

/* SE Code region protected by MPU isolation */
define exported symbol __ICFEDIT_SE_Code_region_ROM_start__       = __ICFEDIT_intvec_start__ + __ICFEDIT_Vector_size__;
define exported symbol __ICFEDIT_SE_CallGate_region_ROM_start__   = __ICFEDIT_SE_Code_region_ROM_start__ + 4;
define exported symbol __ICFEDIT_SE_CallGate_Region_ROM_End__     = __ICFEDIT_SE_Code_region_ROM_start__ + 0x1FF;

/* SE Startup */
define exported symbol __ICFEDIT_SE_Startup_region_ROM_start__    = __ICFEDIT_SE_CallGate_Region_ROM_End__ + 1;
define exported symbol __ICFEDIT_SE_Code_nokey_region_ROM_start__ = __ICFEDIT_SE_Startup_region_ROM_start__ + 0x100;

/* SE Embedded Keys */
define exported symbol __ICFEDIT_SE_Key_region_ROM_start__        = 0x0803E800;
define exported symbol __ICFEDIT_SE_Key_region_ROM_end__          = 0x0803EFFF;
define exported symbol __ICFEDIT_SE_Code_region_ROM_end__         = __ICFEDIT_SE_Key_region_ROM_end__;
```

2.2.2 Parameters for firmware image slot definition

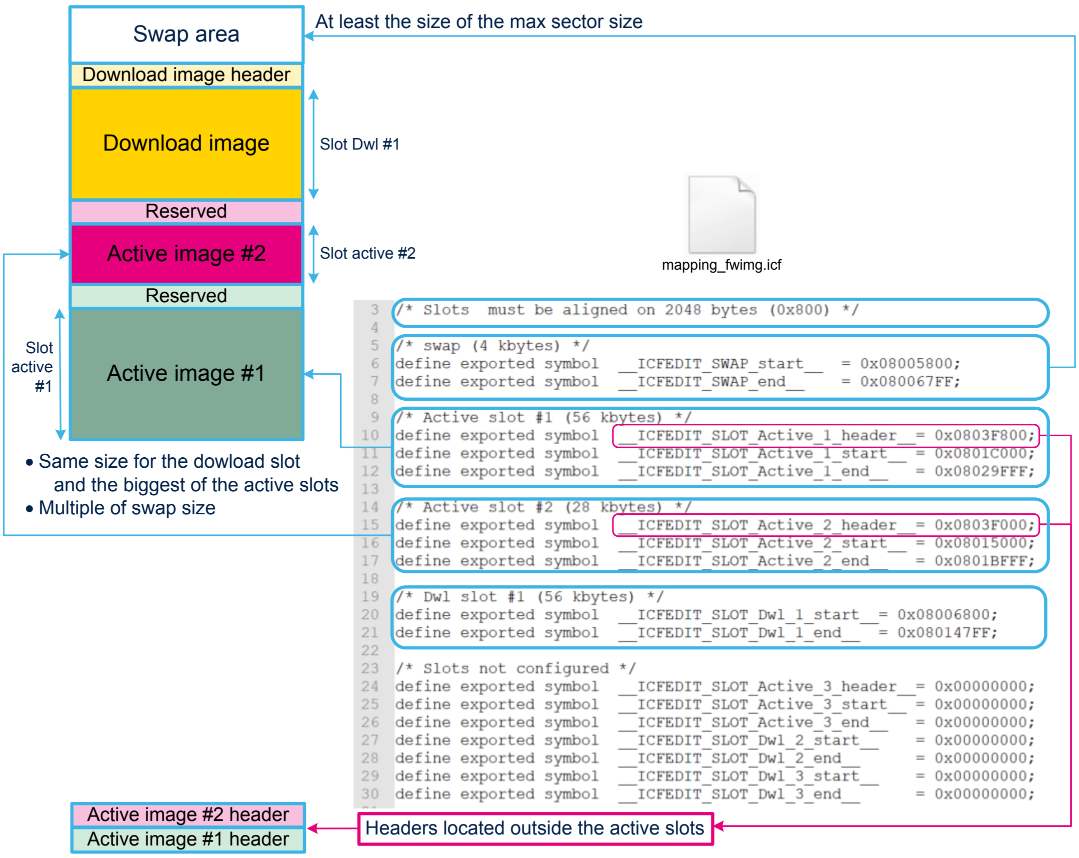
The BFU single-core single-slot project uses one active slot without dedicated download slot. The SBSFU dual-core single-slot project uses two active slots without dedicated download slot. The slot named “Blob Dwl 1” is reserved for KMS use.

Figure 12. Firmware image slot definitions (mapping_fwimg.icf from BFU single-core single-slot project)



The figure below presents the parameters that are used for the configuration of the image regions within the SBSFU dual-core dual-slot project (in the `mapping_fwimg.icf` file).

Figure 13. Firmware image slot definitions (mapping_fwimg.icf from SBSFU dual-core dual-slot project)



The compliance with SBSFU constraints requires that the following conditions are met:

- Slot areas must be aligned on the Flash memory sector size (2048 bytes = 0x800).
- The minimum SWAP size is 4 Kbytes and at least equal to the size of the largest sector.
- The size of active and download slots must be a multiple of the SWAP size.
- The size of the download slot must be equal to the size of the biggest active slot, except when using partial update feature.

With the SBSFU dual-core configuration, the headers of the active slots must be mapped at the end of the Flash memory to be protected by the HDP area at run time.

The `SFU_IMAGE_OFFSET` value depends on the STM32 microcontroller series. For the STM32WL Series, the default value (512 bytes) is used.

The addresses used for WRP, for the Flash secure memory, for HDP and for TZSC (Flash) must be aligned on the Flash memory sector size (2048 bytes).

The addresses used for RAM secure memory and TZSC (RAM) must be aligned on 1024 bytes.

Note:
- *The MPU constraint on the active slot configuration must be verified as illustrated in Figure 9.*
- *The same principle applies to the BFU single-core dual-slot project and to the SBSFU dual-core single-slot project (see Figure 4 and Figure 5 for details about their mapping).*

2.2.3 Project-specific linker files

SECoreBin places critical code and critical data such as the secrets, as illustrated in the figure below.

Figure 14. SECoreBin specific linker file

The SBSFU linker file is in charge of the SBSFU application placement, including SECoreBin binary, as shown in the figure below.

Figure 15. SBSFU specific linker file

```
/*************************************************************************/
/*                      placement instructions                           */
/*************************************************************************/
place at address mem: __ICFEDIT_intvec_start__   { readonly section .intvec };
place at address mem: __ICFEDIT_SE_CallGate_region_ROM_start__  { readonly section SE_CORE_Bin };
place in SE_IF_ROM_region {section .SE_IF_Code};
place in SB_ROM_region     { readonly };
place in SB_RAM_region     { readwrite, block CSTACK, block HEAP};
```

Binary generated by SECoreBin project

2_Images_SBSFU
stm32wl55xx_flash_cm0plus.icf

In all configurations, UserApp must be configured to run in the active slot (slot active start address + `SFU_IMG_IMAGE_OFFSET`) as illustrated in Figure 16 for BFU and in Figure 17/Figure 18 for SBSFU, where `SFU_IMG_IMAGE_OFFSET` is 512 bytes.

Figure 16. 1_Image_UserApp specific linker file (BFU single core)

```
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__      = __ICFEDIT_SLOT_Active_1_start__ + 512;
define symbol __ICFEDIT_region_ROM_end__        = __ICFEDIT_SLOT_Active_1_end__  ;
define symbol __ICFEDIT_region_RAM_start__      = __ICFEDIT_SE_region_RAM_end__ + 1;
define symbol __ICFEDIT_region_RAM_end__        = 0x20017FFF;

/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x800;
define symbol __ICFEDIT_size_heap__ = 0x200;

define region ROM_region      = mem:[from __ICFEDIT_region_ROM_start__ to __ICFEDIT_region_ROM_end__];
define region RAM_region      = mem:[from __ICFEDIT_region_RAM_start__ to __ICFEDIT_region_RAM_end__];

/* to make sure the binary size is a multiple of the AES block size (16 bytes) and WL flash writing unit (8 bytes) */
define root section aes_block_padding with alignment=16
{
udata8 "Force Alignment";
pad_to 16;
};
```

UserApp must be configured to run from active slot start address + SFU_IMG_OFFSET (512).

RAM used by SE cannot be reused.

1_Image_UserApp
stm32wl55xx_flash_cm4.icf

Firmware size must be a multiple of AES block size and Flash writing unit.

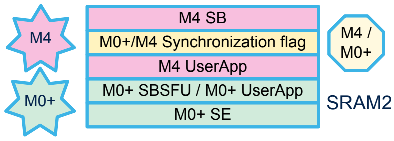
Figure 17. 2_Images_UserApp_M4 specific linker file (SBSFU dual core)

2_Images_UserApp_M4 stm32wl55xx_flash_cm4.icf

```
/*-Memory Regions-*/
define symbol __ICFEDIT_region_M4_APP_ROM_start__  = __ICFEDIT_SLOT_Active_2_start__ + 512;

/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x800;
define symbol __ICFEDIT_size_heap__  = 0x200;

define region M4_APP_ROM_region      = mem:[from __ICFEDIT_region_M4_APP_ROM_start__ to __ICFEDIT_SLOT_Active_2_end__];

/* to make sure the binary size is a multiple of the AES block size (16 bytes) and WL flash writing unit (8 bytes) */
define root section aes_block_padding with alignment=16
{
udata8 "Force Alignment";
pad_to 16;
};
```

M4 UserApp must be configured to run from active slot start address + SFU_IMG_OFFSET (512)

M4 firmware size should be multiple of AES block size and flash writing unit



SRAM2 is reserved to M0+

mapping_sbsfu.icf

SRAM1

```
/* M4 SB */
define symbol __ICFEDIT_M4_SB_region_RAM_start__                  = 0x20000000;
define symbol __ICFEDIT_M4_SB_region_RAM_end__                    = __ICFEDIT_M4_SB_region_RAM_start__ + 0xCDF;

/* M0+/M4 Synchronization flag */
define exported symbol __ICFEDIT_M4_M0PLUS_FLAG_region_RAM_start__ = __ICFEDIT_M4_SB_region_RAM_end__ + 1;
define exported symbol __ICFEDIT_M4_M0PLUS_FLAG_region_RAM_end__   = __ICFEDIT_M4_M0PLUS_FLAG_region_RAM_start__ + 0x1F;

/* M4 UserApp */
define symbol __ICFEDIT_M4_APP_region_RAM_start__                 = __ICFEDIT_M4_M0PLUS_FLAG_region_RAM_end__ + 1;
define symbol __ICFEDIT_M4_APP_region_RAM_end__                   = 0x20007FFF;
```

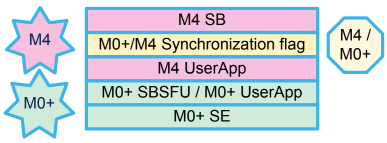
Figure 18. 2_Images_UserApp_M0Plus specific linker file (SBSFU dual core)

```
/*-Memory Regions-*/
define symbol __ICFEDIT_region_M4_start__       = __ICFEDIT_SLOT_Active_2_start__ + 512;
define symbol __ICFEDIT_region_M4_end__         = __ICFEDIT_SLOT_Active_2_end__;
define symbol __ICFEDIT_region_ROM_start__      = __ICFEDIT_SLOT_Active_1_start__ + 512;
define symbol __ICFEDIT_region_ROM_end__        = __ICFEDIT_SLOT_Active_1_end__;
define symbol __ICFEDIT_region_RAM_start__      = __ICFEDIT_SB_region_RAM_start__;
define symbol __ICFEDIT_region_RAM_end__        = __ICFEDIT_SB_region_RAM_end__;

/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x800;
define symbol __ICFEDIT_size_heap__  = 0x200;

define region M4_region      = mem:[from __ICFEDIT_region_M4_start__  to __ICFEDIT_region_M4_end__];
define region ROM_region     = mem:[from __ICFEDIT_region_ROM_start__ to __ICFEDIT_region_ROM_end__];
define region RAM_region     = mem:[from __ICFEDIT_region_RAM_start__ to __ICFEDIT_region_RAM_end__];

/* to make sure the binary size is a multiple of the AES block size (16 bytes) and WL flash writing unit (8 bytes) */
define root section aes_block_padding with alignment=16
{
udata8 "Force Alignment";
pad_to 16;
};
```

UserApp must be configured to run from active slot start address + SFU_IMG_OFFSET (512).

2_Images_UserApp_M0Plus stm32wl55xx_flash_cm0plus.icf

Firmware size must be a multiple of AES block size and Flash writing unit.



SRAM1 is reserved to Cortex-M4 and to the synchronisation flag.

mapping_sbsfu.icf

SRAM2

```
/* SBSFU RAM region */
define exported symbol __ICFEDIT_SB_region_RAM_start__        = __ICFEDIT_M4_APP_region_RAM_end__ + 1;
define exported symbol __ICFEDIT_SB_region_RAM_end__          = 0x2000D3FF;

