How to secure LoRaWAN® and Sigfox™ with STM32CubeWL

Introduction

This application note describes how to secure LoRaWAN® or Sigfox™ applications embedded in the STM32CubeWL MCU Package for STM32WL Series MCUs (microcontrollers), by combining the project with the SBSFU (Secure Boot and Firmware Update) environment.

In several situations, the information shared between communicating parties needs to remain private, uncompromisable, and secure. In these sensitive conditions, maintaining a secure radio network is crucial.

In the first sections, this document gives an overview of the directory structure that combines the SBSFU to the RF dual-core projects, and guides trough the steps in order to compile, download, execute and debug the projects.

This application note describes the main code changes compared to the non-secure version, especially on privileged/ unprivileged mode. This document describes also how the RF application binaries use the SKMS in the SBSFU binary.

The last sections focus on memory mapping, memory footprint, explain how to customize the memory repartition between cores, and how to reduce the SBSFU memory footprint in order to gain Flash memory space for the application.

In order to learn about LoRaWAN, Sigfox and SBSFU projects, it is recommended to read the documents [1], [3] and [4].

1 General information

The STM32CubeWL runs on STM32WL Series microcontrollers based on the Arm® Cortex®-M processor.

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

arm

Table 1. Terms and acronyms

Acronym	Definition
ABP	Activation by personalization
APDU	Application protocol data unit
DAP	Debug access port
End device	Device used as sensor or actuator in a networked system
EE	EEPROM emulator
Firmware image	Binary image (executable) run by the end device as user application
Firmware header	Meta-data describing the firmware image to be installed
FUOTA	Firmware update over the air
GTZC	Global security controller
HAL	Hardware abstract layer
IDWG	Independent watchdog
HDP	Hide protection
KMS	Key management services
LoRa	Long-range radio technology
LoRaWAN	LoRa wide area network
Mbed-Crypto	Mbed cryptography library implementation of the cryptography interface of the Arm PSA (platform security architecture)
MBMUX	Mailbox multiplexer
MPU	Memory protection unit
MSC	Message sequence chart
RDP	Readout protection
RSA	Rivest Shamir Adleman
SBSFU	Secure Boot and Secure Firmware Update
`.sfb` file	Binary file packing the firmware header and the firmware image
SFU	Secure Firmware Update
SKMS	Secure key management services
SVC	Supervisor call
WRP	Write protection

Reference documents

[1] Application note *Integration guide of SBSFU on STM32CubeWL (including KMS)* (AN5544)

[2] User manual *Getting started with the SBSFU of STM32CubeWL* (UM2767)

[3] Application note *How to build a LoRa application with STM32CubeWL* (AN5406)

[4] Application note *How to build a Sigfox® application with STM32CubeWL* (AN5480)

[5] Application note *LoRaWAN firmware update over the air with STM32CubeWL* (AN5554)

[6] Application note *Introduction to STM32 microcontrollers security* (AN5156)

[7] STM32WL online training *STM32WL MBMUX mailbox multiplexer*

2 Secure project overview

The STM32WL55xx MCUs (also named STM32WL5x) embed two cores:

- Cortex-M0+, with high level of security features, used to store secrets (such as keys), and to run critical operations (such as cryptographic algorithms)
- Cortex-M4 for the user application

Note: With the GTZC on STM32WL5x devices, some registers and memories can be accessed only by the Cortex-M0+ and not by the Cortex-M4.

Between several possibilities on how to secure RF applications, it has been chosen to combine them with the SBSFU environment. The SBSFU brings a high-security level.

The STM32CubeWL provides examples of secure RF applications, by combining them with the SBSFU environment. The user can anyway decide to secure the RF application by handling directly the secure peripherals that are provided by STM32WL5x MCUs.

The SBSFU acronym suggests this module provides mainly Secure Boot and Secure Firmware Update, but the SBSFU provides more than that, as detailed below:

- The SBSFU provides a protected enclave managing all critical data and operations such as secure key storage, or cryptographic operations (SKMS).
- The SBSFU handles access to memories and peripherals via WRP, HDP, GTZC and MPU.
- The SBSFU provides protection from external attacks via RDP, anti-tamper and JTAG disconnection.
- The SBSFU can be used to control other protection feature (such as IWDG).

The STM32CubeWL provides two operation modes:

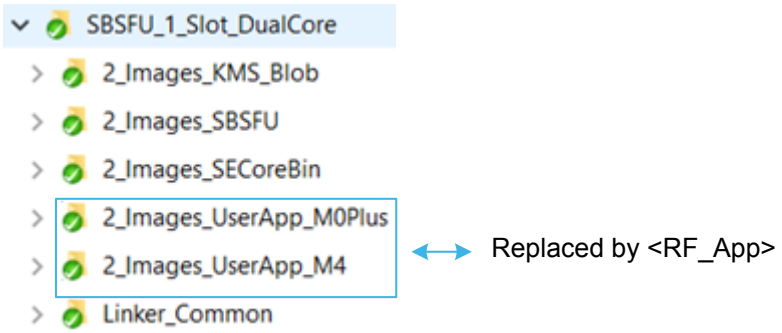
- dual-slot configuration (SBSFU_2_Slots): one active slot by core and one download slot, safe image programming enabled, with resume capability in the case of an installation procedure interruption
- single-slot configuration (SBSFU_1_Slot): one active slot by core, maximized user-application size

SBSFU_2_Slots is necessary for example to download a newer revision via the FUOTA method: the application itself must be running in order to download and assemble a new version of the firmware. But reserving 60-Kbyte memory just for firmware download reduces strongly the available memory for the application. When the application does not need to run during the download, SBSFU_1_Slot is recommended.

This document details how to secure LoRaWAN and Sigfox projects by combining them with a single-slot SBSFU (the firmware update via Y-MODEM does not need a running application).

STM32WL secure projects use both cores (Cortex-M4 and Cortex-M0+) to exploit the Cortex-M0+ security features. In the STM32CubeWL, specific examples are provided about the SBSFU (independently from RF applications). These examples combine the SBSFU with a small "SBSFU-test application" that does not have memory constraint.

Figure 1. SBSFU_1_Slot_DualCore structure



Some adaptations are required when replacing the 'SBSFU UserApp' project with RF stack applications, such as LoRaWAN or Sigfox. A special attention must be given to the Flash memory use.

Note: In this application note, the IAR Embedded Workbench® and Keil® MDK-ARM IDEs are used as example to provide guidelines for project configuration.

2.1 Directory structure

This application note presents a <Secure_RF_App> project description for the STM32WL5x devices, that can be either LoRaWAN_SBSFU_1_Slot_DualCore or Sigfox_SBSFU_1_Slot_DualCore.

<Secure_RF_App> is extended to LoRaWAN_FUOTA_DualCore project only for the information in Section 2 and Section 3 .

<Secure_RF_App> is split into the following subprojects:

- 2_Images_SECoreBin
- 2_Images_SBSFU (CM4 and CM0PLUS)
- <RF_App>
  - CM4: LoRaWAN or Sigfox application
  - CM0PLUS: LoRaWAN stack or Sigfox stack + wrapper
- 2_Images_KMS_Blob, integrated but not used by LoRaWAN_SBSFU_1_Slot_DualCore and Sigfox_SBSFU_1_Slot_DualCore

<RF_App> stands either for LoRaWAN_End_Node_DualCore or Sigfox_PushButton_DualCore.

The middleware is provided in source-code format and is compliant with the STM32WLxx_HAL_Driver .

Figure 2. Project file structure

2.2 SBSFU features and switches

2.2.1 Secure Boot (root-of-trust services)

The Secure Boot software permanently resides in the MCU read-only memory, and checks the integrity/authenticity of the installed user application. The Secure Boot executes every time a reset occurs, and checks if there is a new firmware update process to complete.

Main features of the Secure Boot are listed below:

- Checks and activates the STM32 security mechanisms to protect critical operations and secret data from an attack.
- Checks the integrity/authenticity of the user application code before every execution, to ensure that an invalid or malicious code cannot be run.

SBSFU instantiates the security item selected through `SECBOOT_DISABLE_SECURITY_IPS`. When this symbol is defined, security protections for all peripherals are disabled (such as WRP, RDP, IWDG, or DAP). See the document [6].

2.2.2 SFU (Secure Firmware Update)

The SFU provides a secure implementation of infield firmware updates, enabling secure download of a new firmware image.

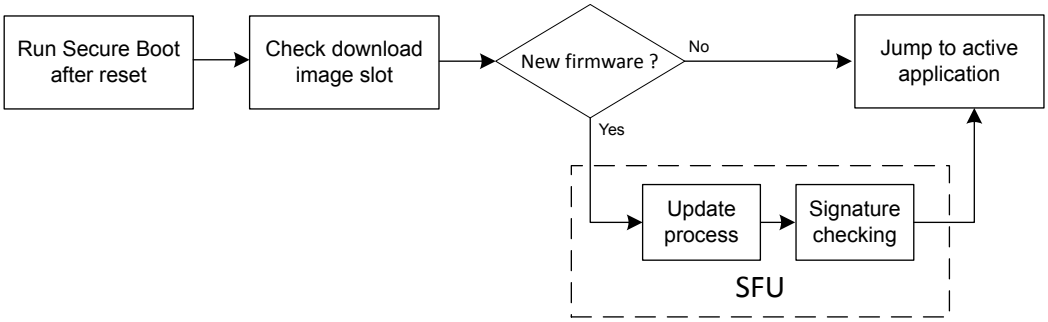
Two firmware update scenarios are possible:

- no new firmware to install

  There is no firmware update process to complete, and the Secure Boot does signature verifications and branches to the current active firmware.
- a new firmware to install

  The Secure Boot transfers control to the SFU that performs the firmware update and transfers control to the Secure Boot. The Secure Boot checks if there is a firmware update to complete, and branches to the new firmware.

Figure 3. Boot flow with SBSFU



There is a potential risk when one of the devices becomes compromised (as an attacker can initiate a multicast session with rogue firmware). To counter this, the SBSFU can add a layer of security by using an asymmetric cryptography scheme. When a firmware update is generated, the update is signed (TAG) with a private key by means of the 'Prepare Image' SBSFU tool (see document [2]). When the end device receives the firmware update, the end device verifies the signatures against the file received and the public key held in the 'secure core' part of the end device.

The main features of the SFU are listed below:

- Detects the requests to download and installs a new firmware (encrypted) version (using Y-MODEM application for firmware download).
- Manages the firmware version by checking for unauthorized update/installation.
- Decrypts the firmware (if encryption activated).
- Checks the firmware authentication and integrity.
- Installs the firmware.
- Executes the installed firmware (once authenticated and integrity checked).

Next tables list the compilation switches used for the SBSFU configuration.

Table 2. Security common switches

Location: <Secure_RF_App>\2_Images_SBSFU\Common\app_sfu_common.h

Symbols	Description	Default state
SECBOOT_DISABLE_SECURITY_IPS	Disables simultaneously all security peripherals when activated	Disabled
SFU_WRP_PROTECT_ENABLE	Write access protection to protect trusted code	Enabled
SFU_DAP_PROTECT_ENABLE	Debug access port protection	Enabled
SFU_DMA_PROTECT_ENABLE	DMA access protection	Enabled
SFU_IWDG_PROTECT_ENABLE	IDWG protection	Disabled
SFU_C2_DDS_PROTECT_ENABLE	Static Cortex-M0+ debug protection	Enabled
SFU_SECURE_USER_PROTECT_ENABLE	Secure user-memory protection	Enabled
SFU_FINAL_SECURE_LOCK_ENABLE	Secure production protection	Disabled
SFU_HIDE_PROTECTION_CFG	HDP area configuration	Enabled
OB_SECURE_SYSTEM_AND_FLASH	Flash memory and system secure area protection	Enabled
OB_SECURE_SRAM1	SRAM1 area protection	Disabled
OB_SECURE_SRAM2	SRAM2 area protection	Enabled

Table 3. Security Cortex-M0+ switches

Location: <Secure_RF_App>\2_Images_SBSFU\CM0PLUS\SBSFU\App\app_sfu.h

Symbols	Description	Default state
SFU_RDP_PROTECT_ENABLE	Readout protection	Enabled
SFU_TAMPER_PROTECT_ENABLE	Tamper protection (hardware pin)	Disabled
SFU_MPU_PROTECT_ENABLE	MPU protection on Cortex-M0+ regions	Enabled
SFU_MPU_USERAPP_ACTIVATION	User-application memory protection during execution	Enabled
SFU_GTZC_PROTECT_ENABLE	GTZC protection	Enabled
SFU_C2SWDBG_PROTECT_ENABLE	Dynamic Cortex-M0+ debug protection	Enabled

Table 4. Security Cortex-M4 switches

Location: <Secure_RF_App>\2_Images_SBSFU\CM4\Inc\app_sfu.h

Symbols	Description	Default state
SFU_MPU_PROTECT_ENABLE	MPU protection on Cortex-M4 regions	Enabled
SFU_MPU_USERAPP_ACTIVATION	User-application memory protection during execution	Enabled

2.2.3 SKMS (secure key management services)

In the dual-core configuration, KMS within SBSFU framework is executed inside a protected/isolated environment to ensure that any key value cannot be accessed by an unauthorized code running outside this protected/isolated environment (referred as Secure KMS).

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

The main SKMS features are listed below:

- Provides cryptographic services to the user application through the `PCKS #11` APIs, that are executed inside a protected/isolated environment. For more details, refer to the document [1].
  - Encrypt
  - Decrypt
  - Digest
  - Sign
  - Verify
  - Derive key
  - Search key
- Provides cryptographic services to the SFU to authenticate the user application with some protected keys.

The enabled algorithms are listed in the table below.

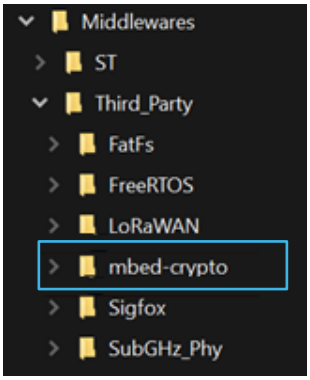
Table 5. SKMS features default configuration

Features	Configuration
AES CBC algorithm support	Encryption/decryption
AES CCM algorithm support	No
AES ECB algorithm support	Encryption/decryption and key derivation
AES CGM algorithm support	Encryption/decryption
AES CMAC algorithm support	Signature and verification
RSA algorithm support	No
RSA algorithm	Not activated
RSA 1024-bit modulus length	No
RSA 2048-bit modulus length	No
ECDSA algorithm support	Verification
ECDSA algorithm	Activated and associated to an elliptic curve
Elliptic curve SECP-192	No
Elliptic curve SECP-256	Yes
Elliptic curve SECP-384	No
SHA1 digest algorithm	
SHA256 digest algorithm	Digest

2.2.4 SBSFU cryptographic middleware

The SBSFU for STM32CubeWL supports the mbed-crypto (open-source code) cryptographic services for SHA256.