/* SE RAM region protected area with 1 kBytes alignment constraint (TZIC) ==> 0x2000D400 */
define exported symbol __ICFEDIT_SE_region_RAM_start__        = __ICFEDIT_SB_region_RAM_end__ + 1;
define exported symbol __ICFEDIT_SE_region_RAM_stack_top__    = 0x2000DB00; /* Secure Engine's private stack */
define exported symbol __ICFEDIT_SE_region_RAM_end__          = 0x2000FFFF;
```

Protected RAM used by SE (TZSC) cannot be re-used

2.2.4 Multiple-image configuration

Up to three active slots (SFU_NB_MAX_ACTIVE_IMAGE) and three download slots (SFU_NB_MAX_DWL_AREA) can be configured.

During the installation process, the active slot is identified with the SFU magic tag inside the firmware image header (SFU1, SFU2, or SFU3).

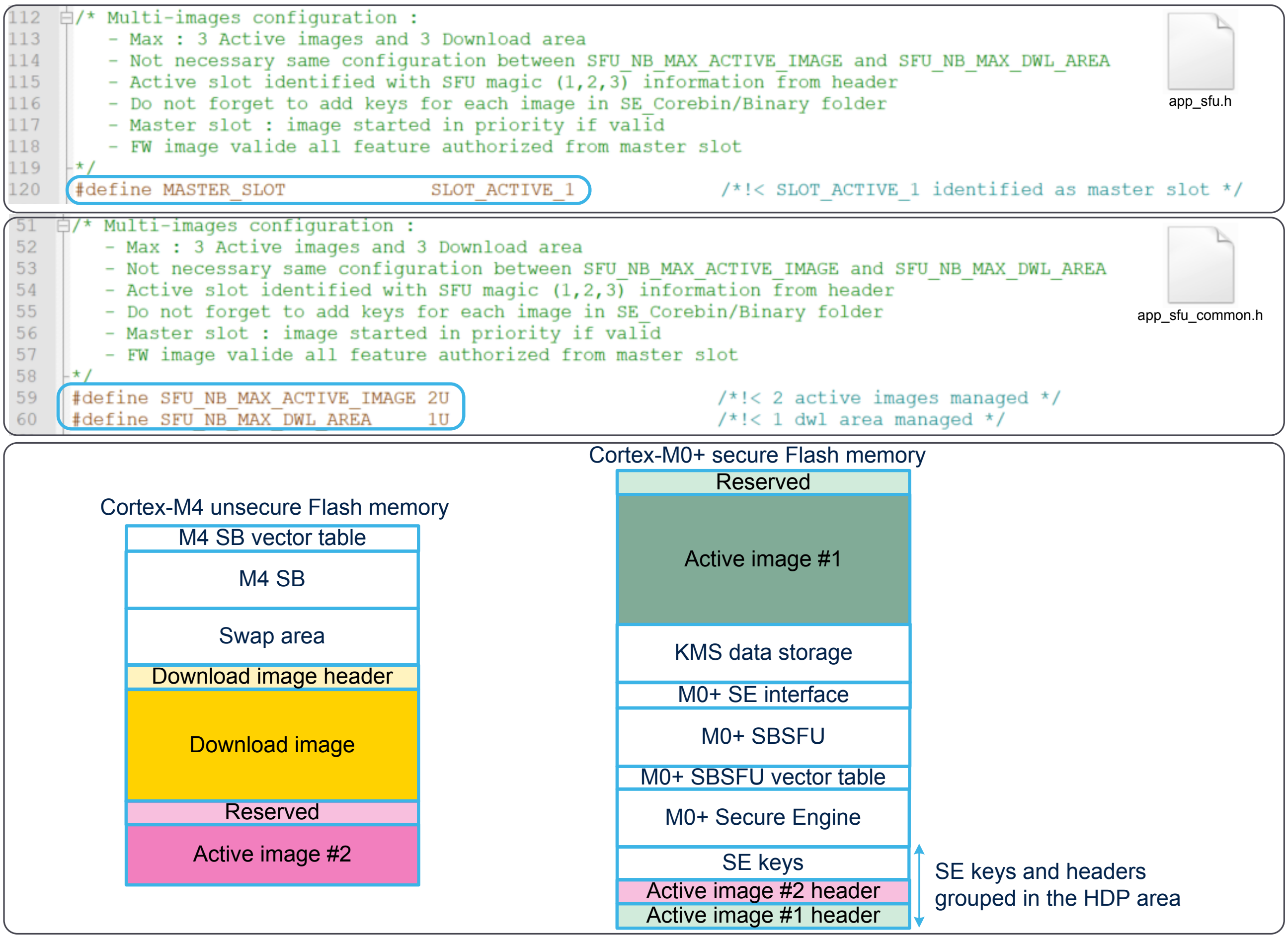
In the SBSFU_2_Slots_DualCore application, a single download slot is configured for the two active slots to optimize the memory footprint.

At boot, after verification of the authenticity and integrity of all firmware images, the Cortex-M0+ SBSFU jumps into the active firmware image located inside the MASTER_SLOT while the Cortex-M4 Secure Boot jumps into the other available active firmware image. If two firmware images are not available, no jump is done, and the download process is started instead (if a loader is available on the Cortex-M4 side).

As a constraint, all the headers must be grouped in the same area to be protected inside the isolated environment. Each header must be in its own Flash memory sector.

The figure below shows the multiple-image configuration provided in the SBSFU_2_Slots_DualCore application.

Figure 19. Multiple-image configuration



In the SBSFU_1_Slot_DualCore application, there is no download slot associated to the active slots and the loader is available on the Cortex-M0+ side.

The following configuration is available in `Firmware/Projects/NUCLEO-WL55JC/Applications/SBSFU_1_Slot_DualCore/2_Images_SBSFU/CM0PLUS/SBSFU/App/app_sfu.h`:

```
/* Multi-images configuration :
   - Max : 3 Active images
   - Do not forget to add keys for each image in SE_Corebin/Binary folder
   - Master slot : image started in priority if valid
*/
#define SFU_NB_MAX_ACTIVE_IMAGE 2U             /*!< 2 active images managed */
#define SFU_NB_MAX_DWL_AREA     1U             /*!< 1 blob dwl area managed */
#define MASTER_SLOT             SLOT_ACTIVE_1  /*!< SLOT_ACTIVE_1 identified as master slot
*/
```

3 SBSFU configuration

3.1 Features to be configured

The STM32CubeWL SBSFU supports:

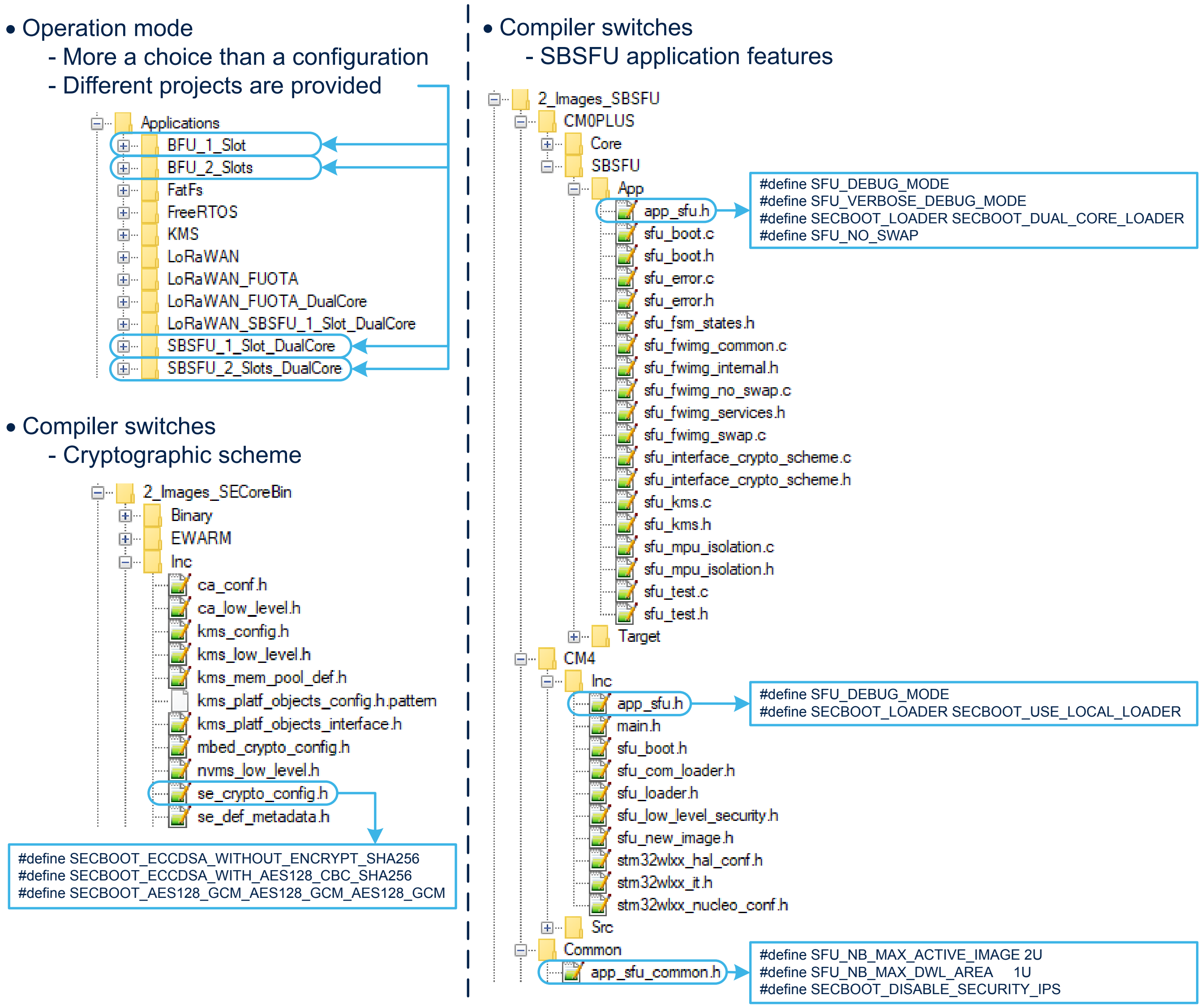
- two operation modes: dual- and single-slot configurations
- three cryptographic schemes using symmetric and asymmetric cryptographic operations

The configuration possibilities go beyond these options through compilation switches as follows:

- The local loader can be removed to reduce the memory footprint (dual-slot configuration only).
- Verbose switch can be activated to make the debug easier.
- The debug mode can be disabled (no more printf on the terminal during the SBSFU execution) to reduce the memory footprint.
- Security peripherals can be turned off to make the debug easier.
- The multiple-image configuration can be used for a complex system with multiple firmware such as protocol stack, middleware, and user application.

The figure below presents the SBSFU configuration solutions, with the related files and compilation switches.

Figure 20. SBSFU configuration



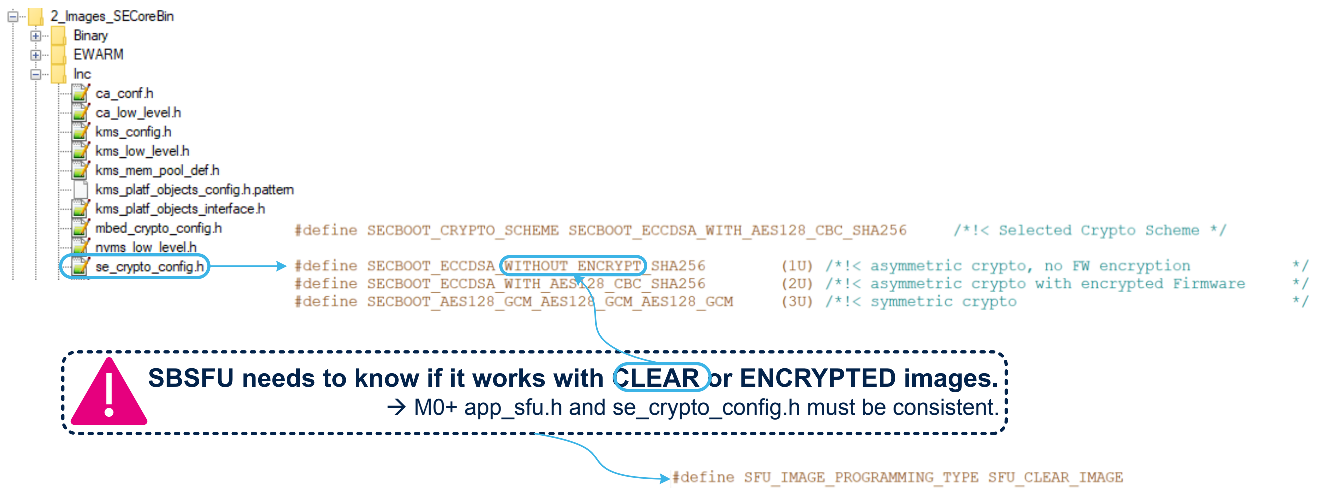
3.2 Cryptographic scheme selection

The STM32CubeWL SBSFU is delivered with the following cryptographic schemes, using both asymmetric and symmetric cryptography:

- ECDSA asymmetric cryptography for firmware verification and AES-CBC symmetric cryptography for firmware decryption
- ECDSA asymmetric cryptography for firmware verification without firmware encryption.
- AES-GCM symmetric cryptography for both firmware verification and decryption

The selection among these schemes is done by means of the `SECBOOT_CRYPTO_SCHEME` compilation switch, as depicted in the figure below.

Figure 21. Switching the cryptographic scheme



3.3 Security configuration

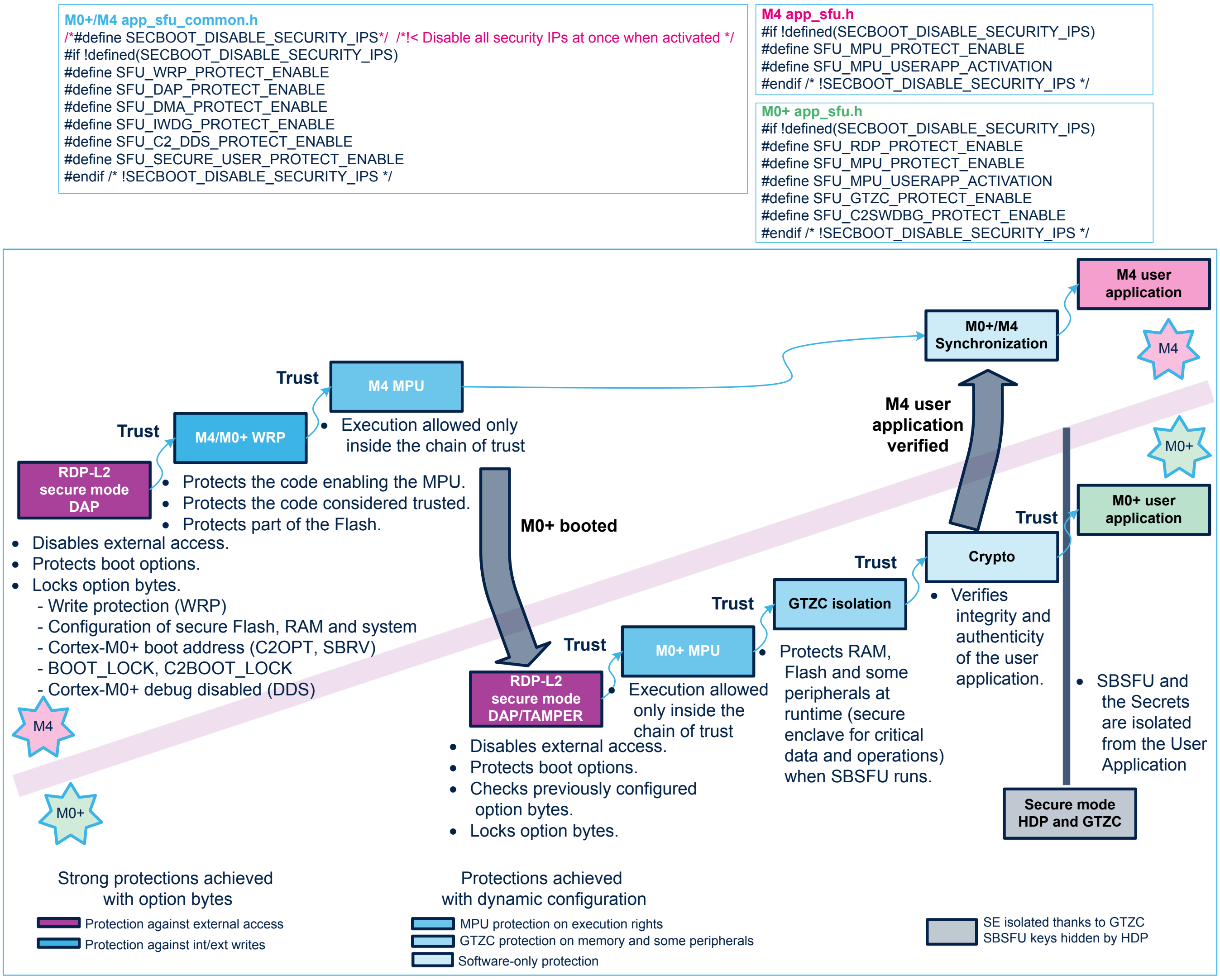
The SBSFU example is delivered with the STM32 security protection configuration that is used to protect secrets against both outer and inner attacks.

STM32 security peripherals can be deactivated independently as per user's decision, to achieve a different protection level (for example, GTZC TZSC and TZIC allow the activation of protections against inner attacks). Any STM32 security configuration modification requires a security protection evaluation at system product level, to ensure that protections are well set according to product constraints and specifications.

During the development phase, the disabling of all peripherals may be required to make debugging easier.

The figure below shows the various security configuration solutions available in the files: `M4 app_sfu.h`, `M0+ app_sfu.h` and `M4/M0+ app_sfu_common.h`

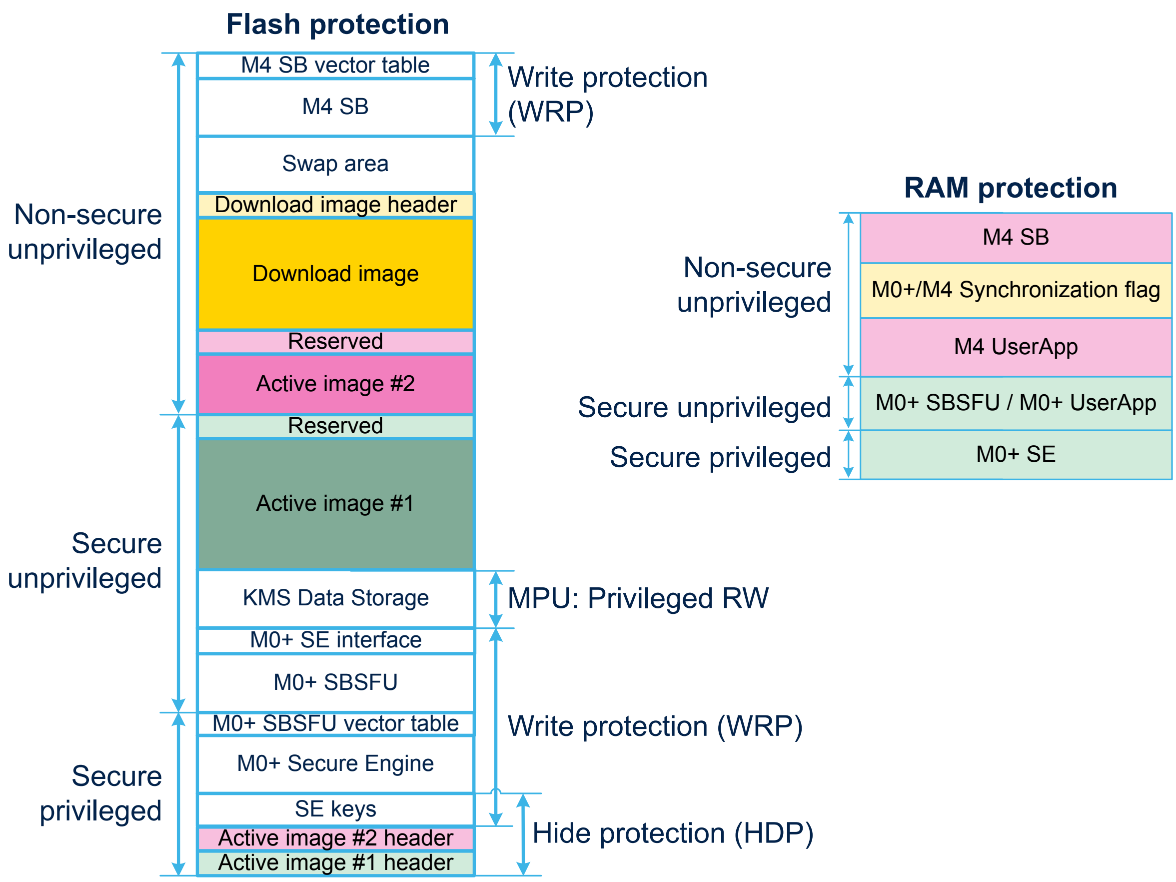
Figure 22. Security configuration (`M4 app_sfu.h`, `M0+ app_sfu.h` and `M4/M0+ app_sfu_common.h`)



Note: When the RDP Level 2 is enabled, the secure Cortex-M0+ can still change the option bytes (STM32CubeWL SBSFU does not offer this change possibility). The RDP Level 2 strengthens the protection of the option bytes as the secure Cortex-M0+ is the only one authorized to change them: neither the non-secure Cortex-M4 nor external tools like STM32CubeProgrammer can change them anymore.

The figure below shows the main Flash memory and RAM protections. The attack surface reduction ensured by the Cortex-M4 and Cortex-M0+ MPUs, is not detailed here.

Figure 23. Flash and RAM protections (except attack surface reduction)



3.4 Development or production mode configuration

The first step before any code modification is often to configure the SBSFU project in development mode, to enable the IDE debug facilities and add SBSFU debug traces:

1. Deactivate all security protections `SFU_xxx_PROTECT_ENABLE`.
2. Deactivate `SFU_FINAL_SECURE_LOCK_ENABLE`.
3. Activate `SFU_FWIMG_BLOCK_ON_ABNORMAL_ERRORS_MODE`.
4. Activate `SECBOOT_OB_DEV_MODE`.
5. Activate the verbose mode `SFU_VERBOSE_DEBUG_MODE` (optional, see Section 5.2 for details on the impact on mapping).

At the end of the development phase, the SBSFU project must be configured in production mode for the final release:

1. Activate all required security protections `SFU_xxx_PROTECT_ENABLE`.
2. Deactivate verbose mode: `SFU_VERBOSE_DEBUG_MODE`.
3. Deactivate `SFU_FWIMG_BLOCK_ON_ABNORMAL_ERRORS_MODE`.
4. Deactivate `SECBOOT_OB_DEV_MODE`.
5. Activate `SFU_FINAL_SECURE_LOCK_ENABLE` to configure the RDP Level 2.
6. Deactivate `SFU_DEBUG_MODE` to remove all prints of SBSFU that can be valuable information for an attacker.

The RDP Level 2 is **mandatory** to achieve the highest level of protection and to implement a Root of Trust. It is the user's responsibility to activate it in the final software to be programmed during the product manufacturing stage.

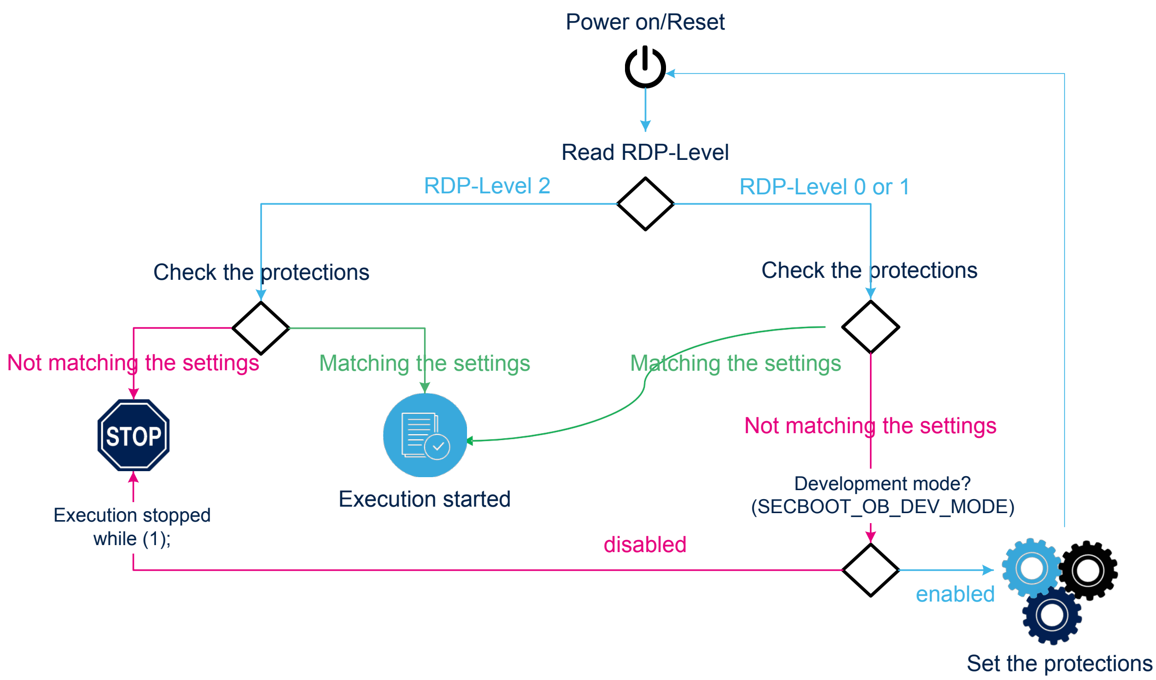
In production mode, the Secure Boot checks the option byte values (RDP, WRP, secure mode configuration, C2OPT, SBRV, BOOT_LOCK, C2BOOT_LOCK, DDS) and blocks the execution in case a wrong configuration is detected.

Caution:
- The option bytes must be configured to the production mode values by means of STM32CubeProgrammer (STM32CubeProg), just after programming the software during the production stage. If this is not done, the device remains unsecured (refer to the UM2237 for the way to use STM32CubeProgrammer).