Figure 4. Cryptographic library structure



2.2.5 SBSFU cryptographic schemes

The SBSFU for STM32CubeWL is delivered with the following cryptographic schemes that use symmetric and asymmetric cryptographic operations:

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

Table 6. Cryptographic switches

Symbols	Description
SECBOOT_ECCDSA_WITHOUT_ENCRYPT_SHA256	No firmware encryption. Only authentication and integrity are ensured with asymmetric cryptography.
SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256	Authentication, integrity, and confidentiality are ensured with asymmetric cryptography.
SECBOOT_AES128_GCM_AES128_GCM_AES128_GCM	Authentication, integrity, and confidentiality are ensured with symmetric cryptography.

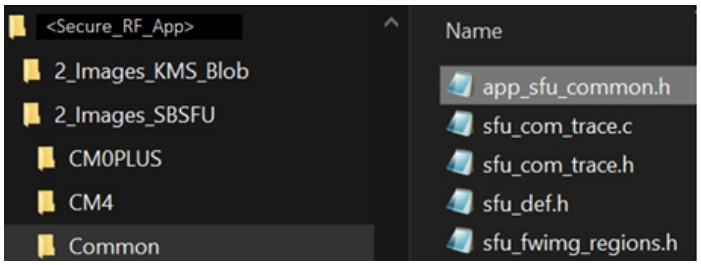
2.3 SBSFU configuration in RF applications

All secure peripherals are enabled by default except:

- IWDG as the user application does not contain the functionality required for its refresh
- TAMP (see Section 7.3.2)

2.3.1 Common SFU configuration

Figure 5. File structure of common security configuration



Common definitions apply to both cores. When `SECBOOT_DISABLE_SECURITY_IPS` is defined, most of security peripherals (such as WRP, DAP, IWDG, MPU) become disabled.

It can be useful to define `SECBOOT_DISABLE_SECURITY_IPS` during debug, and to comment it as follows to use security.

```
/*!< Disable all security IPs at once when activated */
/*#define SECBOOT_DISABLE_SECURITY_IPS*/
```

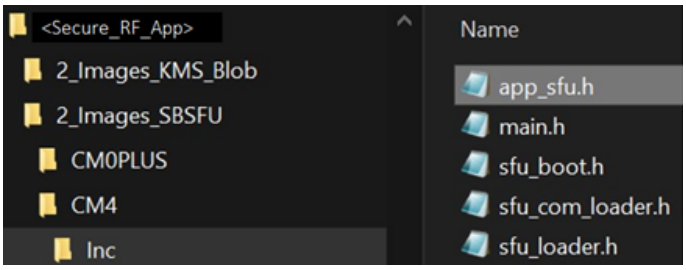
Each peripheral protection can be enabled separately by defining the corresponding definition.

By default (in Release mode), all security peripheral protections are enabled except for IWDG and TAMPER. In production, it is recommended to enable all peripheral protections (additional application code is needed if the IWDG protection is enabled).

```
#if !defined(SECBOOT_DISABLE_SECURITY_IPS)
#define SFU_WRP_PROTECT_ENABLE
#define SFU_DAP_PROTECT_ENABLE
#define SFU_DMA_PROTECT_ENABLE
/*#define SFU_IWDG_PROTECT_ENABLE*/
#define SFU_C2_DDS_PROTECT_ENABLE
#define SFU_SECURE_USER_PROTECT_ENABLE
#endif /* !SECBOOT_DISABLE_SECURITY_IPS */

#if defined(SFU_SECURE_USER_PROTECT_ENABLE)
#define SFU_HIDE_PROTECTION_CFG OB_SECURE_HIDE_PROTECTION_ENABLE
#define SFU_SECURE_MODE (OB_SECURE_SYSTEM_AND_FLASH_ENABLE | \
                         SFU_HIDE_PROTECTION_CFG | \
                         OB_SECURE_SRAM1_DISABLE | \
                         OB_SECURE_SRAM2_ENABLE)
#endif /* SFU_SECURE_USER_PROTECT_ENABLE */
```

2.3.2 Cortex-M4 SFU configuration

Figure 6. File structure of Cortex-M4 security configuration

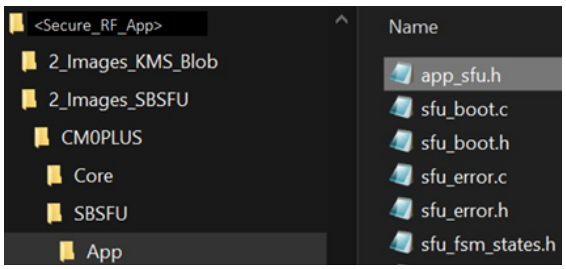


The following definitions apply only to Cortex-M4:

```
#if !defined(SECBOOT_DISABLE_SECURITY_IPS)
#define SFU_MPU_PROTECT_ENABLE
#define SFU_MPU_USERAPP_ACTIVATION
#endif /* !SECBOOT_DISABLE_SECURITY_IPS */
```

2.3.3 Cortex-M0+ SFU configuration

Figure 7. File structure of Cortex-M0+ security configuration

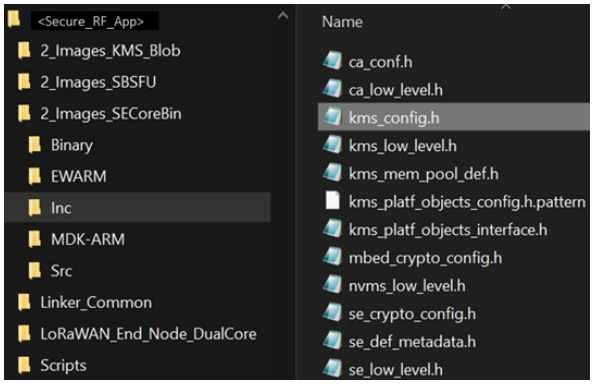


The following definitions apply only to Cortex-M0+:

```
#if !defined(SECBOOT_DISABLE_SECURITY_IPS)
#define SFU_RDP_PROTECT_ENABLE
/*#define SFU_TAMPER_PROTECT_ENABLE*/
#define SFU_MPU_PROTECT_ENABLE
#define SFU_MPU_USERAPP_ACTIVATION

#if defined(SFU_SECURE_USER_PROTECT_ENABLE)
#define SFU_GTZC_PROTECT_ENABLE
#define SFU_C2SWDBG_PROTECT_ENABLE
#endif /* SFU_SECURE_USER_PROTECT_ENABLE */
#endif /* !SECBOOT_DISABLE_SECURITY_IPS */
```

2.3.4 SKMS and cryptographic configuration

Figure 8. File structure of KMS and cryptographic definition

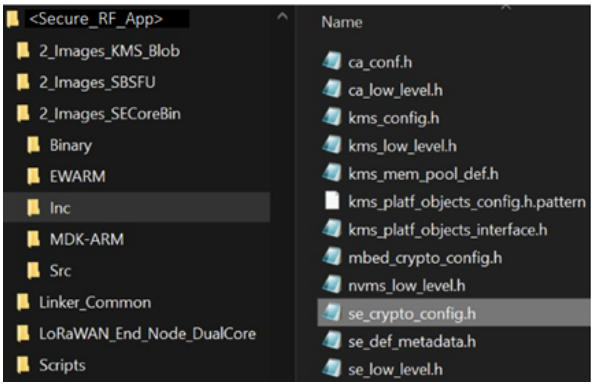


KMS definitions:

```
#define KMS_NVM_ENABLED
#define KMS_ENCRYPT
#define KMS_DECRYPT
#define KMS_DIGEST
#define KMS_SIGN
#define KMS_VERIFY
#define KMS_DERIVE_KEY
#define KMS_SEARCH
/*#define KMS_GENERATE_KEYS*/

#define KMS_AES_CBC         (KMS_FCT_ENCRYPT|KMS_FCT_DECRYPT)
/*#define KMS_AES_CCM       (KMS_FCT_ENCRYPT|KMS_FCT_DECRYPT)*/
#define KMS_AES_ECB         (KMS_FCT_ENCRYPT|KMS_FCT_DECRYPT|KMS_FCT_DERIVE_KEY)
#define KMS_AES_GCM         (KMS_FCT_ENCRYPT|KMS_FCT_DECRYPT)
#define KMS_AES_CMAC        (KMS_FCT_SIGN|KMS_FCT_VERIFY)
/* #define KMS_RSA          (KMS_FCT_SIGN|KMS_FCT_VERIFY) */
/* #define KMS_RSA_1024 */
/* #define KMS_RSA_2048 */
#define KMS_ECDSA           (KMS_FCT_VERIFY)
/*#define KMS_EC_SECP192*/
#define KMS_EC_SECP256
/*#define KMS_EC_SECP384*/
/*#define KMS_SHA1          (KMS_FCT_DIGEST)*/
#define KMS_SHA256          (KMS_FCT_DIGEST)
```

By default, the <Secure_RF_App> project is configured with asymmetric cryptography. The firmware authentication, integrity, and confidentiality (encryption) are ensured.

Figure 9. File structure of cryptographic scheme



Cryptographic scheme definitions:

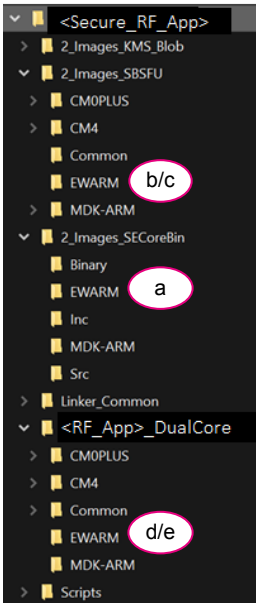
```
#define SECBOOT_CRYPTO_SCHEME    SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256

#define SECBOOT_ECCDSA_WITHOUT_ENCRYPT_SHA256        (1U)
#define SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256        (2U)
#define SECBOOT_AES128_GCM_AES128_GCM_AES128_GCM     (3U)
```

3 Firmware programming guide

This section describes how to generate a <Secure_RF_App> application, that refers, in this section, to LoRaWAN_SBSFU_1Slot_DualCore, Sigfox_SBSFU_1Slot_DualCore or LoRaWAN_FUOTA_DualCore project .

The developer must follow step-by-step this flow. A top-level view of the file structure is shown in the figure below.

Figure 10. Project order structure



Steps (detailed in next sections):

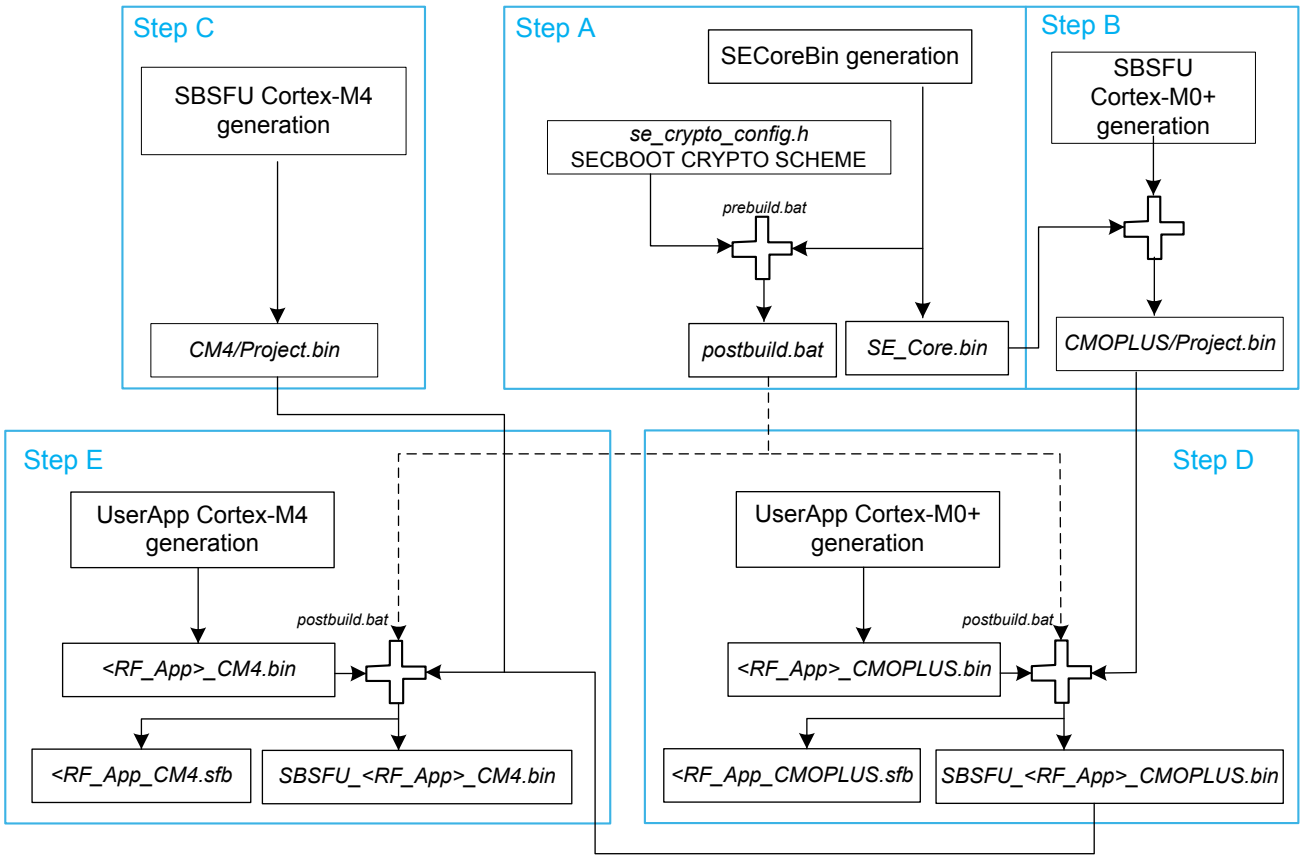
a: 2_Images_SECoreBin
b: 2_Images_SBSFU_CM0PLUS
c: 2_Images_SBSFU_CM4
d: <RF_APP>_DualCore_CM0PLUS
e: <RF_APP>_DualCore_CM4

Additional information on how configure <RF_App> are provided in the documents [3] and [4].

3.1 How to generate a <Secure_RF_App>

The steps details in the figure below must be followed to generate a RF_SBSFU_1_Slot_DualCore application. For each step, open the associated subproject in the dedicated IDE folder, and regenerate (make) the respective binary files.