- The secure Cortex-M0+ is able to perform a regression from RDP Level 2 to RDP Level 0 (use case not supported by X-CUBE-SBSFU). This direct regression is not recommended as there is no mass erase in this case. If needed, only regression from RDP Level 2 to RDP Level 1 can be performed (partial mass erase also).

Note: The secure Cortex-M0+ remains able to update the option bytes. SBSFU does not use this feature.

The figure below shows how the option bytes are managed at SBSFU startup.

Figure 24. Option-byte management



4 Generate a cryptographic key

4.1 Generate a new firmware AES encryption key

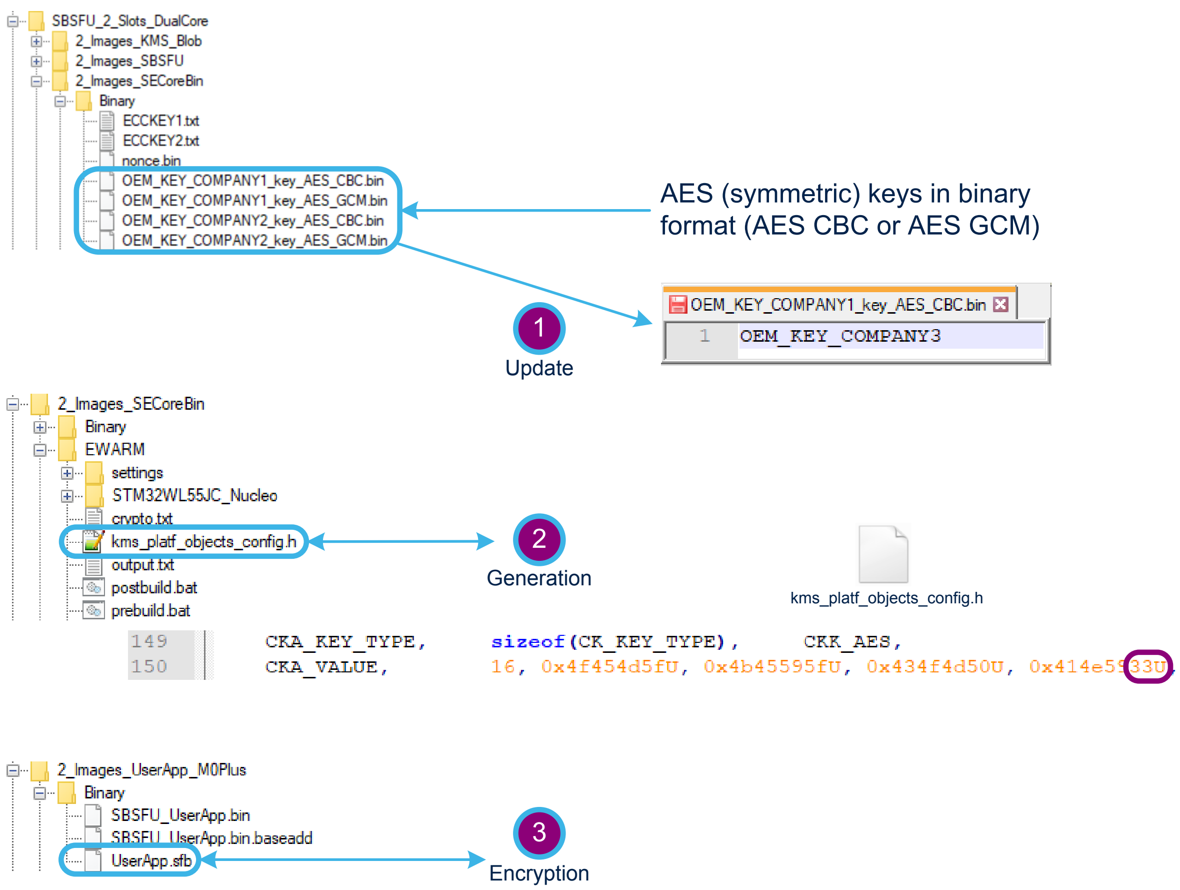
The key generation and firmware encryption are performed automatically during the compilation process with the `prebuild.bat` and `postbuild.bat` scripts (refer to the UM2767 for a detailed description of the build process).

The figure below shows the steps needed to modify the firmware encryption key of the active slot #1. The same applies for the active slot #2 and active slot #3 (if configured by the user):

1. Change the key value in the `OEM_KEY_COMPANY1_keys_AES_xxx.bin` file.
2. Compile SECoreBin: `prebuild.bat` is executed and the `kms_platf_objects_config.h` file is generated.
3. Compile Cortex-M0+ UserApp: `postbuild.bat` is executed and Cortex-M0+ UserApp is encrypted.

The same process is applied for firmware ECDSA verification key, BLOB AES encryption key, and BLOB ECDSA verification key.

Figure 25. New firmware encryption key



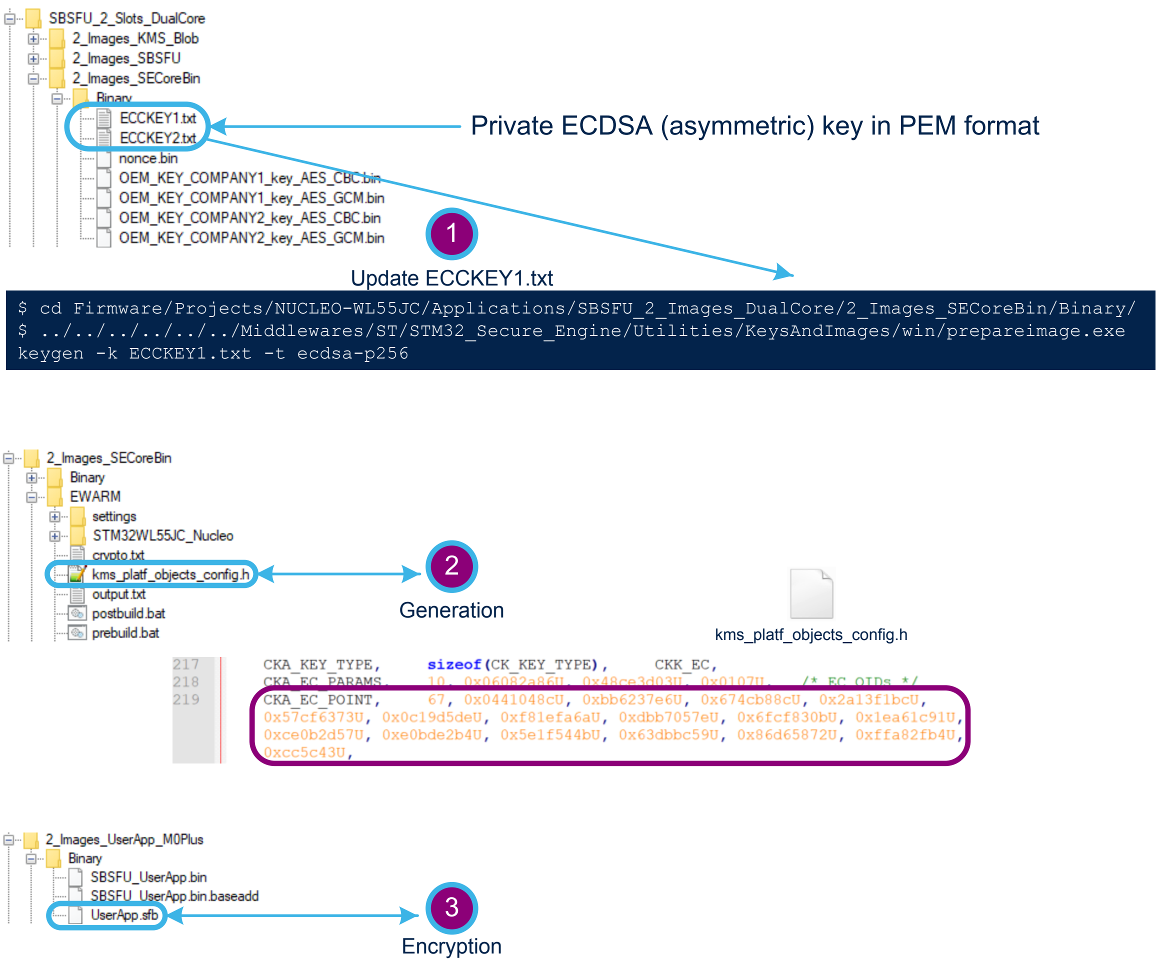
4.2 Generate a new public/private ECDSA pair of keys for firmware verification

As for the AES encryption key, the public key (`kms_platf_objects_config.h`) is automatically modified when the private key (`ECCKEY1.txt`) is changed.

The figure below shows the steps needed to modify the private and public keys for ECDSA asymmetric cryptography firmware verification of the active slot #1. The same applies for active slot #2 and active slot #3 (if configured by the user):

1. Change the key value in the `ECCKEY1.txt` file.
2. Compile SECoreBin: `prebuild.bat` is executed and the `kms_platf_objects_config.h` file is generated.
3. Compile Cortex-M0+ UserApp: `postbuild.bat` is executed and Cortex-M0+ UserApp is encrypted.

Figure 26. New private/public keys

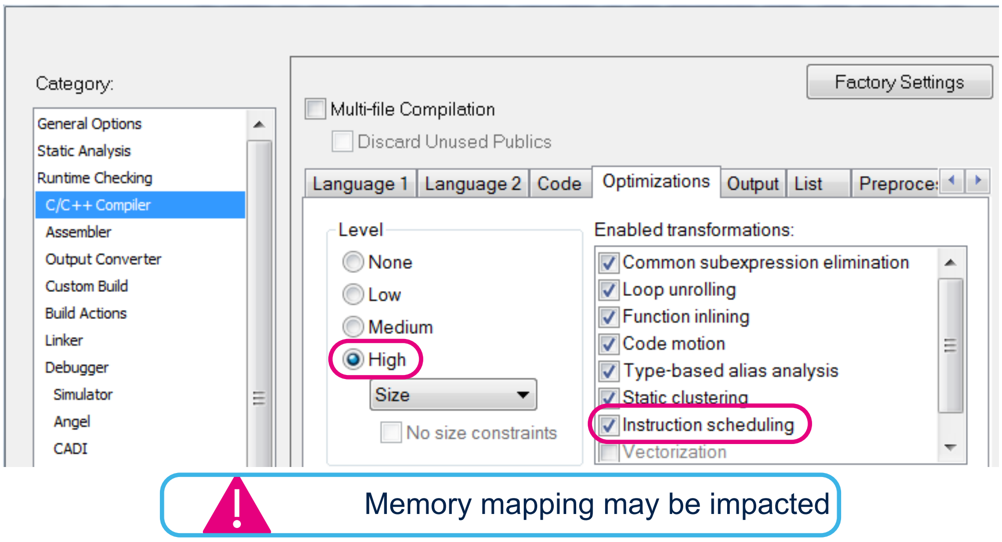


5 Tips for debug

5.1 Compiler optimizations level

Projects are delivered with the highest level of compiler optimizations turned on for size aspects. Such optimizations can make the debug complex. Changing the compiler optimization level possibly impacts the memory mapping.

Figure 27. **Compiler optimizations**



5.2 Memory mapping adaptation

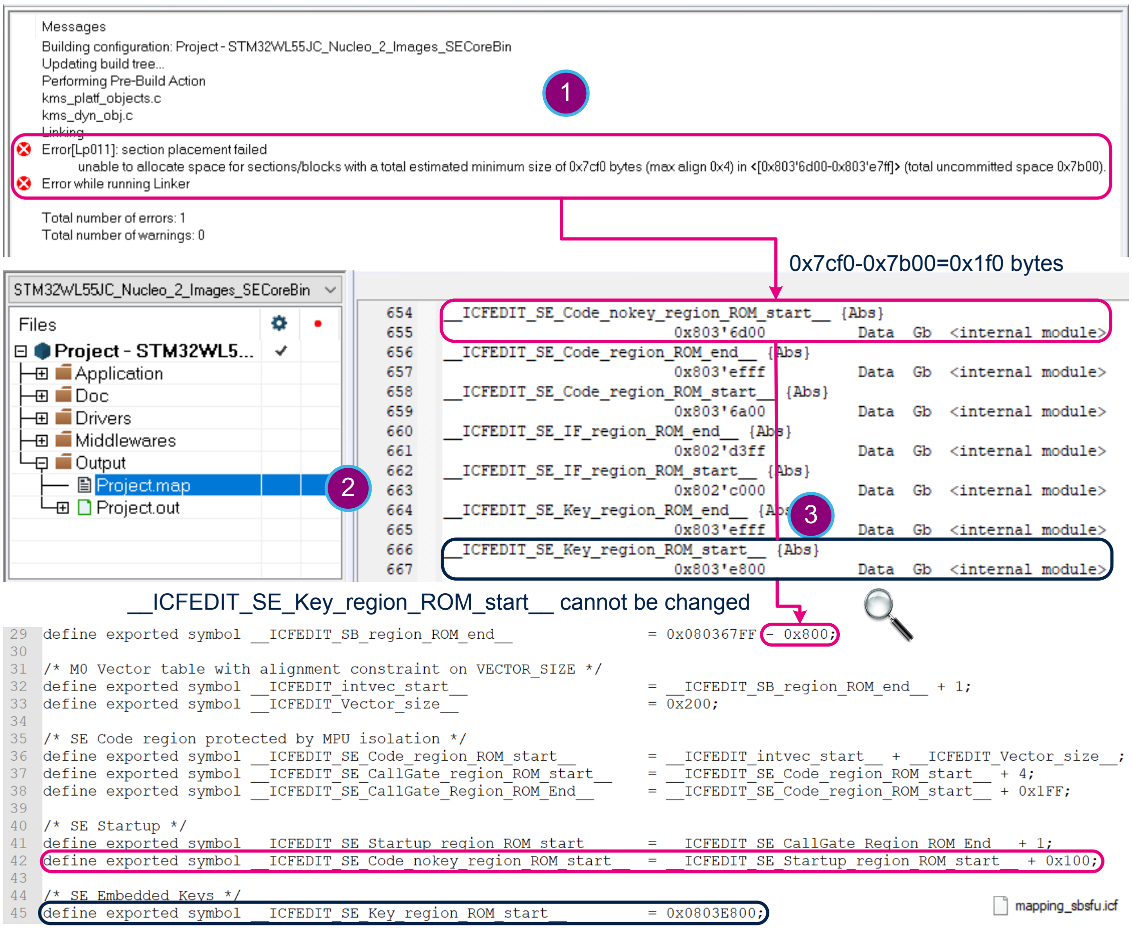
When changing the compiler optimization level or activating the development mode with the verbose compilation switch, the user may have to adapt the SBSFU memory mapping (for instance reducing firmware image slots to avoid overlap).

Caution: The security peripheral configuration (RDP, WRP, GTZC, HDP, secure memory) is automatically computed based on the SBSFU linker symbols, except for the MPU configuration due to the constraints detailed in Section 2.2 . Disabling temporarily the MPU protection can be an efficient workaround for the debug.

The figure below depicts the memory adaptation steps, based on an example:

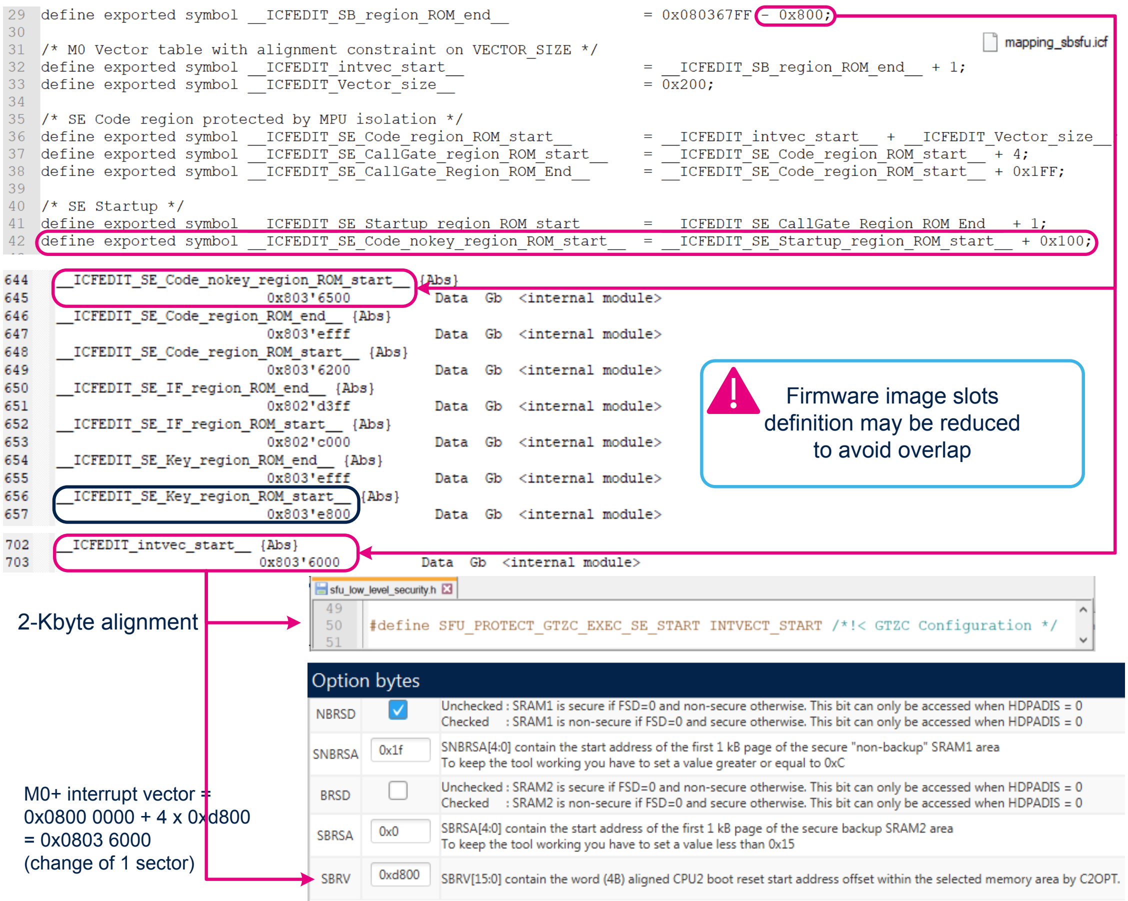
1. Identify the gap by analyzing the linker message (0x1F0 bytes).
2. Identify the concerned region by consulting the `project.map` file: `__ICFEDIT_SE_Code_nokey_region_ROM_start__`
3. Apply the modification in the `mapping_sbsfu.icf` file (0x800 bytes).
4. Check that the constraints related to this specific region are respected:
   – The lower regions (SE keys and active image headers cannot be reduced or moved). So the end address is fixed.
   – The sizes of the “SE call gate” and “SE startup” regions are default ones and must not be changed.
   – The first movable frontier (related to `__ICFEDIT_intvec_start__`) is also used as TZSC start address and must be aligned on 2 Kbytes.

Figure 28. Memory mapping adaptations



The impact of the memory mapping adaptation on security peripheral configuration must be checked even though it is automatically computed. For example, check the SBRV configuration using STM32CubeProgrammer (STM32CubeProg) as shown in the figure below.

Figure 29. **Check the protections**



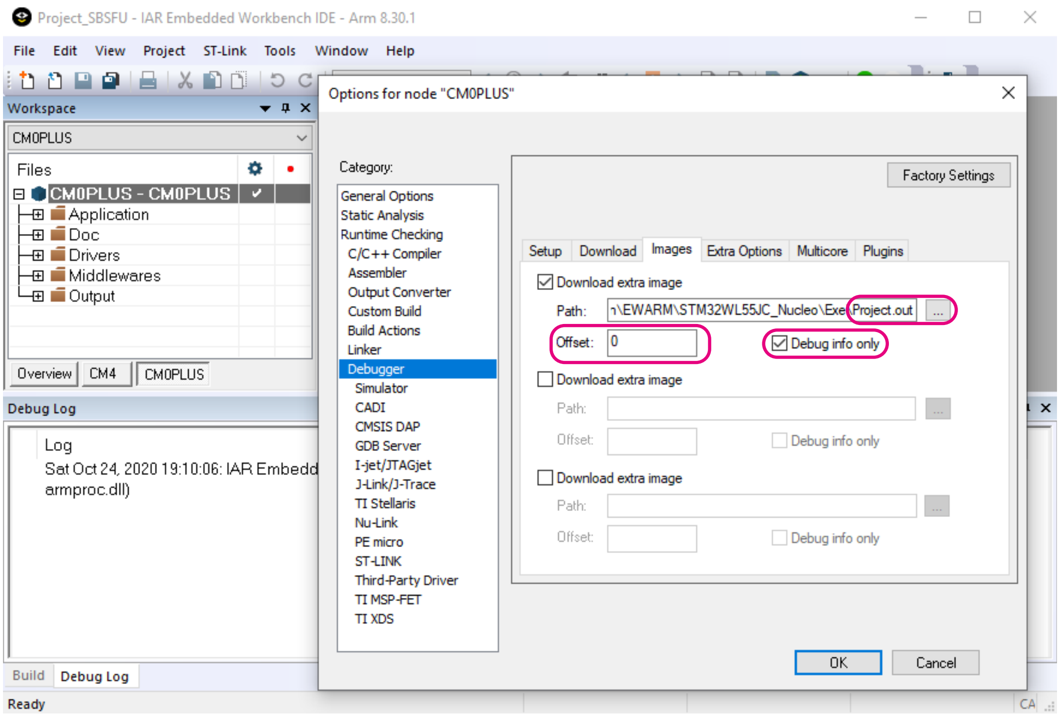
Depending on the modified regions, other configurations, like WRP or secure Flash, may have to be checked using STM32CubeProgrammer (STM32CubeProg).

5.3 Debug inside SECoreBin

To debug inside SECoreBin, the SBSFU project options must be changed to load SECoreBin symbols. This is performed in the *Debugger* menu as presented in the figure below:

- Browse to select the `Project.out` file.
- Set *Offset* to 0.
- Check the *Debug info only* box.

Figure 30. Debug inside SECoreBin



5.4 Disable the watchdog while debugging

In the dual-core configuration, when debugging a use case with an enabled watchdog, it is mandatory to freeze it as soon as the system enters debug mode (to perform step-by-step execution for instance). Otherwise, debug operations (like step-by-step execution) cannot be performed for more than a few seconds, because the watchdog timer expires and the board resets. With the change described below, the watchdog freezes when CPU1 or CPU2 is in debug mode.

The code lines in **bold** below must be added in *Firmware/Projects/NUCLEO-WL55JC/Applications/SBSFU_2_Slots_DualCore/2_Images_SBSFU/CM4/Src/main.c* .