Figure 11. Application generation steps



The following output binaries are generated in these steps (all of them in clear format, not encrypted):

- `SE_Core.bin (2_Images_SECoreBin)`
- `CM0PLUS/Project.bin (2_Images_SBSFU and includes SE_Core.bin)`
- `CM4/Project.bin (2_Images_SBSFU)`
- `<RF_App>_CM0PLUS.bin (<RF_App>)`
- `<RF_App>_CM4.bin (<RF_App>)`

In addition, the following output files are generated through the postbuild process:

- `<RF_App>_CM0PLUS.sfb` (`<RF_App>_CM0PLUS.bin` encrypted + header)
- `<RF_App>_CM4.sfb` (`<RF_App>_CM4.bin` encrypted + header)
- `SBSFU_<RF_App>_CM4.bin` (five first binary files merged with the memory placement to produce the final memory image)

Note: The document [2] explains how to configure a complete SBSFU for STM32WL project.

The various steps to follow are detailed below:

Step 1: 2_Images_SECoreBin

This step is needed to create the SECoreBin binary including all the required “trusted” code and keys. The binary is linked to the SBSFU_M0+ code in step 2. Static embedded keys of the SBSFU and RF application are stored in the SECoreBin.

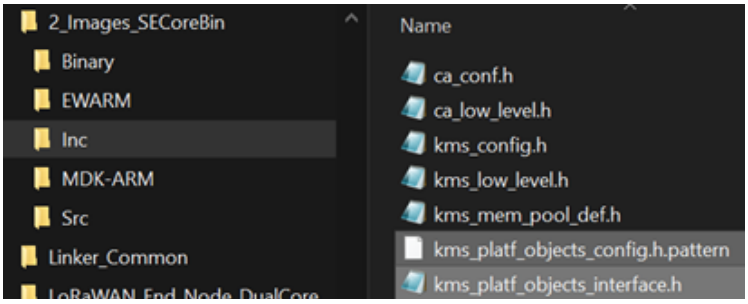
The RF static embedded keys are stored through the `kms_platf_objects_config.h.pattern` configuration file:

- For LoRaWAN project

  `kms_platf_objects_config.h` defines four static embedded keys via the inclusion the `Commissioning.h` header file from LoRaWAN_End_Node_DualCore project:
  - APP_KEY
  - NWK_KEY
  - NWK_S_KEY (used only in ABP)
  - APP_S_KEY (used only in ABP)
- For Sigfox project

  `kms_platf_objects_config.h` defines static embedded keys via the inclusion the `sigfox_data.h` header file from Sigfox_PushButton_DualCore project:
  - Sigfox_Data_Key
  - Sigfox_pac
  - Sigfox_id

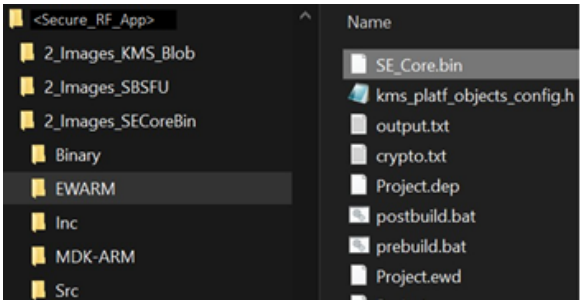
Figure 12. File structure of KMS user key configuration



Dynamic keys are detailed in Section 7.3 . For more details about the KMS configuration, see specific sections in the documents [3] and [4].

The generated `SE_Core.bin` output file is located in the IDE folder.

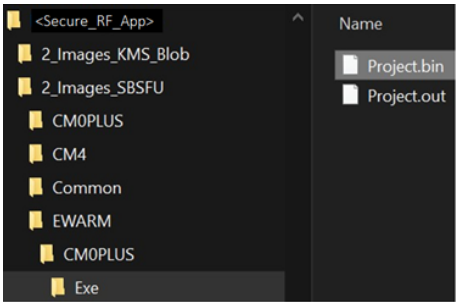
Figure 13. File structure of SECoreBin output



Step 2: 2_Images_SBSFU_CM0PLUS

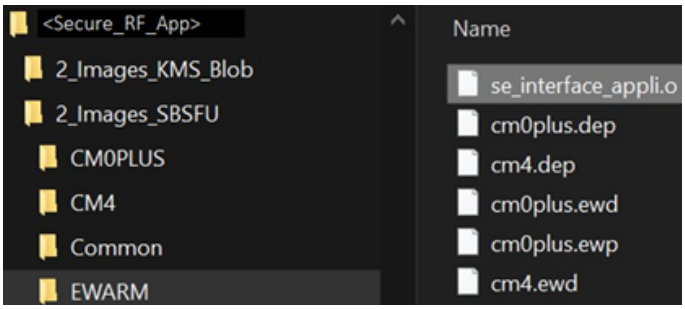
This step compiles the SBSFU Cortex-M0+ source code that implements the state machine and Cortex- M0+ protection configurations. This step links the code with the secure-engine bin, including the “trusted” code. The generated `Project.bin` output file is located to the IDE folder.

Figure 14. File structure of SBSFU Cortex-M0+ output (EWARM example)



This step also generates a file that includes symbols used by the user application to call the secure-engine interface public functions.

Figure 15. File structure of SE interface (EWARM example)



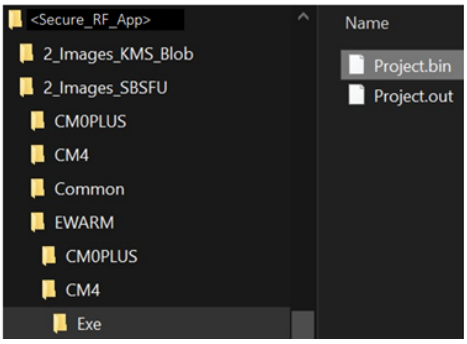
The `CM0PLUS Project.bin` contains `se_interface_appli.o` and `SE_Core.bin` files.

Step 3: 2_Images_SBSFU_CM4

This step compiles the SBSFU Cortex-M4 source code that implements the startup sequence to release Cortex-M0+ and Cortex-M4 protection configurations.

The generated `Project.bin` output file is located in the IDE folder.

Figure 16. File structure of SBSFU Cortex-M4 output (EWARM example)

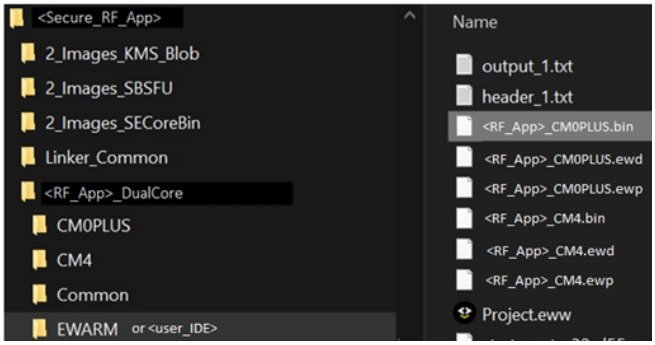


Step 4: <RF_App>_DualCore_CM0PLUS

This step compiles the <RF_App> Dual Core CM0PLUS source code including the correspondent middleware part. See documents [3] and [4] to know how to configure <RF_App>.

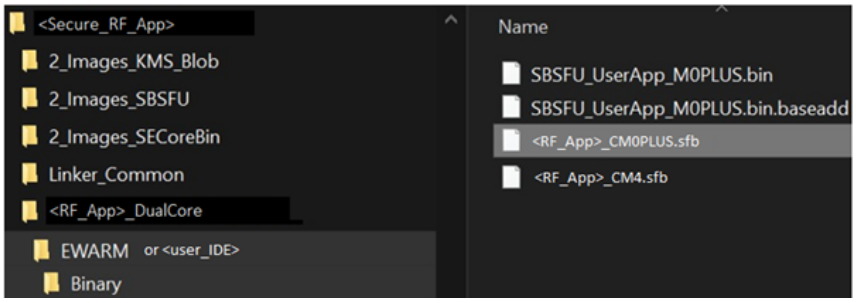
The generated `<RF_App>_CM0PLUS.bin` output file is located to the IDE folder.

Figure 17. File structure of <RF_App> Cortex-M0+ output



This step also generates the `<RF_App>_CM0PLUS.sfb`, UserApp CM0PLUS binary in encrypted format, including the SFU header.

Figure 18. File structure of <RF_App> Cortex-M0+ encrypted output

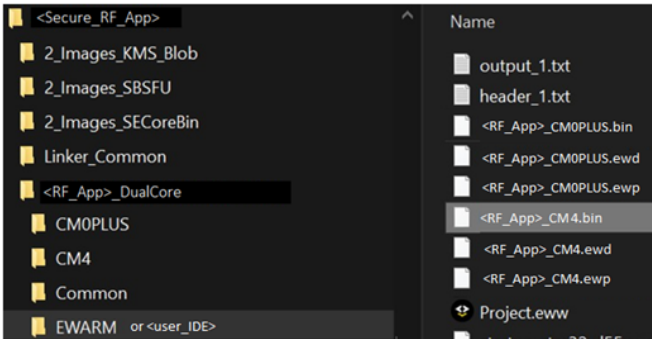


Step 5: <RF_App>_DualCore_CM4

This step compiles the <RF_App> Dual Core CM4 source code implementing the user application and sequence configuration. See documents [3] and [4] to know how to configure <RF_App>.

The generated `<RF_App>_CM4.bin` output file is located to the IDE folder.

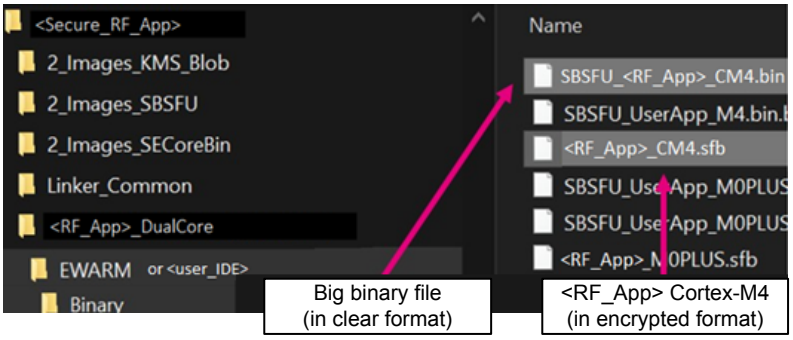
Figure 19. File structure of <RF_App> Cortex-M4 output



This step also generates the following files in the `Binary/` directory:

- `<RF_App>_CM4.sfb`: UserApp Cortex-M4 binary in encrypted format, including the SFU header
- `SBSFU_<RF_App>_CM4.bin`: final big binary that concatenates the SBSFU binaries and user-application binaries in clear format

Figure 20. File structure of <RF_App> Cortex-M4 encrypted + big binary



The `SBSFU_<RF_App>_CM4.bin` must be used to program the STM32WL5x Flash memory on first use. To generate a new firmware update, use `<RF_App>_CM0PLUS.sfb` or `<RF_App>_CM4.sfb` depending on the change location.

Note: The SBSFU provides an internal firmware version in the firmware header. To update this value, refer to the document [1].

3.2 How to download and execute the firmware

During the development, when a device is not fully protected, the firmware can be downloaded in two ways:

- entirely (SBSFU + UserApp) - see Section 3.2.1

  the final big binary `SBSFU_<RF_App>_CM4.bin` is downloaded using on the following method:

  – through the STM32CubeProgrammer tool

  – using a provided script that automates the generation and download processes

Warning:

This action requires to erase the full Flash memory of the device, and to remove all security option bytes. This can be done only if security option bytes allow it, which is typically not the case when option bytes are configured for production (such as RDP Level 2).

- partially using SFU (only UserApp) - see Section 3.2.2

  SBSFU is not downloaded but only used to handle the separate download of the `<RF_App>_CM4.sfb` or `<RF_App>_CM0PLUS.sfb` files. This is achieved through the Y-MODEM (part of the SBSFU) that handles the UART. This is the only way to update the LoRaWAN_SBSFU_1_Slot_DualCore and Sigfox_SBSFU_1_Slot_DualCore firmware when option bytes are configured for production.

3.2.1 Generate and download the big binary file

Specific documents describe how to use the STM32CubeProgrammer tool. This section focuses on scripts available in the project directory to facilitate the compilation and download.

Three scripts are available to automate the compilation of all SBSFU projects and the programming of the concatenate binary on the STM32WL5x Flash memory.

Figure 21. File structure of automated process scripts

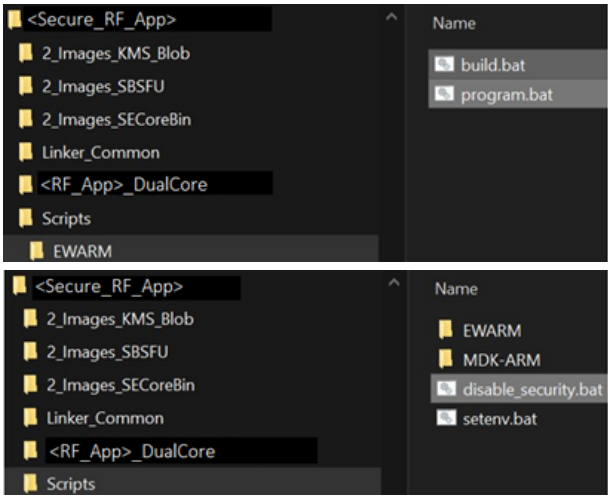


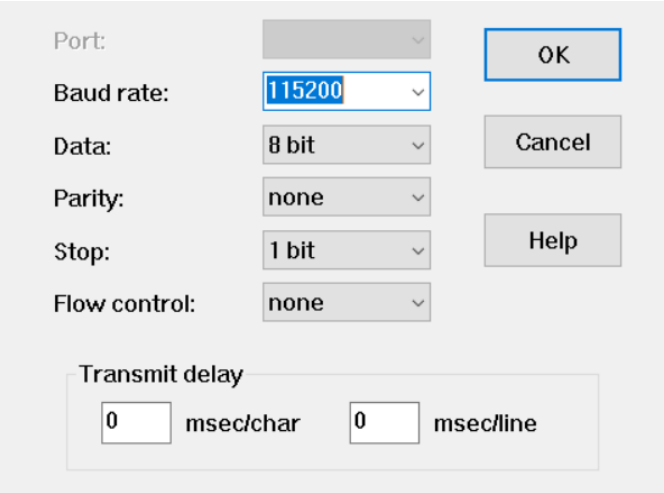
Table 7. Automated process scripts

Script	Description
`Scripts\EWARM\build.bat`	Compiles all project files with IAR Embedded Workbench (including `prebuild.bat` and `postbuild.bat` scripts) with the mandatory project order. The `-app` parameter is used to compile only the user application if the SBSFU projects are not modified.
`Scripts\EWARM\program.bat`	Runs the `disable_security.bat` script to remove the write access protection. Programs the `SBSFU_UserApp_M4.bin` to the STM32WL5x device, with STM32CubeProgrammer.
`Scripts\disable_security.bat`	Resets all option bytes to be compliant with a non-secure firmware (including a full erase memory).