```
/* Configure the security features */
if (SFU_BOOT_CheckApplySecurityProtections(SFU_INITIAL_CONFIGURATION) != SFU_SUCCESS)
{
  SFU_EXCPT_Security_Error();
}
FLOW_CONTROL_CHECK(uFlowProtectValue, FLOW_CTRL_RUNTIME_PROTECT);

/* Disable IWDG while debugging */
SET_BIT(DBGMCU->APB1FZR1, DBGMCU_APB1FZR1_DBG_IWDG_STOP);
SET_BIT(DBGMCU->C2APB1FZR1, DBGMCU_C2APB1FZR1_DBG_IWDG_STOP);

/* Boot CPU2 */
HAL_PWREx_ReleaseCore(PWR_CORE_CPU2);
```

5.5 Debug the Cortex-M0+ with HDP enabled

When the HDP is active, the Cortex-M0+ core is not accessible in debug mode, until the following changes are deployed:

- Disable the protections that forbid the debug (see the code below).
- Authorize explicitly the Cortex-M0+ debug (automatically disabled when HDP is active).

The code below shows the changes needed and where to apply them (location indications in italic, code to add in bold and code to delete in strikethrough):

- In `Firmware/Projects/NUCLEO-WL55JC/Applications/SBSFU_2_Slots_DualCore/2_Images_SBSFU/CM0PLUS/Core/Src/main.c`

```
@@ -41,6 +41,9 @@ int main(void)
   /* Reset of all peripherals, Initializes the Flash interface and the Systick*/
   (void) HAL_Init();

+  /* Enable C2 Debug */
+  HAL_FLASHEx_EnableC2Debug();
+
   /* Board BSP  Configuration-------------------------------------------------*/
   /*
    * As the secure mode has not been entered yet, we do not configure BSP right now .
```

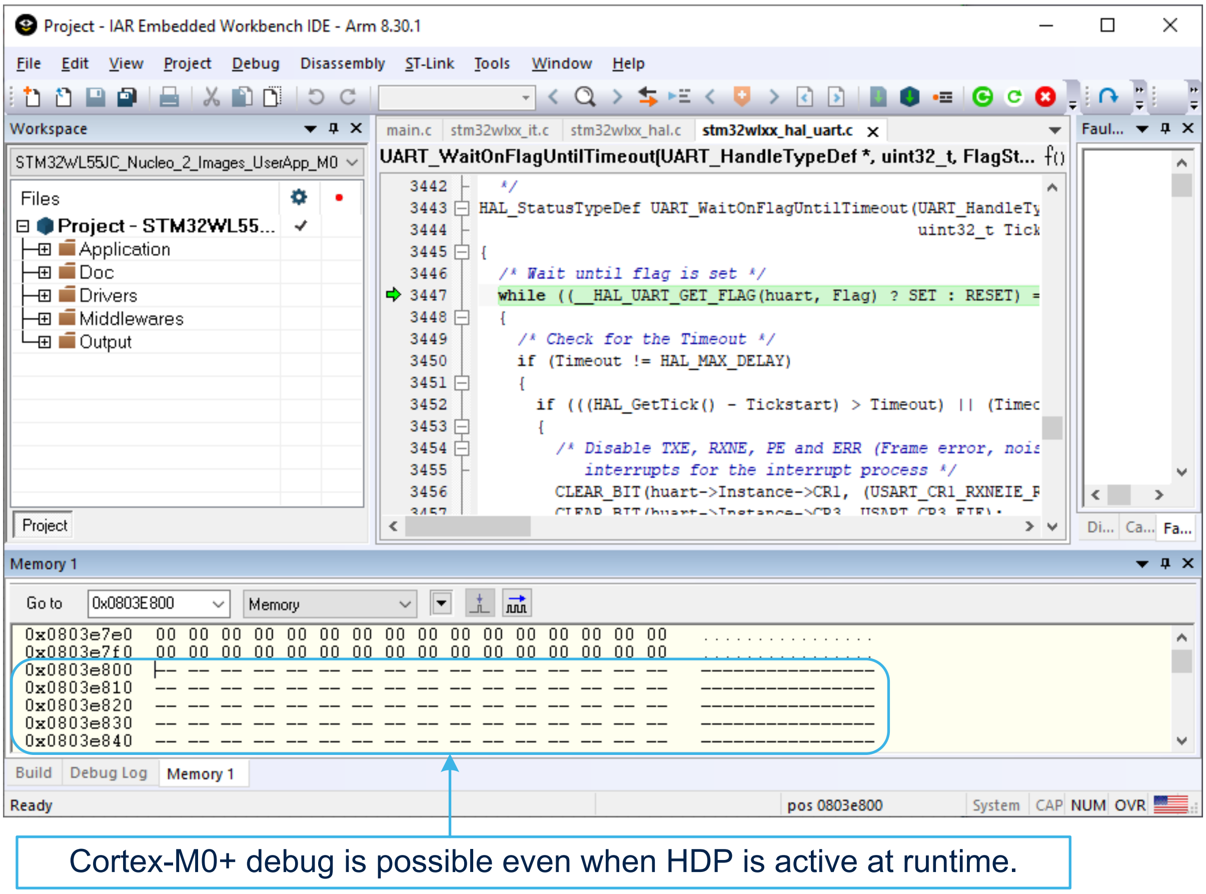
- In `Firmware/Projects/NUCLEO-WL55JC/Applications/SBSFU_2_Slots_DualCore/2_Images_SBSFU/CM0PLUS/SBSFU/App/app_sfu.h`

```
@@ -125,7 +125,7 @@ extern "C" {
    In debug mode it can be better to disable some of the following protection
    for a better Debug experience (WRP, RDP, IWDG, DAP, etc.) */

-#define SFU_RDP_PROTECT_ENABLE
+/*#define SFU_RDP_PROTECT_ENABLE*/
 /*#define SFU_TAMPER_PROTECT_ENABLE */  /*!< WARNING : Tamper protection deactivated.
As the tamper tamper pin is
                                              neither connected to GND nor to 5V
(floating level), there are too many
                                              spurious tamper event detected */
@@ -140,7 +140,7 @@ extern "C" {
 #if defined(SFU_SECURE_USER_PROTECT_ENABLE)
 #define SFU_GTZC_PROTECT_ENABLE     /*!< GTZC protection (dependent on
SFU_SECURE_USER_PROTECT_ENABLE):
                                         Enables/Disables the GTZC protection. */
-#define SFU_C2SWDBG_PROTECT_ENABLE /*!< Dynamic disabling of the CPU2 debug:
+/*#define SFU_C2SWDBG_PROTECT_ENABLE*/ /*!< Dynamic disabling of the CPU2 debug:
                                         not writable if ESE=0, no meaning if DDS=1. */
 #endif /* SFU_SECURE_USER_PROTECT_ENABLE */
```

- In `Firmware/Projects/NUCLEO-WL55JC/Applications/SBSFU_2_Slots_DualCore/2_Images_SBSFU/Common/app_sfu_common.h`

```
@@ -75,15 +75,15 @@ extern "C" {
    for a better Debug experience (WRP, RDP, IWDG, DAP, etc.) */

 #define SFU_WRP_PROTECT_ENABLE
-#define SFU_DAP_PROTECT_ENABLE /*!< WARNING: Be Careful if enabling this protection.
Debugger will be disconnected.
+/*#define SFU_DAP_PROTECT_ENABLE*/ /*!< WARNING: Be Careful if enabling this
protection. Debugger will be disconnected.
                                             It might be difficult to reconnect the
Debugger.*/
 #define SFU_DMA_PROTECT_ENABLE
-#define SFU_IWDG_PROTECT_ENABLE     /*!< WARNING:
+/*#define SFU_IWDG_PROTECT_ENABLE*/     /*!< WARNING:
      1. Be Careful if enabling this protection. IWDG will be active also after
         switching to UserApp: a refresh is needed.
      2. The IWDG reload in the SB_SFU code will have to be tuned depending on your
         platform (flash size...)*/
-#define SFU_C2_DDS_PROTECT_ENABLE       /*!< Static disabling of the CPU2 debug */
+/*#define SFU_C2_DDS_PROTECT_ENABLE*/      /*!< Static disabling of the CPU2 debug */
 #define SFU_SECURE_USER_PROTECT_ENABLE /*!< Only accessible in Secure access mode,
         the Secure user software is stored in the secure user memory, a configurable
         protected area which is part of the user main memory. */
```

Figure 31. Debug Cortex-M0+ with HDP enabled



5.6 Debug a Cortex-M0+ user application without SBSFU

To debug a Cortex-M0+ user application more easily, it is possible to run it in standalone mode. However, the Cortex-M4 Secure Boot must be kept as it powers on the Cortex-M0+ core.

The following steps must be followed:

1. Enable `SECBOOT_DISABLE_SECURITY_IPS`: without SBSFU, it is better to disable the protections.
2. Update the constant used by the Cortex-M4 Secure Boot to configure the option byte SBRV. With that change, the Cortex-M0+ boots at the user application address instead of trying to execute the SBSFU code.
3. Keep the Cortex-M0+ user application in privileged mode, to avoid right issues when resetting the debugged application with the debugger.
4. Load the Cortex-M4 Secure Boot and the Cortex-M0+ user application with the STM32CubeProgrammer for example. The application starts on its own as soon as the Cortex-M0+ core is booted.

Note: Do not forget to uncheck C2BOOT_LOCK, otherwise it is impossible to update SBRV.

The figure below shows the changes needed and where to apply them (file and path).

Figure 32. Debug Cortex- M0+ UserApp as a standalone application

Firmware/Projects/NUCLEO-WL55JC/Applications/SBSFU_2_Slots_DualCore/2_Images_SBSFU/Common/app_sfu_common.h

```
@@ -66,7 +66,7 @@ extern "C" {
  *
  */
 -/*#define SECBOOT_DISABLE_SECURITY_IPS*/  /*!< Disable all security IPs at once when activated */
+#define SECBOOT_DISABLE_SECURITY_IPS  /*!< Disable all security IPs at once when activated */
  #if !defined(SECBOOT_DISABLE_SECURITY_IPS)
```

Firmware/Projects/NUCLEO-WL55JC/Applications/SBSFU_2_Slots_DualCore/2_Images_SBSFU/Common/sfu_def.h

```
@@ -54,7 +54,8 @@ typedef enum
 #ifdef CORE_CM0PLUS
 #define SFU_BOOT_BASE_ADDR       ((uint32_t) INTVECT_START)           /* SFU Boot Address */
 #else /* CORE_CM0PLUS */
-#define SFU_C2_BOOT_BASE_ADDR    ((uint32_t) INTVECT_START)           /* SFU Boot Address */
+/* __ICFEDIT_SLOT_Active_1_start__ + 512 = 0x0801C000 + 0x200 */
+#define SFU_C2_BOOT_BASE_ADDR    ((uint32_t) 0x0801C200)              /* SFU Boot Address */
 #define SFU_C2_AREA_ADDR_END     ((uint32_t) SE_KEY_REGION_ROM_END)   /* SBSFU end Address (covering
all the SBSFU
                                                                         executable code and the
related keys) */
 #define SFU_C1_BOOT_BASE_ADDR    ((uint32_t) M4_SB_REGION_ROM_START)  /* SB Boot Address */
```

Firmware/Projects/NUCLEO-WL55JC/Applications/SBSFU_2_Slots_DualCore/2_Images_UserApp_M0Plus/Src/main.c

```
@@ -121,7 +121,7 @@ int main(void)
   printf("\r\n=====================================================================");
   printf("\r\n\r\n");