Note: The path of the tools must be updated according to the versions and location of the user installations, by modifying `Scripts\setenv.bat` content.

Once the code is downloaded, unplug/plug the USB cable depending on the scenario. In order to see the SBSFU trace, the user can connect a terminal and configure the UART to 115200 bit/s.

Figure 22. Terminal configuration



When the SBSFU finished to validate the application integrity/authenticity, the SFBU directly starts the user application. The user application can use a different trace configuration. If the <RF_App> project has a different baudrate compared to the SBSFU baudrate, the terminal baudrate can be changed accordingly. Terminal settings can be changed dynamically but some traces may be lost during the switching. See Section 3.3 to solve this issue during debugging. Below the baudrate values used by most relevant RF projects:

- The Sigfox_PushButton_DualCore application uses UART with baudrate 9600 bit/s (the embedded firmware uses LPUART).
- LoRaWAN_AT_Slave_DualCore and Sigfox_AT_Slave_DualCore are not provided in secure version by the STM32CubeWL. The LPUART is used as well (to wake up the MCU from low-power when characters are sent). If the user wants to combine these projects with the SBSFU, after SBSFU execution, the user must switch the terminal baudrate to 9600 bit/s to use LoRaWAN_AT_Slave applications.
- LoRaWAN_End_Node_DualCore has the same baudrate as the SBSFU. So trace can be seen sequentially keeping baudrate 115200 bit/s.

3.2.2 How to update/download only <RF_App>_DualCore_CM0PLUS or <RF_App>_DualCore_CM4 via Y-MODEM

Section 3.1 details how to recompile only the application codes. The <RF_App> download is done by the SBSFU that keeps residing and running on the board. This is the typical way to update LoRaWAN_SBSFU_1_Slot_DualCore and Sigfox_SBSFU_1_Slot_DualCore in production.

Concerning LoRaWAN_FUOTA_DualCore, see the document [5].

In order to request to update the <RF_App> firmware, the user must follow these steps:

1. Press the push button 1 (PB1) on the board.
2. Hold PB1 down and press the reset button of the board.
3. Release PB1.

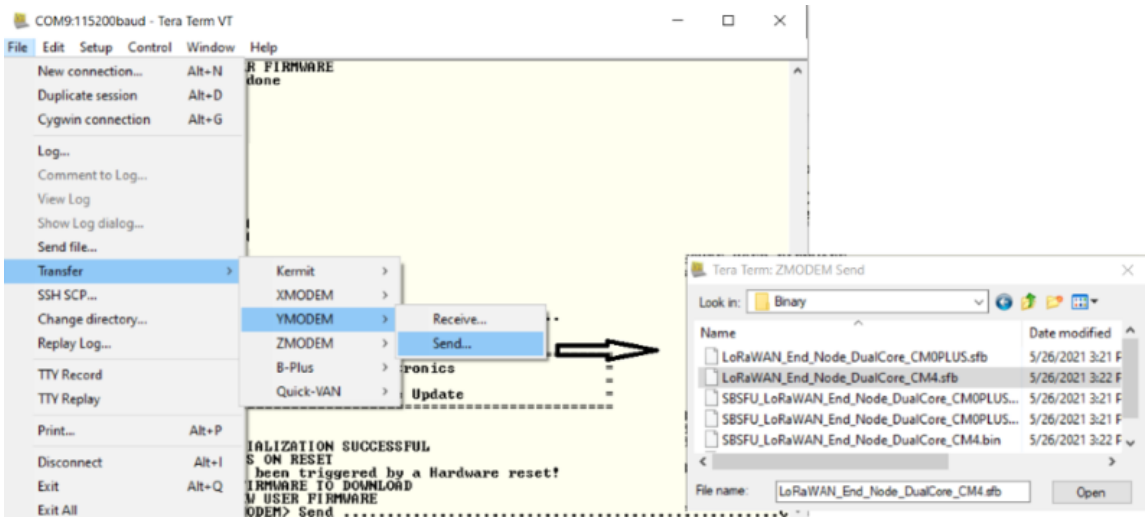
In this case, the SBSFU logs show that the Y-MODEM application waits for code to be downloaded.

Figure 23. Y-MODEM logs

```
= [SBOOT] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SBOOT] STATE: CHECK STATUS ON RESET
          INFO: A Reboot has been triggered by a Hardware reset!
= [SBOOT] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SBOOT] STATE: DOWNLOAD NEW USER FIRMWARE
          File> Transfer> YMODEM> Send .....
```

Updated <RF_App> is downloaded via UART, using the terminal interface.

Figure 24. How to use Y-MODEM from terminal



The user can access the <RF_App> generated in previous step by selecting `<RF_App>_CM4.sfb` or `<RF_App>_CM0PLUS.sfb`. The two cores cannot be downloaded simultaneously.

Once the file is transferred, the SBSFU validates the <RF_App> application integrity/authenticity, and directly starts the user application.

The user application can use a different trace configuration (such as UART baudrate) as described in Section 3.2.1 .

In summary, when using the board for the 1st time, the entire firmware (big binary) has to be downloaded via script or STM32CubeProgrammer. The power (USB cable) must be unplugged and plugged again. Then each time the board is reset, the SBSFU code runs first, and <RF_App> starts automatically once SBFU validated the integrity/authenticity of the firmware. To update the <RF_App>, the Y-MODEM routine can be started by holding PB1 pressed, while pressing the reset button.

Note:
- *Cortex-M4 and Cortex-M0+ <RF_App> must be compatible (same project, same version).*
- *The <RF_App> firmware can have different UART baudrate with respect to SBSFU baudrate.*

3.3 How to debug <RF_App>

The complete system consists of a Secure Boot and an <RF_App> application. When the target resets, the Cortex-M4 Secure Boot starts first. After a low-level initialization, the Cortex-M0+ SBSFU starts and checks all required security steps. If the SBSFU does not detect any system error, the two Secure Boot codes (Cortex-M4 and Cortex-M0+) jump to the entry point of Cortex-M4 and Cortex-M0+ applications.

Since the <RF_App> application is linked to the Secure Boot, the `<RF_App>_CM<x>.bin` binaries cannot be downloaded directly with the debugger. The code start running directly: the debugger does not stop at the beginning of the `main()` function.

To allow debug, SBSFU compilation flags must be compiled in addition to the <RF_App> compilation flags.

3.3.1 Configure SBSFU firmware to allow debug

The following steps are needed:

- In `\2_Images_SBSFU\CM0PLUS\app_sfu.h` (see Figure 7), change or undefine the following code:

```
/*#define SFU_RDP_PROTECT_ENABLE*/
/*#define SFU_C2SWDBG_PROTECT_ENABLE*/
```

- In `\2_Images_SBSFU\Common\app_sfu_common.h` (see Figure 5), change or undefine the following code:

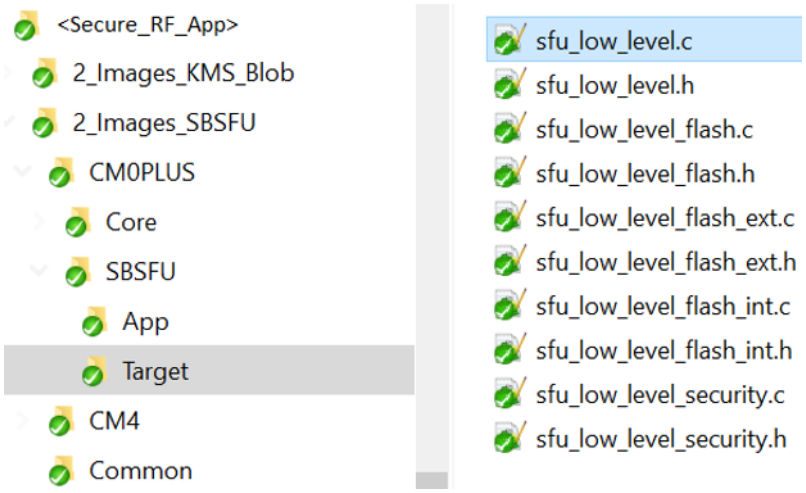
```
/*#define SFU_DAP_PROTECT_ENABLE*/
/*#define SFU_C2_DDS_PROTECT_ENABLE*/
#define SFU_HIDE_PROTECTION_CFG OB_SECURE_HIDE_PROTECTION_DISABLE
```

The user can also change the SBSFU baudrate to align it to the <RF_App> one, with the code

```
set UartHandle.Init.BaudRate = <myBaydrate>;
```

in `\2_Images_SBSFU\CM0PLUS\SBSFU\Target\sfu_low_level.c`.

Figure 25. UART baudrate configuration



For the example of a Sigfox_PushButton_DualCore application, the user can set `<myBaudrate> = 9600` to avoid switching the hyper-terminal value. The drawback is that it slows the download operation.

See the document [1] for more details on how to debug an application running on SBSFU.

3.3.2 Configure <RF_App> firmware to allow debug

Set the debugger and low power defines on Cortex-M0+ and/or Cortex-M4 in `\<RF_App>_DualCore\CM<x>\Core\Inc\sys_conf.h`

Figure 26. File structure of End_Node dual-core debug configuration


```
/**
  * @brief Enable MCU Debugger pins (dbg serial wires, sbg spi, etc)
  */
#define DEBUGGER_ENABLED            0

/**
  * @brief Disable Low Power mode
  * @note  0: LowPowerMode enabled. MCU enters stop2 mode,
  *             1: LowPowerMode disabled. MCU enters sleep mode only
  */
#define LOW_POWER_DISABLE           0
```

The `DEBUGGER_ENABLED` flag allows the debugger to attach via the serial wires. This flag can be enabled only on one core or on both, depending on the core to be debugged.

`LOW_POWER` must be disabled at least on one of the two cores, otherwise when the device is in Stop mode, the debugger does not wake up anymore. It is usually simpler to disable `LOW_POWER` on the core that needs to be debugged, unless debugging a low-power issue.

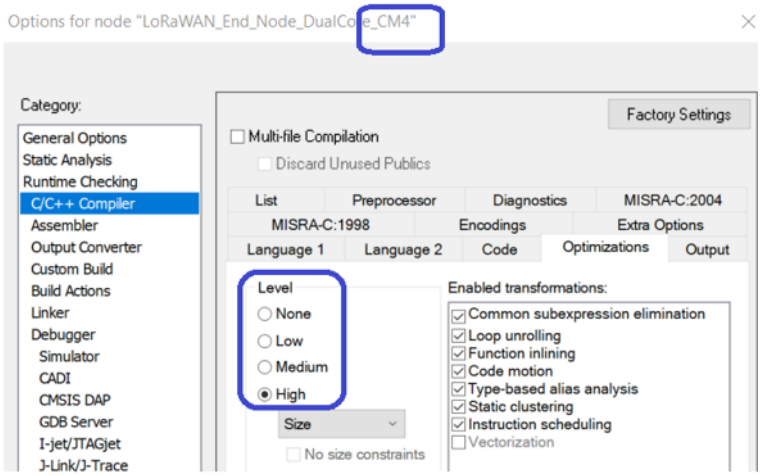
`PROBE_PINS` can be optionally enabled to monitor the behavior of internal signals via an oscilloscope or a logic analyzer.

```
/**
  * @brief Enable four wires usable as probes
  */
#define PROBE_PINS_ENABLED          0
```

The compiler option can be changed to an inferior optimization level if the memory space allows it (memory space available for that core). The memory space depends on <RF_App> and on the IDE used to compile.

<Secure_RF_App> is such that the maximum space is reserved for <RF_App>_CM4, which allow the Cortex-M4 application to be recompiled with lower optimization level.

Figure 27. Compile optimization level (example for IAR Embedded Workbench)



To do the same with <RF_App>_CM0PLUS, the memory mapping must be reworked (mapping provided in examples does not allocate enough Flash memory to Cortex-M0+, see Section 7.2) .

Refer to the document [1] for details about SBSFU debugging.

3.3.3 Compile the big binary file and download

Follow the steps described at the beginning of this Section 3.1 to compile and download the code.

3.3.4 Attach the debugger

When <RF_App> is compiled with `DEBUGGER_ENABLED` set to 1 at the beginning of the code, a while (1) loop is added that waits the PB1 to be pressed. This avoids the code going too far in the execution after SBSFU started the <RF_App> execution. Before pressing PB1, the user can perform the following actions:

- Attach the debugger.
- Set the wished breakpoints.
- Adapt the terminal baudrate if it differs from the <RF_App> one.

The PB1 can be pressed to reach the first breakpoint, and the user can play with the debugger as usual (for example stepping).

4 Privileged/unprivileged coding

The following major difference between firmware provided for non-secure <RF_App> projects versus secure ones, improves the Cortex-M0+ security:

- The firmware for the non-secure <RF_App> projects (like classical \LoRaWAN\LoRaWAN_End_Node) always runs in privileged mode.
- The firmware for the <Secure_RF_App> projects runs as much as possible in unprivileged mode, and switches in privileged mode only when necessary (only true for Cortex-M0+ code).

The unprivileged mode is more resistant to hacker attacks. Hackers use many ways to break a non-secure firmware, sometime even just playing with data input (such as giving as parameter a function that reads a buffer a size bigger than the buffer size itself).

SBSFU ensures that only “trusted” applications are installed on the device. No malicious code can be downloaded to read internal data. But, if the application code is not written carefully (for example without checking that the `size` parameter is minor or equal than the buffer size), hackers can succeed to extract information despite the SBSFU protection. If pointer ranges are not checked, a `write` function can be used to change a register value.

Thanks to the GTZC (configured by the SBSFU), sensitive data and registers on STM32WL devices are only accessible by the Cortex-M0+. Writing not carefully Cortex-M4 code is not such an issue. But to ensure that all the Cortex-M0+ code (around 50 Kbytes) is written carefully requires specific expertise and is costly in term of development time and code size. The Cortex-M0+ code is mainly <RF> protocol stack written by third parties that are not necessarily concerned by security.

GTZC can be configured to provide an additional restriction: access to sensitive data and registers allowed only by the Cortex-M0+ code when running in privileged mode. For example, MPU registers can only be accessed in privileged mode. By running most of the code in unprivileged mode, the remaining privileged code that hackers can use is strongly reduced. In addition, this small portion of privileged code can be written carefully.