-  MPU_EnterUnprivilegedMode();
+  /* MPU_EnterUnprivilegedMode(); */

   /* User App firmware runs*/
   FW_APP_Run();
```

M0+ interrupt vector =
0x0800 0000 + 4 x 0x7080 =
0x0801 C200

SBRV	0x7080	SBRV[15:0] contain the word (4B) aligned CPU2 boot reset start address offset within the selected memory area by C2OPT.

6 Adapt the SBSFU

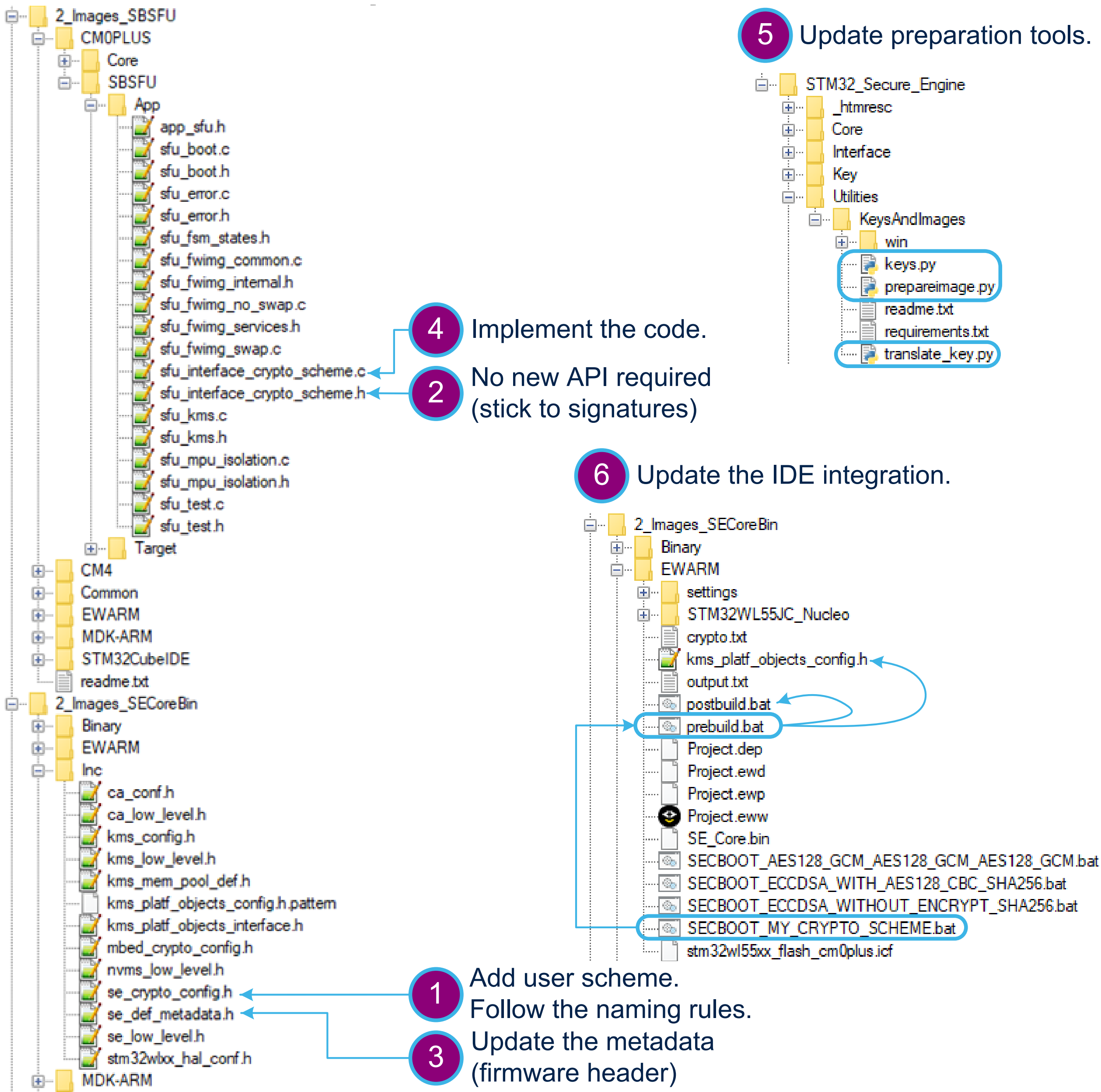
6.1 Implement a new cryptographic scheme for the SBSFU

The STM32CubeWL SBSFU comes with some predefined cryptographic schemes (refer to Section 3.2). The package can be extended with the user's cryptographic scheme.

To implement a new cryptographic scheme for the SBSFU, the following steps are needed (see the figure below):

1. Update the code running on the device side:
    a. **Step 1**: Define a new value for `SECBOOT_CRYPTO_SCHEME`.
    b. **Step 2**: Look carefully at the signatures of the APIs that the bootloader requires. The cryptographic services must have the same signatures to avoid updating the SBSFU code.
    c. **Step 3**: Define a new `SE_FwRawHeaderTypeDef` structure and respect the constraints to remain compatible with the existing SBSFU code.
    d. **Step 4**: Implement the code of the cryptographic services in the `sfu_interface_crypto_scheme.c` file.
2. Update the tools running on the host side to prepare the keys and the firmware image:
    a. **Step 5**: Update the preparation tools to support the new cryptographic scheme (`prepareimage.py`, `translate_key.py`, and `keys.py`).
    b. **Step 6**: Update the IDE integration to generate the appropriate keys and firmware image.
        - A new batch file is required to call the preparation tools with the appropriate commands. `prebuild.bat` copies this batch file to create `postbuild.bat`.
        - `prebuild.bat` must be updated to take into account the new cryptographic scheme and generate the proper keys and `postbuild.bat`.

Figure 33. User cryptographic scheme implementation



6.2 Optimize the memory mapping

Several options exist to reduce the SBSFU code size and maximize the size of the user application slot. Some of these options are summarized in the table below.

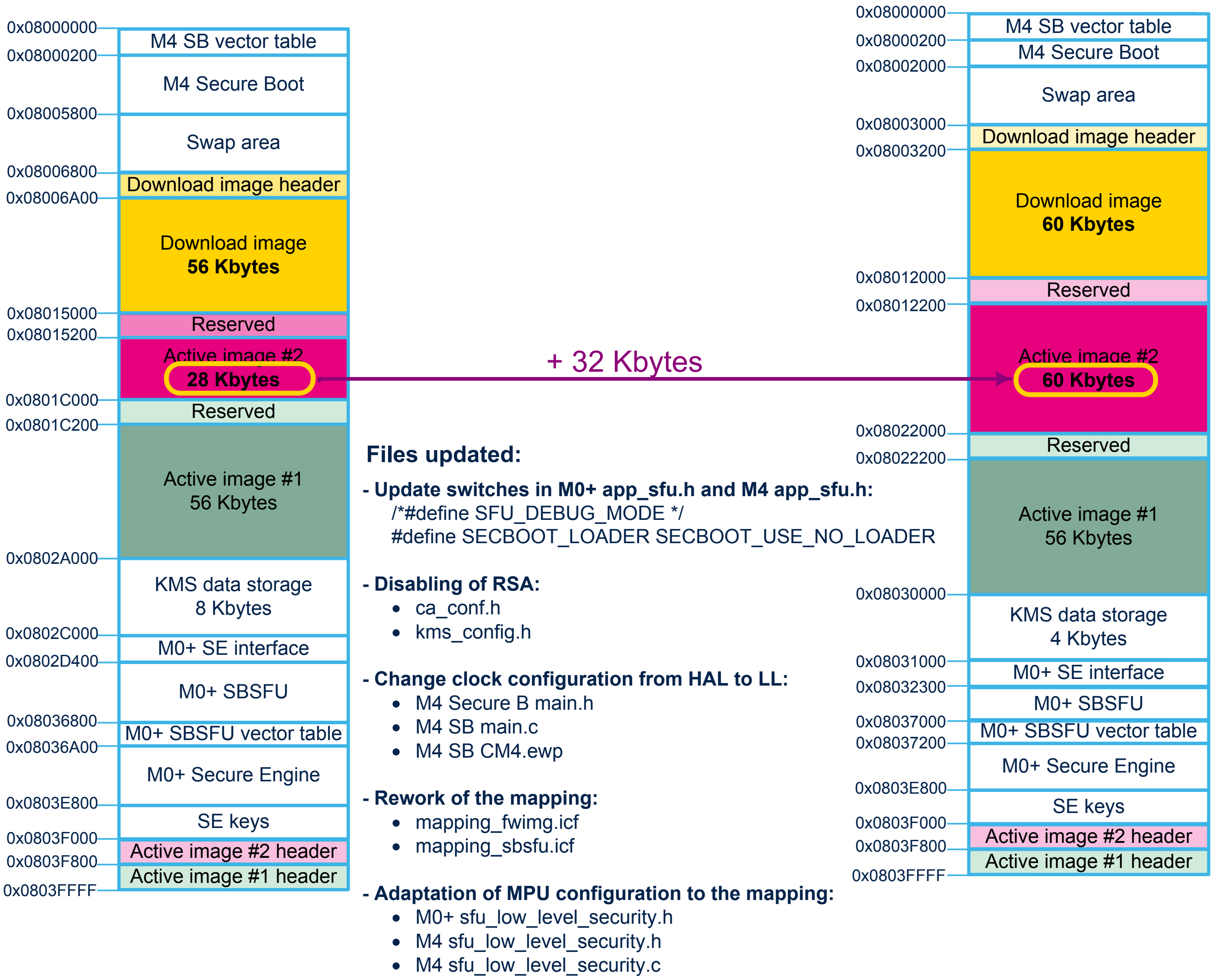
Table 2. SBSFU code size reduction

Option	Description/consequence	Gain
Select 1-slot variant.	Download a new firmware image from the user application is no more possible.	Slot size is doubled versus 2-slot projects
Disable the RSA feature.	This only removes the ability to handle RSA keys.	~ 2 Kbytes
Select the AES-GCM symmetric cryptographic scheme.	Shared symmetric key secret stored in the device	Up to 6 Kbytes if the feature “import blob” is also disabled (no ECDSA, no RSA)
Disable `SFU_DEBUG_MODE`.	No more information displayed on the terminal during the SBSFU execution	Up to 22 Kbytes when there is no loader requiring to keep the UART driver
Disable `SECBOOT_USE_LOCAL_LOADER`.	No more local loader inside the SBSFU application (not compatible with 1-image variant)	Up to 13 Kbytes when there is no debug trace requiring to keep the UART driver
Implement a hardware decryption.	Selects STM32 devices integrating cryptographic hardware peripheral.	Already implemented in the STM32CubeWL SBSFU
If all the code running on STM32 is fully trusted and robust, then Secure Engine internal isolation based on MPU/GTZC can be removed.	Removes alignment constraints with MPU/GTZC regions.	Up to 12 Kbytes
Reduce KMS data storage.	Reduces the number of keys stored in the KMS NVM.	~4 Kbytes
Configure the system clock with LL interface.	The code is a bit more complex and TAMPER must not be used as the removed HAL dependencies are restored.	~ 2 Kbytes

The total gain depends on the mapping constraints described in Section 2.2 .

The example detailed in the figure below, highlights the mapping modifications. Starting from two images with an asymmetric cryptographic scheme, the `SFU_DEBUG_MODE` and `SECBOOT_USE_LOCAL_LOADER` switches are disabled (`SECBOOT_USE_LOCAL_LOADER` = `SECBOOT_USE_NO_LOADER` in `app_sfu.h` of each core) . The Cortex-M4 clock configuration is done using LL APIs. The RSA feature is disabled. These changes results in a 32-Kbyte increase of the user application size. This configuration can be found in the LoRaWAN_FUOTA application.

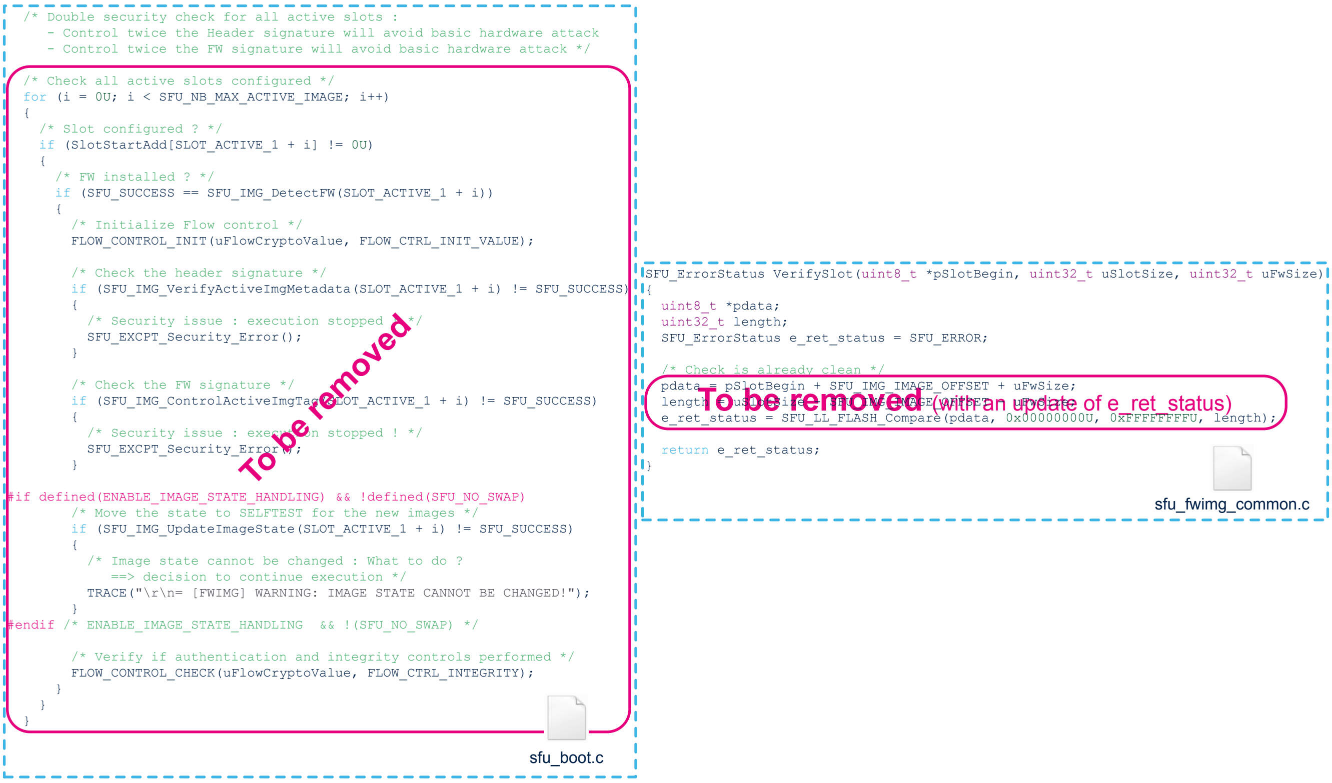
Figure 34. Example of memory mapping optimization on two images



6.3 How to improve the boot time

To resist to a basic fault injection attack, some critical actions are duplicated, thus impacting the time to start the user application. If such protections are not needed (for example, if there is no physical access to the device), these counter-measures can be removed as shown in the figure below.

Figure 35. Boot time



7 Adapt the user application

7.1 How to make an application SBSFU compatible

First of all, the mapping of the Cortex-M0+ user application must be modified to allow the application to run in the active slot #1:

- The code section starting by the vector table must be configured to run from the active slot #1, just after the reserved section (size = 512): `__ICFEDIT_SLOT_Active_1_start__` + 512 (`SFU_IMG_OFFSET` = 512)
- The data section must start at the beginning of the secure SRAM2: (`__ICFEDIT_SB_region_RAM_start__`: see `mapping_sbsfu.icf`)
- The data section must end before the Secure Engine protected area: (`__ICFEDIT_SB_region_RAM_end__`: see `mapping_sbsfu.icf`)

The Cortex-M4 user application must be modified to allow the application to run in the active slot #2:

- The code section starting by the vector table must be configured to run from active slot #2, just after the reserved section (size = 512): `__ICFEDIT_SLOT_Active_2_start__` + 512 (`SFU_IMG_OFFSET` = 512)
- The data section must start after the Cortex-M0+/Cortex-M4 synchronization flag: (`__ICFEDIT_M4_APP_region_RAM_start__`: see `mapping_sbsfu.icf`)
- The data section must end before the Cortex-M0+ secure data section: (`__ICFEDIT_M4_APP_region_RAM_end__`: see `mapping_sbsfu.icf`)

Refer to Section 2.2 for more details on memory constraints.

Then, during the system initialization, VTOR must be set to the new location of the vector table as shown in the figures below.

Figure 36. Cortex-M0+ vector table position update

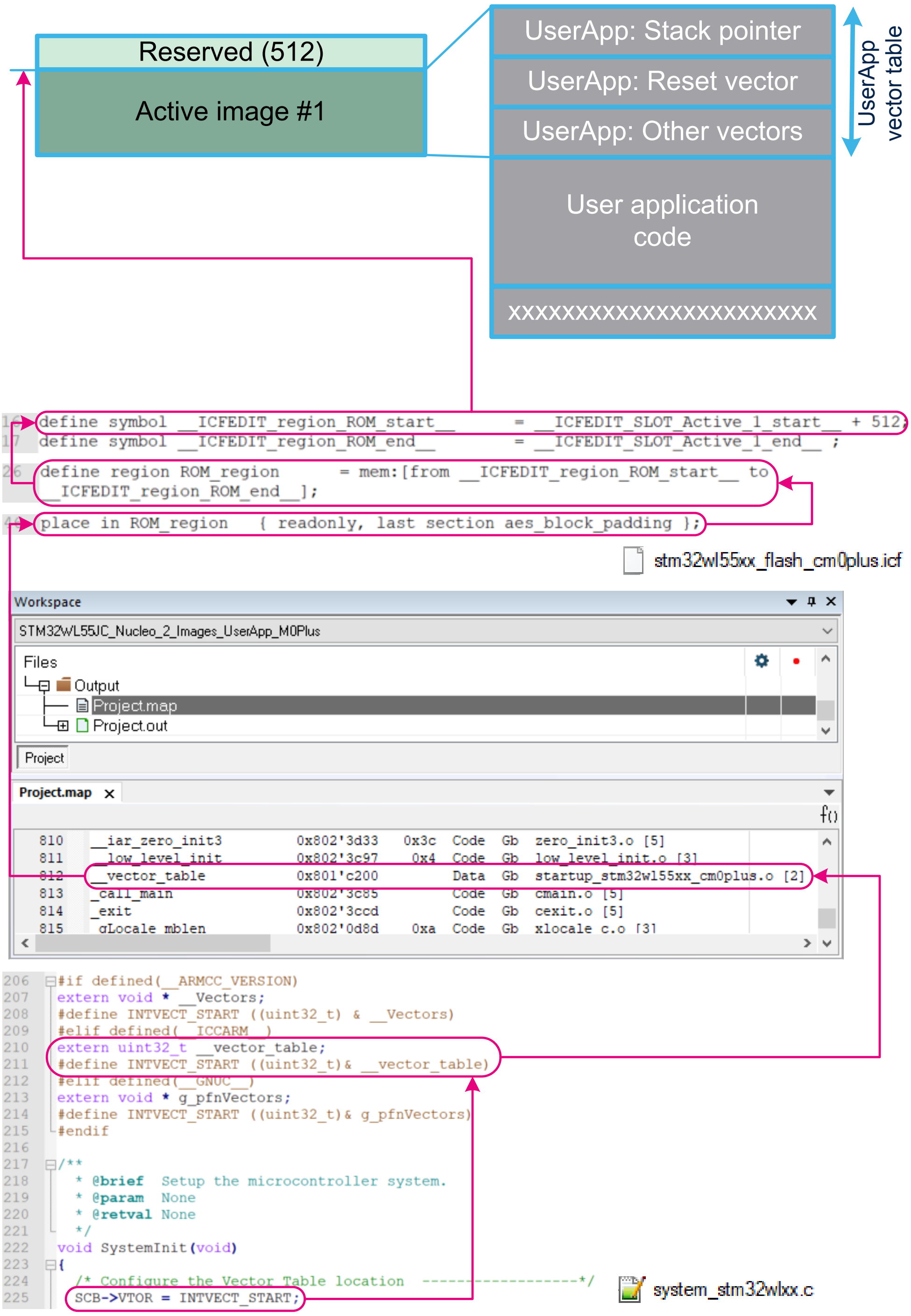
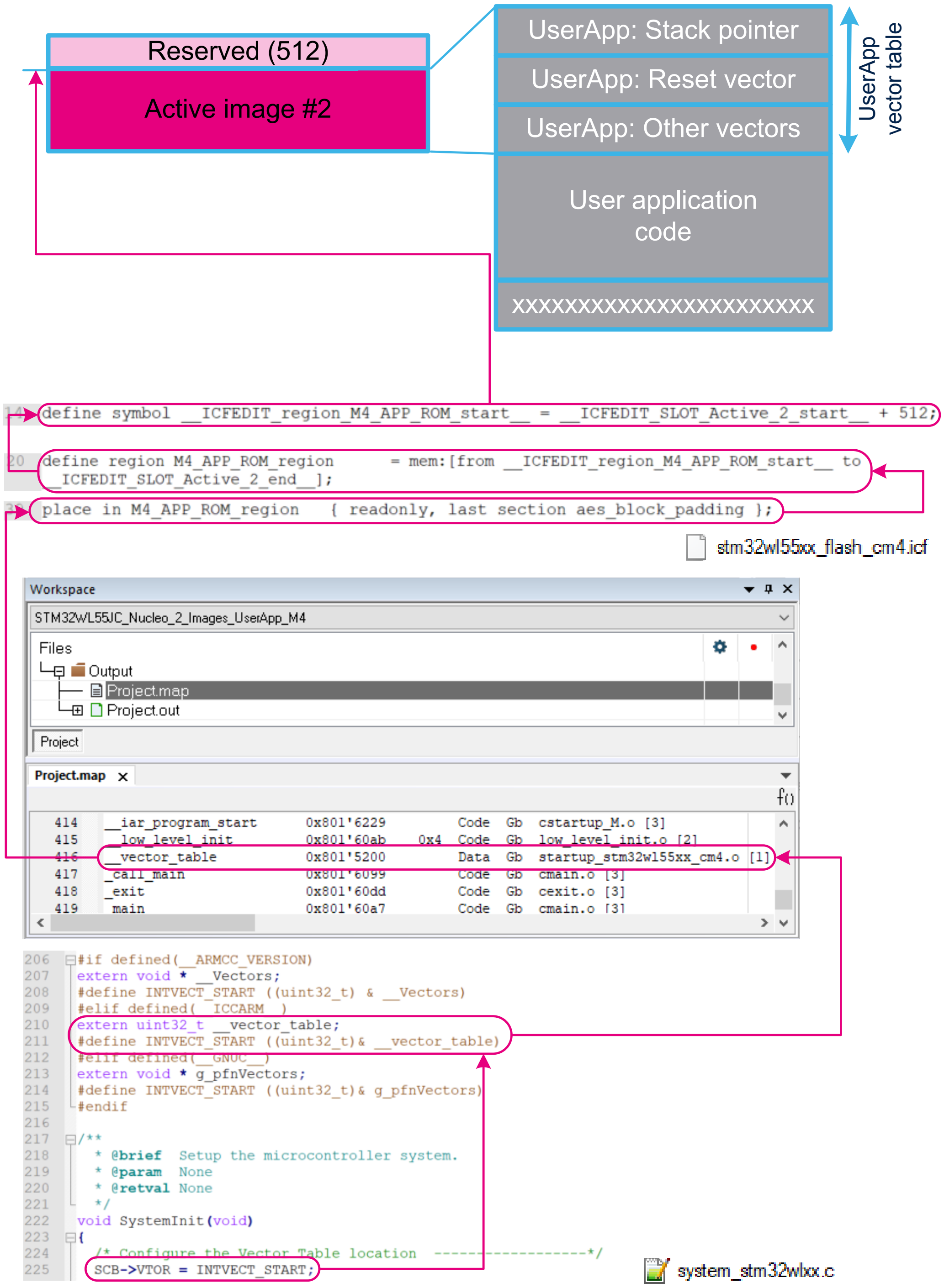


Figure 37. Cortex-M4 vector table position update



For user application encryption, the user application binary file length must be a multiple of 16 bytes. The figure below shows how to update the linker file to verify this constraint.

Figure 38. User application binary file length

stm32wl55xx_flash_cm0plus.icf