Note: The unprivileged code has its own memory stack separated by the main stack used in privileged mode. Refer to the Arm documentation for details about Thumb states (Handler versus Thread mode), MSP (main stack pointer) vs PSP (process stack pointer). Exceptions and interrupt service routines always run in privileged mode and use the MSP.

This section explains how the Cortex-M0+ code has been adapted to run most of the time in unprivileged mode.

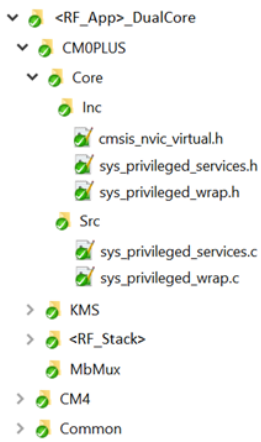
The main concern is that parts of the code need to run in privileged mode. The firmware must be able to switch between the modes when requested. Once in unprivileged mode, switch back in privileged mode is possible via a SVC call. This is required for certain instruction and registers access such as:

- NVIC inline functions handling
- critical sections and low-power
- cryptographic functions

Note: Cryptographic functions need a specific attention because they are encapsulated in the SKMS (part of the SBSFU binary) that needs to be called from the <RF_App> binary.

Secure projects contain five additional files that provide services to abstract the privileged/unprivileged switch.

Figure 28. `sys_privileged_services.c/h` and `sys_privileged_wrap.c/h`



Some other Cortex-M0+ files have been modified for the scope, such as:

- <RF_App>_DualCore\CM0PLUS\Core\Inc\sys_conf.h to define SECURE_UNPRIVILEGE_ENABLE.
- <RF_App>_DualCore\CM0PLUS\Core\Inc\stm32wlxx_hal_conf.h, stm32wlxx_it.c/h and main.c to define the GTZC HAL and TZIC_IRQHandler().
- <RF_App>_DualCore\CM0PLUS\Core\Inc\utilities_def.h and CM0PLUS\LoRaWAN\Target\radio_conf.h to add the following processes to ensure the code does not run anymore in ISR (interrupt service routine, privileged) but on sequencer task (unprivileged):
  - CFG_SEQ_Task_RadioIrq_Process
  - CFG_SEQ_Task_RadioRxTimeout_Process
  - CFG_SEQ_Task_RadioTxTimeout_Process
  - CFG_SEQ_Task_UtilTimer_Process
- <RF_App>_DualCore\CM0PLUS\Core\Inc\utilities_conf.h to redefine UTIL_xxx_CRITICAL_SECTION() macros.
- <RF_App>_DualCore\CM0PLUS\Core\Src\sys_app.c to switch to unprivileged mode and to resynchronize the two cores at initialization.

  In non-secure mode, the Cortex-M0+ is started by the Cortex-M4. When SBSFU is used, it starts both <RF_App> cores simultaneously and the Cortex-M0+ can start before Cortex-M4. A resynchronization is required to initialize MBMUX in the correct order.
- <RF_App>_DualCore\CM0PLUS\MbMux\mbmux.c to double check the addresses of the Cortex-M0+ buffers before using them. This double check avoids basic fault injection by hardware (such as power glitched) that can lead to jump the single check inside the MBMUX_SEC_VerifySramBuffer function.
- <RF_App>_DualCore\CM4\LoRaWAN\App\lora_app.c and CM4\Core\Src\sys_app.c to implement a push button at start to facilitate the debug (see Section 3.3), and to resynchronize the two cores at initialization (complement of Cortex-M0+ side).

In order to change from privileged to unprivileged, the Cortex CONTROL register must be set with the __set_CONTROL instruction in CM0PLUS\Core\Src\sys_app.c with the code below.

```
ThumbState_RemapMspAndSwitchToPspStack();
ThumbState_EnterUnprivilegedMode();
```

Once in unprivileged mode, several registers (including the CONTROL one) cannot be changed. An interrupt (specifically the SVC call) can be used to return in privileged mode. The SVC call causes an interrupt handled by __SVC_Handler() that calls SVC_APP_Handler() (see svc_handler.s).

SVC_APP_Handler() is defined in sys_privileged_services.c as follow:

```
void SVC_APP_Handler(uint32_t *args)
{
  uint8_t svc_index = ((char *)args[6])[-2];

  switch (svc_index)
  {
    case 0x0: /* SE SVC CALL : called by SECoreBin*/
      SE_APP_SVC_Handler(args);
      break;
    case 0x1:
      APP_CRITICALSECTION_SVC_Handler(args);
      break;
    case 0x2:
      APP_NVIC_SVC_Handler(args);
      break;
    default:
      break;
  }
}
```

The three subcases are detailed in Section 4.1 and Section 4.2 starting from the last (the easier) to the first.

4.1 NVIC

Access to NVIC registers requires privileged mode. Each time the existing code calls NVIC, the function must be deviated (redefined) to go through SVC first.

The CMSIS inline function can be remapped thanks to `cmsis_nvic_virtual.h` (see `core_cm0plus.h` when `CMSIS_NVIC_VIRTUAL` is defined). `cmsis_nvic_virtual.h` remaps the NVIC inline function as follow:

```
#define NVIC_EnableIRQ              SYS_PRIVIL_NVIC_EnableIRQ
#define NVIC_GetEnableIRQ           SYS_PRIVIL_NVIC_GetEnableIRQ
#define NVIC_DisableIRQ             SYS_PRIVIL_NVIC_DisableIRQ
#define NVIC_GetPendingIRQ          SYS_PRIVIL_NVIC_GetPendingIRQ
#define NVIC_SetPendingIRQ          SYS_PRIVIL_NVIC_SetPendingIRQ
#define NVIC_ClearPendingIRQ        SYS_PRIVIL_NVIC_ClearPendingIRQ
#define NVIC_SetPriority            SYS_PRIVIL_NVIC_SetPriority
#define NVIC_GetPriority            SYS_PRIVIL_NVIC_GetPriority
#define NVIC_SystemReset            SYS_PRIVIL_NVIC_SystemReset
```

where:

- `SYS_PRIVIL_NVIC_xxx(…)` are defined in `sys_privileged_wrap.c/h`.
- `SYS_PRIVIL_NVIC_xxx(…)` call the SVC that causes the SVC interrupt (switching in privileged mode).
- The SVC interrupt calls `SVC_APP_Handler(…)` with `svc_index=0x2`.
- `SVC_APP_Handler(…)` calls `APP_NVIC_SVC_Handler(…);` that is defined with a parameter identifying the NVIC function to be called.
- The classical NVIC inline function is finally called in privileged mode.

When the SVC interrupt call ends, the system automatically goes back to unprivileged mode.

4.2 Critical sections

In the non secure <RF_App>, `UTIL_xxx_CRITICAL_SECTION()` macros are defined as follows:

```
#define UTILS_ENTER_CRITICAL_SECTION() uint32_t primask_bit= __get_PRIMASK();\
                                       __disable_irq()
#define UTILS_EXIT_CRITICAL_SECTION()  __set_PRIMASK(primask_bit)
```

This code does not work if called on SVC interrupt (as seen in Section 4.1 for NVIC). After calling `ENTER_CRITICAL_SECTION()`, the execution goes to unprivileged after `__disable_irq()`. All interrupts are disabled including the SVC one that cannot be used to switch back to privileged mode (such as `EXIT_CRITICAL_SECTION`). When the code between entering and exiting critical sections need to be executed in unprivileged mode, `__disable_irq()` cannot be used.

The `UTIL_xxx_CRITICAL_SECTION()` macros must be redefined in `CM0PLUS\Core\Inc\utilities_conf.h` as follows:

```
#define UTILS_ENTER_CRITICAL_SECTION() nvic_iser_state= SYS_PRIV_EnterCriticalSection()
#define UTILS_EXIT_CRITICAL_SECTION()  SYS_PRIV_ExitCriticalSection(nvic_iser_state)
```

where

```
uint32_t SYS_PRIVIL_EnterCriticalSection( void )
{
  uint32_t nvic_iser_state;

  if (ThumbState_IsUnprivileged() != 0)
  {
    /* disable NVIC irqs, then back to PSP and Unpriv */
    SYS_CRITICALSECTION_SvcCall(&nvic_iser_state, SVC_DISABLE_ALL_NVIC_IRQS);
  }
  else
  {
    nvic_iser_state = NVIC->ISER[0];
    NVIC->ICER[0] = nvic_iser_state;
  }
  return nvic_iser_state;
}
```

and

```
void SYS_PRIVIL_ExitCriticalSection( uint32_t nvic_iser_state)
{
  if (ThumbState_IsUnprivileged() != 0)
  {
    uint32_t dummy_ret = 0;
    SYS_CRITICALSECTION_SvcCall(&dummy_ret, SVC_RESTORE_NVIC_IRQS, nvic_iser_state);
  }
  else
  {
    NVIC->ISER[0] = nvic_iser_state | NVIC->ISER[0];
  }
}
```

`SYS_CRITICALSECTION_SvcCall()` calls `_svc(#0x1)` (`SVC_APP_Handler(0x1)`), that calls `APP_CRITICALSECTION_SVC_Handler(…)`.

To avoid calling SVC when not necessary (for example when the caller runs already in privileged mode), the following check can be done:

```
if (ThumbState_IsUnprivileged() != 0)
```

When `SECURE_UNPRIVILEGE_ENABLE == 1`, this check results always true, and the SVC call is used.

`APP_CRITICALSECTION_SVC_Handler(…)` does not disable all interrupts but only the NVIC ones. The SVC interrupt (switching from unprivileged to privileged) can still be used.

```
void APP_CRITICALSECTION_SVC_Handler(uint32_t *args)
{
  uint32_t nvic_iser_state;

  switch (args[1])
  {
    case SVC_DISABLE_ALL_NVIC_IRQS:
    {
      nvic_iser_state = NVIC->ISER[0];
      NVIC->ICER[0] = nvic_iser_state;
      break;
    }
    case SVC_RESTORE_NVIC_IRQS:
    {
      NVIC->ISER[0] = args[2] | NVIC->ISER[0];
      break;
    }
```

Critical section and low power mode

An additional constraint must be managed if entering in a critical section when going in low-power mode (STM32CubeWL examples work in Stop mode). In order to wake up from a low-power mode, `_WFI` is expected. Typically `_WFI` is triggered by NVIC interrupts.

The sequencer code main loop is implemented as follows:

```
  UTIL_SEQ_ENTER_CRITICAL_SECTION_IDLE( );
  if (!(((TaskSet & TaskMask & SuperMask) != 0U) || ((EvtSet & EvtWaited)!= 0U)))
  {
      UTIL_PowerDriver.EnterSleepMode( );
      UTIL_PowerDriver.ExitSleepMode( );
  }
  UTIL_SEQ_EXIT_CRITICAL_SECTION_IDLE( );
```

If `xxx_CRITICAL_SECTION_IDLE()` are implemented by clearing all NVIC interrupts with the code:

```
    nvic_iser_state = NVIC->ISER[0];
    NVIC->ICER[0] = nvic_iser_state;
```

this prevents `_WFI` to wake up the MCU from Stop mode. `__disable_irq()` must be used as it does not prevent `_WFI`, but this disables all interrupts, including the SVC one.

The `ENTER_CRITICAL_SECTION_IDLE()` macro must be written to remain in privileged mode at the end of the SVC interrupt service routine. If the execution mode goes to unprivileged, it is not anymore possible to exit the critical section.

In `CM0PLUS\Core\Inc\utilities_conf.h`, the `xxx_CRITICAL_SECTION_IDLE()` macros are redefined as follows:

```
#define UTIL_SEQ_ENTER_CRITICAL_SECT_IDLE()   SYS_PRIVIL_DisableIrqsAndRemainPriv()
#define UTIL_SEQ_EXIT_CRITICAL_SECTION_IDLE() SYS_PRIVIL_EnableIrqsAndGoUnpriv()
```

In `sys_privileged_wrap.c`, `SYS_PRIVIL_DisableIrqsAndRemainPriv()` calls the SVC. The SVC handler, after disabling the IRQs, sets the MCU control to remain in privileged mode.

```
void APP_CRITICALSECTION_SVC_Handler(uint32_t *args)
{
  switch (args[1])
  {
    case SVC_DISABLE_ALL_NVIC_IRQS:
      /*all __NVIC_GetEnableIRQ*/
      nvic_iser_state = NVIC->ISER[0];
      /* clear all positive interrupt e.g. no see IRQn_Type in core_cm0plus.h*/
      NVIC->ICER[0] = nvic_iser_state;
      break;
    case SVC_RESTORE_NVIC_IRQS:
      NVIC->ISER[0] = args[2] | NVIC->ISER[0];
      break;
    case SVC_DISABLE_ALL_EXCEPTIONS:
      __disable_irq();
      __set_CONTROL(__get_CONTROL() & 0xFFFE); /* bit 0 = 0: remain privileged */
      /* note: exiting the SVC will go back to PSP stack but remain priv mode */
      break;
  }
}
```

The code executed between entering and exiting critical section is small (not a big problem if executed in privileged mode).

`UTIL_SEQ_EXIT_CRITICAL_SECTION_IDLE()` is executed in privileged mode, and does not need to call SVC. This function is mapped on `SYS_PRIVIL_EnableIrqsAndGoUnpriv()` defined directly in `sys_privileged_wrap.c` as follows:

```
void SYS_PRIVIL_EnableIrqsAndGoUnpriv(void)
{
  __enable_irq();
  ThumbState_EnterUnprivilegedMode(); /* Goes always Unpriv */
}
```

Note:

- *The `primask` value is not saved in `UTIL_SEQ_ENTER_CRITICAL_SECTION_IDLE()` (like in the original `CRITICAL_SECTION` macro). As these two functions are only called in the main sequencer loop (part of `UTIL_SEQ_Run()` function), they can never be encapsulated under another critical section. `UTIL_SEQ_ENTER__CRITICAL_SECTION_IDLE()` can only be executed when `primask = 0`. For the same reason, when calling `UTIL_SEQ_EXIT_CRITICAL_SECTION_IDLE()`, `__enable_irq()` can be used instead of `__set_PRIMASK(primask_bit)`.*
- *As functions `UTIL_SEQ_ENTER__CRITICAL_SECTION_IDLE()` and `UTIL_SEQ_EXIT_CRITICAL_SECTION_IDLE()` are called in the context of `UTIL_SEQ_Run()`, when the compilation flag `SECURE_UNPRIVILEGE_ENABLE == 1`, the function `SYS_PRIVIL_DisableIrqsAndRemainPriv()` is always supposed to be executed in unprivileged mode. If `SYS_PRIVIL_DisableIrqsAndRemainPriv()` is called in privileged mode, the SVC is skipped thanks to `ThumbState_IsUnprivileged()`.*

  Whatever the entering execution mode, `SYS_PRIVIL_EnableIrqsAndGoUnpriv()` switches the mode to unprivileged at exit. This is an additional security.

4.3 Cryptographic functions

As mentioned in Section 3.1 , four binaries are downloaded on the STM32CubeWL: two binaries on the Cortex-M4 and two binaries on the Cortex-M0+.

The cryptographic functions encapsulated in the SKMS are part of the SBSFU binary running on the Cortex-M0+. The <RF_App> application binary uses these functions. The link between the two binaries is handled by `se_interface_appli.o` generated via script (see `se_interface_appli.txt` and SBSFU documentation [1] and [2]).

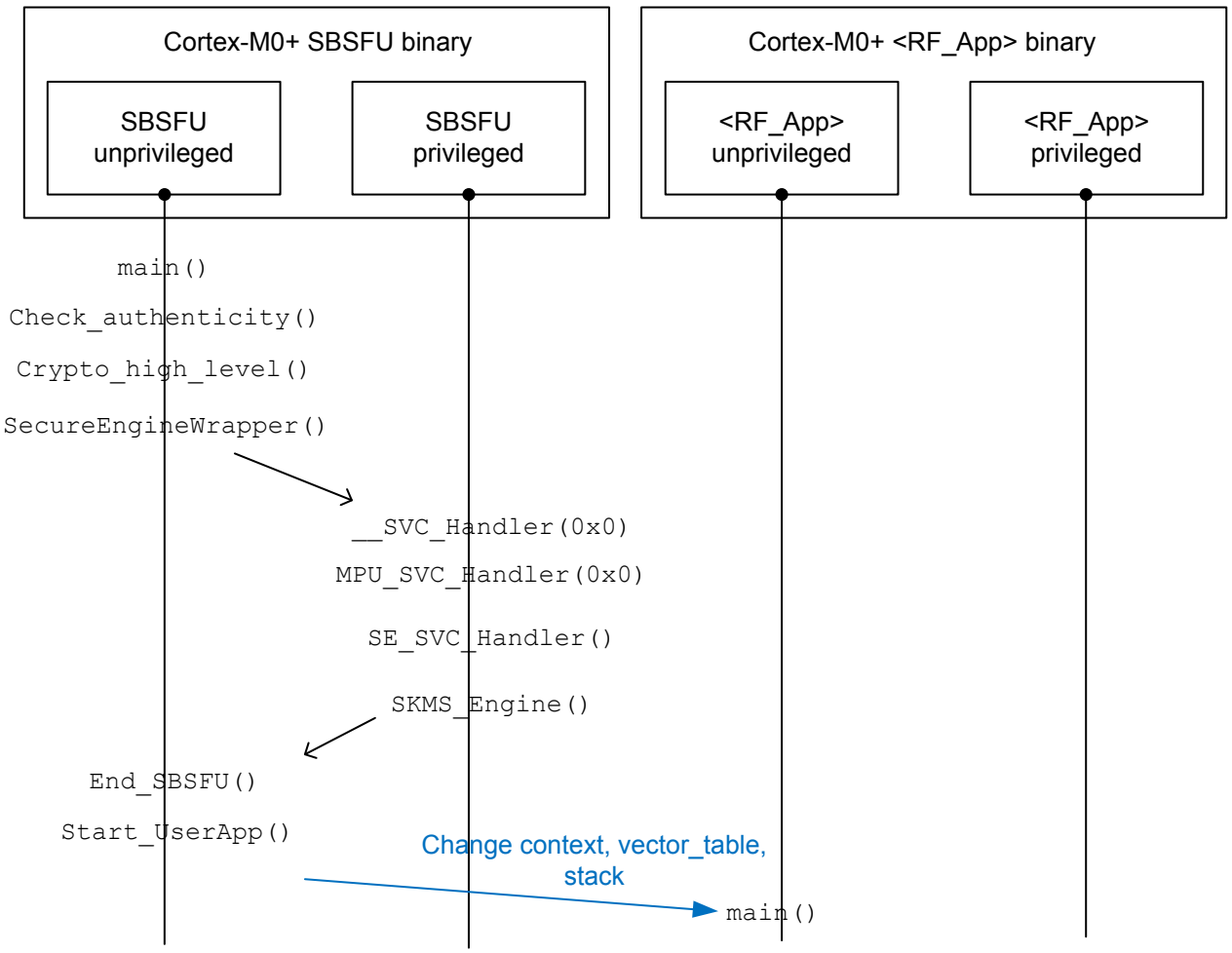
The SBSFU runs in both privileged and unprivileged mode.

When cryptographic functions are called in the SBSFU binary, a wrapper moves to privileged via SVC before calling the cryptographic functions (same as <RF_App> binaries in Section 4.2). This SBSFU mechanism is named SecureEngineWrapper in this application note (simplification to keep consistency with the previous section).

When the SBSFU is combined with a user application (such as <RF_App>), the SBSFU code configures the security features (like TZ, MPU, IWDG, or DAP), checks the integrity/authenticity of the <RF_App> application binary, and, if requested, downloads a new version. For these checks, the SBSFU uses SKMS functions called by the SBSFU binary.

After all these actions, the SBSFU 'jumps' to the application binary (just downloaded or already present), and remaps the interrupt vector table on one of the new binary. The <RF_App> application binary has its own main, its own interrupt vector table, and its own SVC handler.

Figure 29. SBSFU binary calling SKMS for integrity and authenticity checks



The <RF_App> starts its execution and needs SKMS at a time to encrypt/decrypt RF transmission keys (such as LoRaWAN or Sigfox ones).

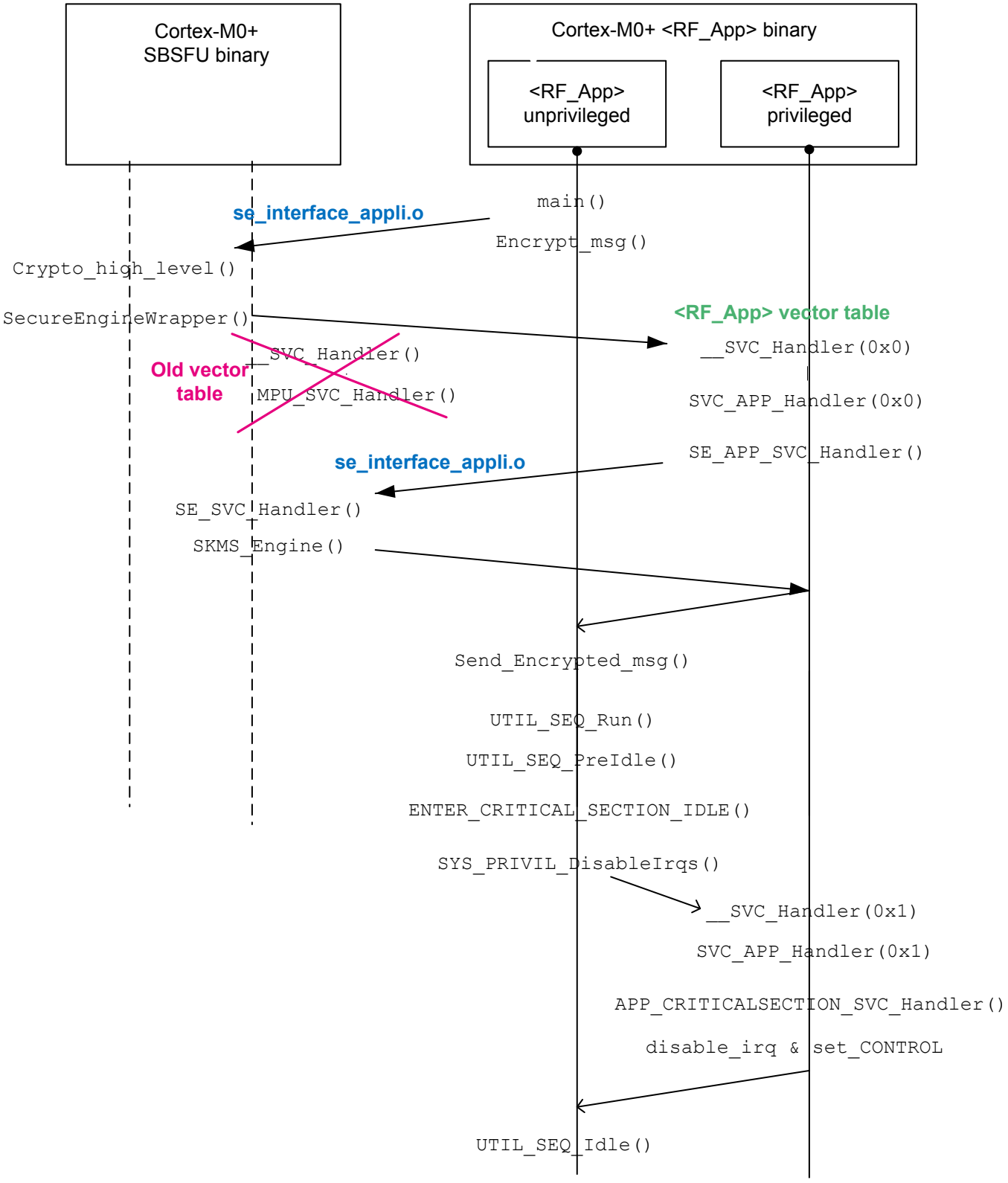
The non-secure <RF_App> uses non-secure KMS as part of the <RF_App> binary. The following non-secure KMS files are part of `<RF_App>_DualCore\CM0PLUS\Core` and the associated binary:

- `ca_low_level.c/h`
- `kms_low_level.c/h`
- `kms_platf_objects_config.h`
- `kms_platf_objects_interface.h`

The secure <RF_App> uses the SKMS from the SBSFU binary. The above files are removed from the secure <RF_App> project.

Thanks to `se_interface_appli.o`, <RF_App> calls `Crypto_high_level()` on the SBSFU binary to encrypt/decrypt the transmission keys. The SBSFU SecureEngineWrapper calls the SVC but the interrupt vector table is now the one associated to the <RF_App> application, and the program counter goes to the SVC handler of the <RF_App> application.

Figure 30. <RF_App> binary calling SKMS (part of SBSFU binary)



The `__SVC_Handler()` of the user application calls the SVC_APP_Handler (instead of the MPU_SVC_Handler, see `svc_handler.s`). The SVC_APP_Handler must then be compatible to decode the SVC indexes of the SBSFU cryptographic function, by using the same indexes coded on the SBSFU code.

From `sfu_mpu_isolation.c`, the SBSFU code (MPU_SVC_Handler) `__SVC_Handler()` uses these indexes:

```
void MPU_SVC_Handler(uint32_t *args)
{
  uint8_t code = ((uint8_t *)args[6])[-2];
  switch (code)
  {
    case 0x0:
      /* A Secure Engine service is called */
      SE_SVC_Handler(args) ;
      break;
    case 0x1:
      /* Internal SB_SFU privileged service */
      SFU_MPU_SVC_Handler(args);
      break;
    default:
      HAL_NVIC_SystemReset();
      break;
  }
}
```

The cryptographic functions used by the application are in the `switch case 0x0`, using the SE_SVC_Handler for the secure engine. <RF_App> must then reserve the `switch case 0x0` for the same purpose: 0x0 is the value called by the SBSFU cryptographic high-level code (part of the SBSFU binary).

```
void SVC_APP_Handler(uint32_t *args)
{
  uint8_t svc_index = ((char *)args[6])[-2];
  uint32_t nvic_iser_state;

  switch (svc_index)
  {
    case 0x0: /* SE SVC CALL : called by SECoreBin*/
      SE_APP_SVC_Handler(args);
      break;
    case 0x1:
      APP_CRITICALSECTION_SVC_Handler(args);
      break;
    case 0x2:
      APP_NVIC_SVC_Handler(args);
      break;
    default:
      break;
  }
}
```

Note: `SE_APP_SVC_Handler()` is the interface given in `se_interface_application.o` that is defined on the SBSFU as follows:

```
__root void SE_APP_SVC_Handler(uint32_t *args)
{
  SE_SVC_Handler(args);
}
```

5 Memory mapping

The Flash memory mapping of the device contains some elements described in the table below. The following items concern exclusively the SBSFU that runs before switching to <RF_App> execution:

- SB CM4: Secure Boot binary
- SBSFU + SE CM0+: Secure Boot, Secure Firmware Update, Secure Engine and SKMS binary
- Firmware Header: Flash memory area where the not contiguous firmware headers are stored
- Blob download area: not used in the scope of the provided <Secure_RF_App> projects

The items listed below are also used by the <RF_App>:

- Active slots (contains <RF_App> executable code downloaded via SBSFU)
- KMS Data Storage (non-volatile memory area where RF session keys are dynamically derived via SKMS on <RF_App> request)
- User/SE keys (<RF_App> and secure-engine static embedded keys)

Table 8. Flash memory mapping

Start address	End address	Size (Kbytes)	Flash memory region
0x0800 0000	0x0800 27FF	10	Secure Boot Cortex-M4
0x0800 28FF	0x0800 2FFF	2	Blob download(1)
0x0800 3000	0x0801 BFFF	100	Slot Active 2 <RF_App>_CM4
0x0801 C000	0x0802 AFFF	60(2)	Slot Active 1 <RF_App>_CM0
0x0802 B000	0x0802 CFFF	8	KMS Data Storage(3)
0x0802 D000	0x0802 E3FF	5	SE interface Cortex-M0+
0x0802 E000	0x0803 67FF	33	SBSFU Cortex-M0+
0x0803 0000	0x0803 E7FF	32	SE Cortex-M0+
0x0803 E800	0x0803 EFFF	2	UserApp and SE embedded keys(3)
0x0803 F000	0x0803 F7FF	2	SLOT Active 2 header(3)(4)
0x0803 F800	0x0803 FFFF	2	SLOT Active 1 header(3)(4)

1. *Not used by <Secure_RF_App> projects.*
2. *if LoRaWAN_End_Node_DualCore, 60 Kbytes are allocated for LoRaWAN Cortex-M0+ code. If Sigfox_PushButton_DualCore, the 60 Kbytes are the sum of Sigfox Cortex-M0+ code (56 Kbytes) and EE data storage (4 Kbytes) .*
3. *Accessible by the Cortex-M0+ only.*
4. *Flash memory area where not-contiguous firmware headers are stored.*

The RAM memory mapping of the device contains some elements described in the following table.