```
/* to make sure the binary size is a multiple of the AES block size (16 bytes) and WL flash writing unit (8 bytes) */
define root section aes_block_padding with alignment=16
{
udata8 "Force Alignment";
pad_to 16;
};


define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite };
do not initialize  { section .noinit };

place at address mem:__ICFEDIT_region_ROM_start__ { readonly section .intvec };

place in M4_region    { readonly section M4_UserApp_Bin };
place in ROM_region   { readonly last section aes_block_padding };
place in RAM_region   { readwrite, block CSTACK, block HEAP };
```

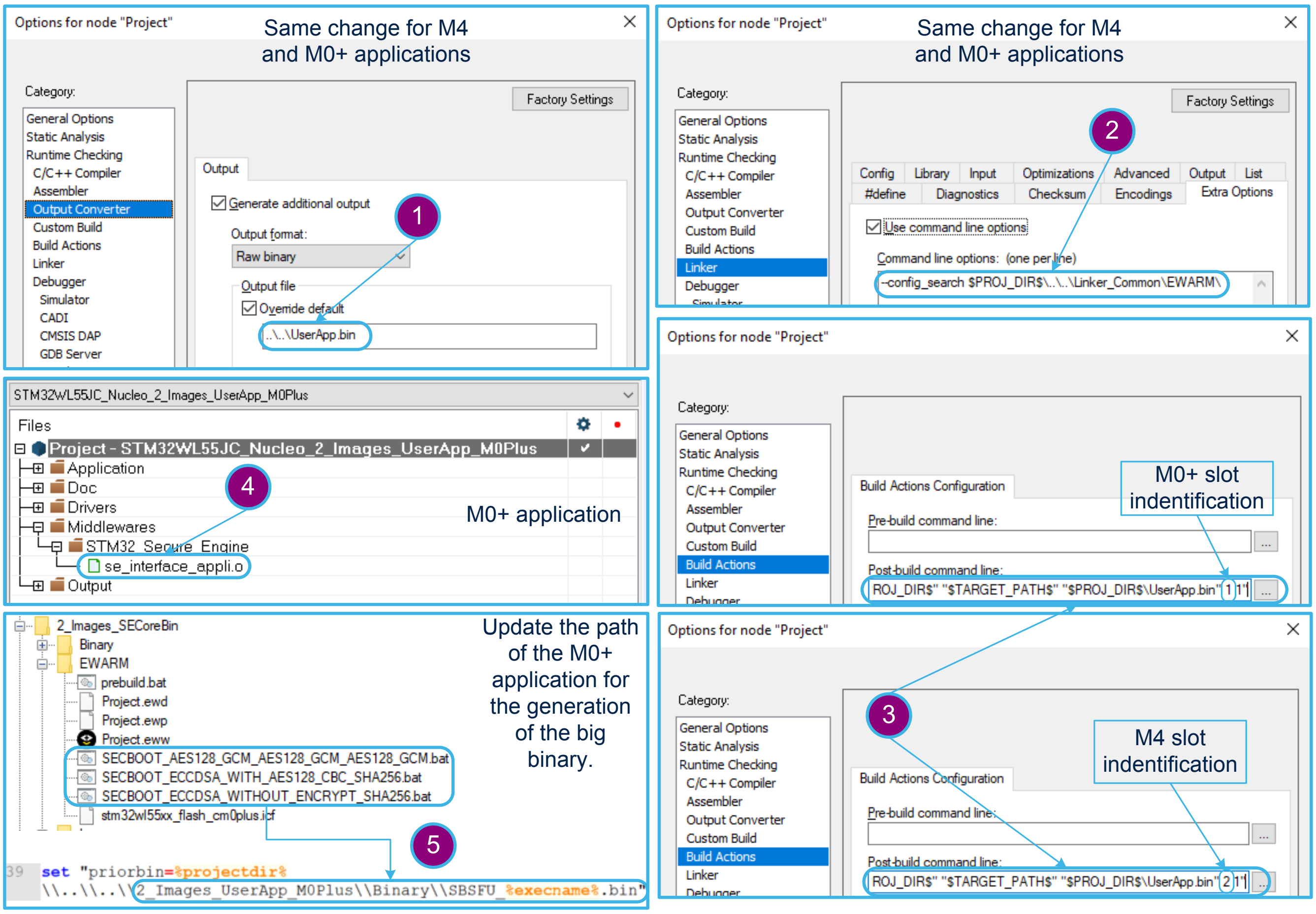
stm32wl55xx_flash_cm4.icf

```
/* to make sure the binary size is a multiple of the AES block size (16 bytes) and WL flash writing unit (8 bytes) */
define root section aes_block_padding with alignment=16
{
udata8 "Force Alignment";
pad_to 16;
};

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite };
do not initialize  { section .noinit };

/**************************************************************************/
/*                           placement instructions                       */
/**************************************************************************/
place at address mem:__ICFEDIT_region_M4_APP_ROM_start__  { readonly section .intvec };
place in M4_APP_ROM_region   { readonly last section aes_block_padding };
place in M4_APP_RAM_region   { readwrite, block CSTACK, block HEAP };
```

Finally, as done in the UserApp example, the IDE configurations must be updated with the following steps:

1. Generate a `UserApp.bin` file.
2. Include search path for linker common files.
3. Call `postbuild.bat` to generate `UserApp.sfb` and `SBFU_UserApp.bin` with the correct slot identification (1/2/3). Only the Cortex-M4 `SBFU_UserApp.bin` is complete.
4. Integrate `se_interface_appli.o` to access the Secure Engine runtime services if any.
5. Update the path to the Cortex-M0+ UserApp for the generation of the big binary (done during the final step, the Cortex-M4 UserApp compilation).

Figure 39. IDE configurations



There are some additional constraints:

- The GTZC/MPU-based Secure Engine isolation relies fully on the fact that a privileged level of software execution is required to access the Secure Engine services. The user application must take this constraint into account and trust any piece of code running in privileged mode.
- The IWDG is started during the SBSFU execution. It must be refreshed within a 6-second period.
- Double security checks must be implemented to ensure the security of the cryptographic operations called through KMS. For instance, signatures can be verified using the public key. The cryptographic operations can also be called twice and, if possible, their result can be compared. Any difference or unexpected result must be considered as a security error. The principle is illustrated in Section 6.3 How to improve the boot time.

Caution: A special care must be taken to review the Cortex-M0+ interrupt handlers. There must be no breach or bug possibility that may offer an access to secure memory (Flash or RAM), or make it possible to change/disable the MPU configuration. The debug features like the infinite loops in the interrupt handlers must also be removed.

7.2 Change the firmware download function in the user application

This possibility is available only in the dual-slot mode of operation.

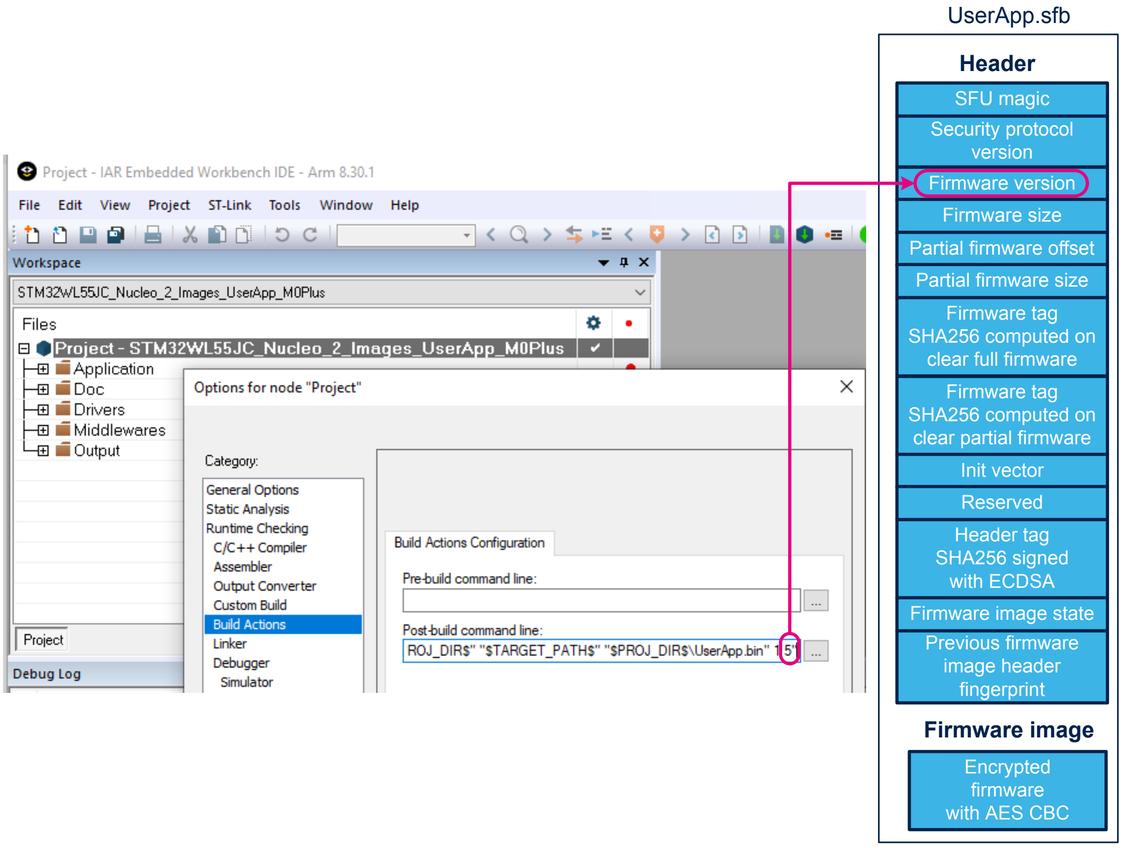
A sample code based on the Ymodem protocol over UART is available in the STM32CubeWL Cortex-M0+ UserApp project. The download procedure is located in the `fw_update_app.c` file, as illustrated in the figure below.

Figure 40. UserApp firmware download overview

7.3 How to change the firmware version

The firmware version is part of the firmware header generated with the `postbuild.bat` script. In the following example, the version is 5.

Figure 41. Firmware version change



Caution: The firmware with the SFU_FW_VERSION_INIT_NUM (`app_sfu.h`) version is the only one allowed for installation when the header of the installed image is not valid. This is the case either because no firmware is installed (development phase), or due to an attack attempt. It is important to keep such firmware private as the only purpose of this version is to analyze and repair devices returned from the field.

Revision history

Table 3. Document revision history

Date	Version	Changes
27-Nov-2020	1	Initial release.
6-Jul-2021	2	Updated: • Note in Introduction • Figure 1 to Figure 8 • Section 2.2.2 Parameters for firmware image slot definition • Figure 14. SECoreBin specific linker file • Section 2.2.4 Multiple-image configuration • Figure 20. SBSFU configuration • Note in Section 3.3 Security configuration • Notes in Section 3.4 Development or production mode configuration • Figure 25. New firmware encryption key • Figure 26. New private/public keys • Section 5.4 Disable the watchdog while debugging • File paths in Section 5.5 Debug the Cortex-M0+ with HDP enabled • Figure 32. Debug Cortex- M0+ UserApp as a standalone application • Table 2. SBSFU code size reduction • Figure 35. Boot time • End of Section 7.1 How to make an application SBSFU compatible Added: • Figure 4. Memory mapping example (BFU single-core dual-slot configuration) • Figure 5. Memory mapping example (SBSFU dual-core single-slot configuration)

Contents

List of tables

List of figures