Table 9. RAM mapping

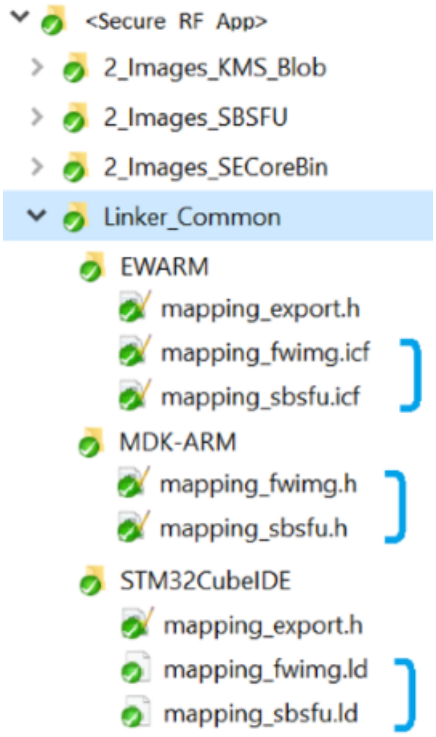
Start address	End address	Size (Kbytes)	RAM region
0x2000 0000	0x2000 0CDF	3	Secure Boot Cortex-M4(1)
0x2000 0CE0	0x2000 0CFF	0.25	Cortex-M0+/Cortex-M4 synchronization flag(1)
0x2000 0D00	0x2000 7FFF	28.75	<RF_App>_CM4(1)
0x2000 8000	0x2000 83FF	1	SHARED_MailBox_MEM1 (allocated by the Cortex-M4)(2)
0x2000 8400	0x2000 8FFF	3	SHARED_MailBox_MEM2 ((allocated by the Cortex-M0+)(2)
0x2000 9000	0x2000 D3FF	17	<RF_App>_CM0+ and SBSFU Cortex-M0+(3)
0x2000 D400	0x2000 FFFF	11	SE Cortex-M0+(3)

1. *SRAM1.*
2. *SRAM2 accessible by Cortex-M4 and Cortex-M0+.*
3. *SRAM2 accessible by Cortex-M0+ only.*

The major boundaries are described in two common linker script files in the `Linker_Common` folder:

- linker file for IAR Embedded Workbench and STM32CubeIDE
- include files are for MDK-ARM

Figure 31. File structure of linker_common



These linker files, thanks to `mapping_export.h`, contain the major boundaries that can be refined in the linker files of the separate projects (2_Image_SBSFU, 2_Image_SeCoreBin, <RF_App>). Refer to the document [1] for more details about this configuration.

The STM32WL devices have the two following blocks of RAM:

- SRAM1 with no retention memory goes until 0x2000 7FFF.
- SRAM2 (starting at 0x2000 8000) has retention properties. In <RF_App> projects configuration, the first part of this memory is not secured (shared between the Cortex-M4 and the Cortex-M0+). The rest is secured (only accessible by the Cortex-M0+).

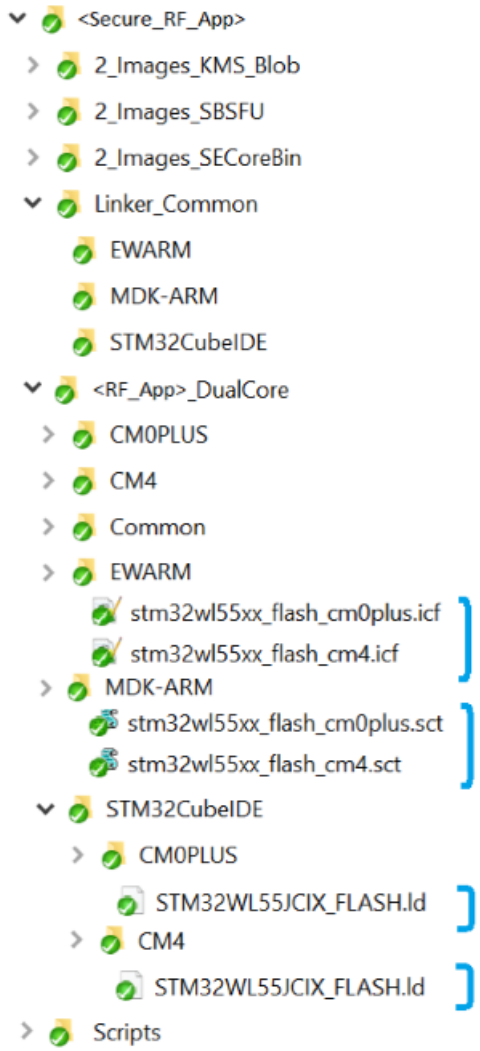
MBMUX is a mailbox communication system between the two cores designed for the <RF_App> on STM32WL5x dual-core devices (see the document [7] for more details). MBMUX uses a part of RAM shared by the two cores for exchanging messages.

The shared memory is divided in two sections:

- SRAM2_SHARED section MB_MEM1: allocated/placed by the Cortex-M4 linker file (contains the mapping table and the wrapper for the Cortex-M4 function calls)
- SRAM2_SHARED section MB_MEM2: allocated/placed by the Cortex-M0+ linker file (contains the wrapper for the Cortex-M0+ function calls, including the buffer for traces)

The RAM repartition is defined into two <RF_App> linker files depending on the IDE.

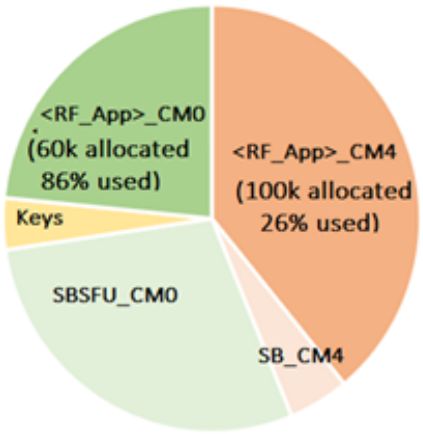
Figure 32. File structure of <RF_App> linker



6 Memory footprint

The previous section shows that almost 96 Kbytes are allocated for SBSFU and 160 Kbytes remain available for the RF application. This section shows that the memory occupied by each block is smaller than what is allocated for this block. The next section explains why it happens and how the mapping can be adapted to the user needs.

Figure 33. Allocation of 256-Kbyte Flash memory (<Secure_RF_App> projects)



6.1 RF dual-core applications

6.1.1 LoRaWAN End_Node dual-core application

Values in Table 10 and Table 11 are measured for the following configuration of the IAR Embedded Workbench compiler (EWARM version 8.30.1):

- optimization level 3 for size
- debug option off
- trace option VLEVEL_M (medium traces)
- target: STM32WL55JC
- End_Node dual-core application
- LoRaMAC Class A
- LoRaMAC region EU868 only

Table 10. Memory footprint for LoRaWAN_Secure_DualCore_End_Node_CM0PLUS

Module	Description	Sum of read only [FLASH]	Sum of read/write [RAM]
Application	Core, application and target components	2577	164
HAL	STM32WL HAL and LL drivers	5402	0
IAR Lib	Proprietary IAR libraries	1012	0
IAR Startup	Int_vect, init routines, init table, CSTACK and HEAP	499	4096
LoRaWAN stack	Middleware LmHandler interface, crypto, MAC and Region	26758	3572
MBMUX	Mailbox multiplexer wrappers and services	2871	856
SubGHz_Phy	Middleware radio interface	7453	413
Utilities	All STM32 services (sequencer, time server, low-power mgr, trace, mem)	3091	1740
Total		49663	10841

Table 11. **Memory footprint for LoRaWAN_Secure_DualCore_End_Node_CM4**

Module	Description	Sum of read only [FLASH]	Sum of read/write [RAM]
Application	Core, application and target components	5115	861
HAL	STM32WL HAL and LL drivers	13693	84
IAR Lib	Proprietary IAR libraries	1340	0
IAR Startup	Int_vect, init routines, init table, CSTACK and HEAP	791	2049
MBMUX	Mailbox multiplexer wrappers and services	2395	918
Utilities	All STM32 services (sequencer, time server, low power mgr, trace, mem)	2714	1740
Total		26048	5652

6.1.2 Sigfox push-button dual-core application

Values in the tables below are measured for the following configuration of the IAR Embedded Workbench compiler (EWARM version 8.30.1):

- optimization level 3 for size
- debug option off
- trace option VLEVEL_M (medium traces)
- target: STM32WL55JC
- PushButton dual-core application

Table 12. **Memory footprint for Sigfox_Secure_DualCore_End_Node_CM0PLUS**

Module	Description	Sum of read only [FLASH]	Sum of read/write [RAM]
Application	Core, application and target components	8945	684
HAL	STM32WL HAL and LL drivers	7346	88
IAR Lib	Proprietary IAR libraries	6539	132
IAR Startup	Int_vect, init routines, init table, CSTACK and HEAP	580	4096
MBMUX	Mailbox multiplexer wrappers and services	2361	528
Sigfox stack	Middleware Sigfox and libraries	15171	1214
SubGHz_Phy	Middleware radio interface	7457	413
Utilities	All STM32 services (sequencer, time server, low power mgr, trace, mem)	2795	972
Total		51194	8127

Table 13. **Memory footprint for Sigfox_Secure_DualCore_End_Node_CM4**

Module	Description	Sum of read only [FLASH]	Sum of read/write [RAM]
Application	Core, application and target components	3476	425
HAL	STM32WL HAL and LL drivers	13652	84
IAR Lib	Proprietary IAR libraries	1474	0
IAR Startup	Int_vect, init routines, init table, CSTACK and HEAP	797	4096
MBMUX	Mailbox multiplexer wrappers and services	2683	750
Utilities	All STM32 services (sequencer, time server, low power mgr, trace, mem)	2526	972

Module	Description	Sum of read only [FLASH]	Sum of read/write [RAM]
Total		24608	6327

6.2 SBSFU application

Values in the tables below are measured for the following configuration of the IAR Embedded Workbench compiler (EWARM version 8.30.1):

- optimization level 3 for size
- debug option off
- trace option off
- target: STM32WL55JC

Table 14. Memory footprint for SECoreBin

Module	Description	Sum of read only [FLASH]	Sum of read/write [RAM]
Application	Core, application and target components	830	4
HAL	STM32WL HAL and LL drivers	4510	76
IAR Lib	Proprietary IAR libraries	274	0
IAR Startup	Int_vect, init routines, init table, CSTACK and HEAP	218	0
KMS	Middleware key management services	21528	9276
SE	Middleware Secure Engine	1380	16
Total		28740	9372

Table 15. Memory footprint for SBSFU Cortex-M0+

Module	Description	Sum of read only [FLASH]	Sum of read/write [RAM]
Application	Core, application and target components	225	4
HAL	STM32WL HAL and LL drivers	6086	160
IAR Lib	Proprietary IAR libraries	6523	132
IAR Startup	Int_vect, init routines, init table, CSTACK and HEAP	560	6660
SBSFU	Secure Firmware Update and Secure boot	15235	3564
SE	Middleware Secure Engine	4576	1
Total		33205	10521

Note: The SBSFU Cortex-M0+ binary is about 60 Kbytes as it integrates the SECoreBin library from Table 12.

Table 16. Memory footprint for SBSFU Cortex-M4

Module	Description	Sum of read only [FLASH]	Sum of read/write [RAM]
Application	Core, application and target components	694	4
HAL	STM32WL HAL and LL drivers	4425	24
IAR Lib	Proprietary IAR libraries	120	0
IAR Startup	Int_vect, init routines, init table, CSTACK and HEAP	719	512
SBSFU	Secure Firmware Update and Secure boot	163	0
Total		6121	540

7 How to customize the memory mapping

7.1 Memory use versus memory allocation

There is a difference between the memory mapping/allocation mentioned in Table 8 and the memory footprint detailed by tables in Section 6 .

For example, Table 15 shows that SBSFU (including the SE interface) occupies 33.2 Kbytes while the memory allocation for these items is 33 + 5 = 38 Kbytes. Table 16 shows that the SBSFU Cortex-M4 occupies 6.1 Kbytes while the memory allocation for it is 10 Kbytes.

The main reasons for these differences are listed below:

- Space may be wasted in padding: some restrictions (for example given by the MPU) require boundaries of the blocks to be n-bytes aligned.
- The memory allocated to SBSFU let some margins for the following:
  - To have a common denominator mapping fitting for the different IDEs.
  - To avoid remappings for small changes.
  - To avoid remappings if the IDE version has changed and uses a bit more memory.
  - To enable some features (currently disabled) without having linker problem (such as tamper, or IWDG).

The remaining Flash memory is for the <RF_App> that includes the RF middleware, the RF drivers, the mailbox, the utilities, and the user application. The mapping provided with <Secure_RF_App> examples maximizes the free space for the Cortex-M4, considering the Cortex-M0+ as coprocessor.

The figure below shows that, on the Cortex-M4, about 75% of the space remains available for developing the application, while, on the Cortex-M0+, the remaining space is about 10%, depending on the IDE. The footprint given here is related to a specific release of the STM32CubeWL and can evolve.

Note: Similar principles apply to LoRaWAN and Sigfox examples.

Figure 34. LoRaWAN_SBSFU_1_Slot_DualCore Flash memory use vs allocation

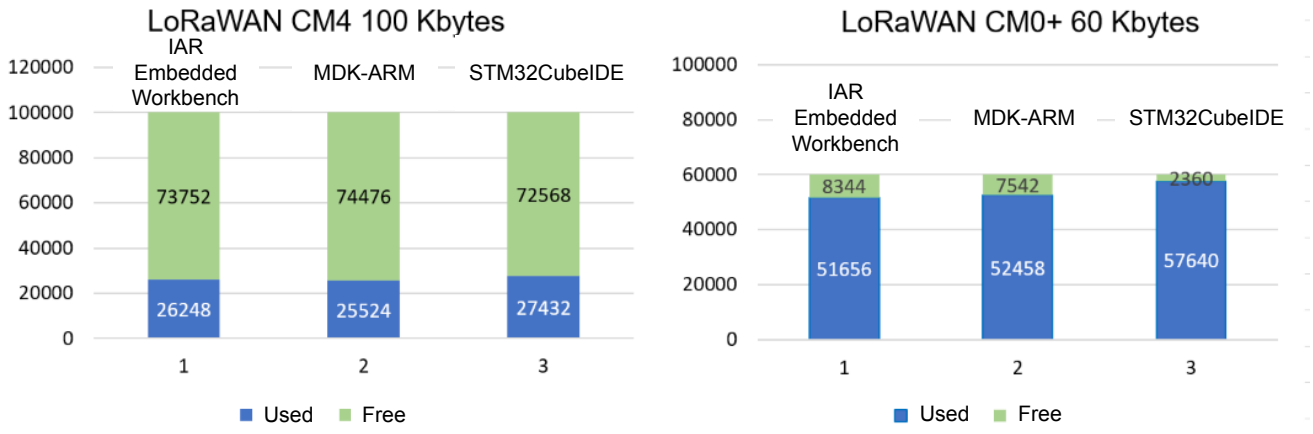
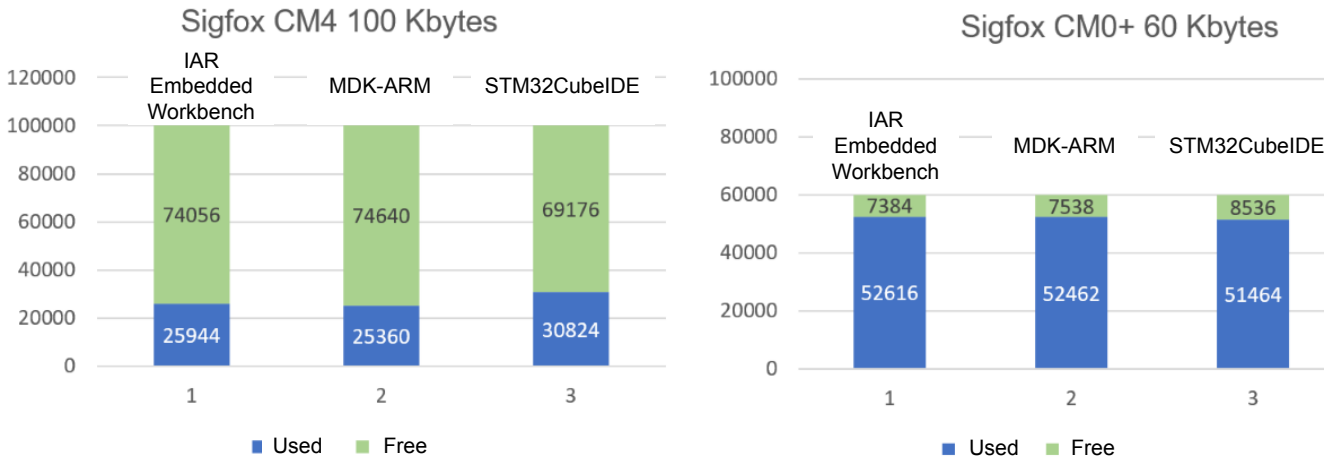


Figure 35. **Sigfox_SBSFU_1_Slot_DualCore Flash memory use vs allocation**



In LoRaWAN_SBSFU_1_Slot_DualCore, the middleware has been configured as follows:

- LoRaMAC Class A
- LoRaMAC region EU868 only
- optimization level 3 for size

Example 1: Memory space demanded by Class B and by RF regions

Class B occupies about 6 Kbytes, depending on the compiler. Each LoRa region needs between 2 and 3 Kbytes, depending on the region and the compiler.

The memory repartition may require modifications to enable class B or several regions simultaneously. For example, items that fit in the 8344 remaining space with IAR Embedded Workbench, and in the 7542 remaining space with MDK-ARM (as described in Figure 34), do not fit in the 2360-byte remaining space with the STM32CubeIDE (that uses the GCC compiler).

Table 17. **LoRaWAN_SBSFU_1_Slot_DualCore regions**

Region	IAR Embedded Workbench	MDK-ARM	STM32CubeIDE
Region.o	1764	2284	1600
RegionAS923.o	2792	2894	3188
RegionAU915.o	2940	2890	3328
RegionBaseUS.o	154	158	146
RegionCN470.o	2112	2128	2582
RegionCN779.o	2720	2822	3068
RegionCommon.o	1980	2008	2064
RegionEU433.o	2712	2802	3076
RegionEU868.o	2968	2936	3264
RegionIN865.o	2744	2880	3100
RegionKR920.o	2692	2732	3048
RegionRU864.o	2696	2790	3044
RegionUS915.o	2880	2870	3284

Example 2: Memory space needed to change the compiler optimization level

It may be also necessary to modify the memory repartition to optimize the speed vs the memory size, or simply to temporarily reduce the optimization level in order to debug the Cortex-M0+.

LoRaWAN_SBSFU_1_Slot CM0PLUS compiled with IAR Embedded Workbench in 'optimize-medium' needs about 56 Kbytes, instead of about 51 Kbytes in 'optimize-max'. Compiling in 'optimize-low' needs about 65.5 Kbytes, which does not fit in the 60 Kbytes available. 'optimize none' demands about 68.5 Kbytes.

The situation is similar for Sigfox_SBSFU_1_Slot CM0PLUS.

See the documents [3] and [4] for more details about the <RF_App> and how to configure/reduce it.

7.2 How to change the memory repartition between the cores

The main constraint when changing the memory repartition between Cortex-M4 and Cortex-M0+ is given by the MPU. At boot time, the SBSFU protects the Active 2 slot from access via the MPU. The related addresses have to be changed.

The secure memory boundaries (Cortex-M4 versus Cortex-M0+) are changed in the common linker file. For the IAR Embedded Workbench, the file is:

`<RF_App>_1_Slot_DualCore\Linker_Common\EWARM\mapping_fwimg.icf` and the concerned definitions are:

```
define exported symbol  __ICFEDIT_SLOT_Active_2_end__    = 0x0801BFFF;
define exported symbol  __ICFEDIT_SLOT_Active_1_start__  = 0x0801C000;
```

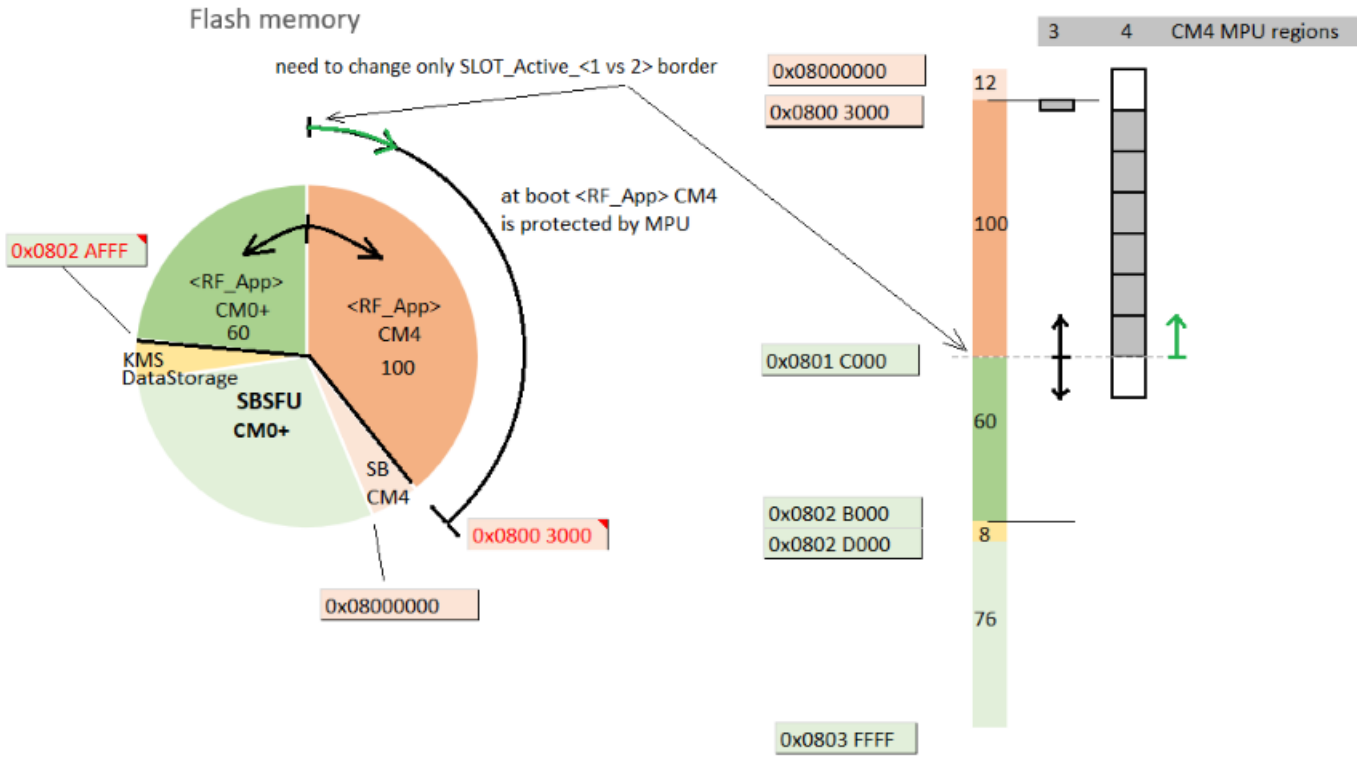
To change the MPU protection, the SBSFU code must be modified, specifically the following file:

`<RF_App>_SBSFU_1_Slot_DualCore\2_Images_SBSFU\CM4\Inc\sfu_low_level_security.h`

The SBSFU mapping is not impacted, but the <RF_App> mapping requires a change in the SBSFU configuration file.

This section explains how to change only the <RF_App> mapping , while maintaining the same total <RF_App>_CM4 + <RF_App>_CM0PLUS (from 0x0800 3000 to 0x0802 AFFF).

Figure 36. <RF_App> memory repartition allocation without impact on SBSFU



The Slot Active 2 area (<RF_App>_CM4) is protected by the following Cortex-M4 MPU regions:

- Cortex-M4 MPU region 3 (4 Kbytes, eight subregions of 0.5 bytes each)
- Cortex-M4 MPU region 4 (128 Kbytes with 96 Kbytes active, six subregions of 16 Kbytes each)

Remember:
A MPU region is composed of a start address, a size (power of two), and eight subregions that are enabled/disabled with an 8-bit mask in the corresponding SRD register (0: subregion enabled, 1: subregion disabled).

The Cortex-M4 MPU region 3 ranges from 0x0800 3000 to 0x0800 3FFF.

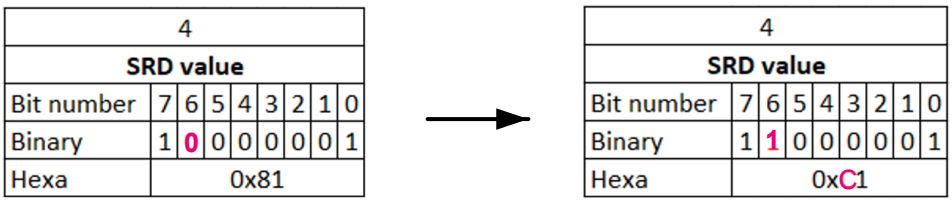
```
/**
  * @brief Region 3 - Forbid all access to the active slot.
  *                   From 0x08003000 ==> 0x08003FFF (4 kbytes)
  */
#define SFU_PROTECT_MPU_APP_FLASHEXE_RGNV  MPU_REGION_NUMBER3
#define SFU_PROTECT_MPU_APP_FLASHEXE_START SLOT_ACTIVE_2_START
#define SFU_PROTECT_MPU_APP_FLASHEXE_SIZE  MPU_REGION_SIZE_4KB
#define SFU_PROTECT_MPU_APP_FLASHEXE_SREG  0x00U  /*!< All subregions activated */
#define SFU_PROTECT_MPU_APP_FLASHEXE_PERM  MPU_REGION_NO_ACCESS
```

The Cortex-M4 MPU region 4 ranges from 0x0800 4000 to 0x0801 BFFF.

```
/**
  * @brief Region 4 - Forbid all access to the active slot.
  *            In addition to region 3, from 0x08004000 ==> 0x0801BFFF (96 kbytes)
  */
#define SFU_PROTECT_MPU_APP_FLASHEXE_ADJUST_RGNV  MPU_REGION_NUMBER4
#define SFU_PROTECT_MPU_APP_FLASHEXE_ADJUST_START FLASH_BASE
#define SFU_PROTECT_MPU_APP_FLASHEXE_ADJUST_SIZE  MPU_REGION_SIZE_128KB
#define SFU_PROTECT_MPU_APP_FLASHEXE_ADJUST_SREG  0x81U    /*!< 128 Kbytes / 8 * 6 ==> 96
Kbytes */
#define SFU_PROTECT_MPU_APP_FLASHEXE_PERM  MPU_REGION_NO_ACCESS
```

The easier change is to move 16 Kbytes from the Cortex-M4 to the Cortex-M0+ by disabling one subregion of the Cortex-M4 MPU region 4. The bit 7 of the SDR value is set from 0 to 1 to disable the sixth subregion, as shown in the figure below.

Figure 37. MPU region 4 - Changing subregion settings



The Cortex-M4 MPU region 4 is then reduced to 5 active subregions (16 Kbytes less) covering from 0x0800 4000 to 0x0801 7FFF.

```
/**
  * @brief Region 4 - Forbid all access to the active slot.
  *            In addition to region 3, from 0x08004000 ==> 0x08017FFF (80 kbytes)
  */
#define SFU_PROTECT_MPU_APP_FLASHEXE_ADJUST_RGNV  MPU_REGION_NUMBER4
#define SFU_PROTECT_MPU_APP_FLASHEXE_ADJUST_START FLASH_BASE
#define SFU_PROTECT_MPU_APP_FLASHEXE_ADJUST_SIZE  MPU_REGION_SIZE_128KB
#define SFU_PROTECT_MPU_APP_FLASHEXE_ADJUST_SREG  0xC1U    /*!< 128 Kbytes / 8 * 5
 ==> 80 Kbytes */
```

With this change, <RF_App>_CM4 is reduced to 84 Kbytes (4 + 80), ranging from 0x0800 3000 to 0x0801 7FFF. <RF_App>_CM0PLUS is augmented to 76 Kbytes (enough size to enable class B or activate RF regions, or to adapt the optimization mode, for example).

The linker file must be updated accordingly. For EWARM, `mapping_fwimg.icf` is modified as follows:

```
/* Active slot #2 (84 kbytes) */
define exported symbol   __ICFEDIT_SLOT_Active_2_header__= 0x0803F000;
define exported symbol   __ICFEDIT_SLOT_Active_2_start__ = 0x08003000;
define exported symbol   __ICFEDIT_SLOT_Active_2_end__   = 0x08017FFF;

/* Active slot #1 (76 kbytes) */
define exported symbol   __ICFEDIT_SLOT_Active_1_header__= 0x0803F800;
define exported symbol   __ICFEDIT_SLOT_Active_1_start__ = 0x08018000;
define exported symbol   __ICFEDIT_SLOT_Active_1_end__   = 0x0802AFFF;
```

If the Cortex-M0+ Flash memory must be increased further, the Cortex-M4 MPU region 4 can be reduced by 16 Kbytes more by setting the SDR to 0x**A**1U: <RF_App>_CM4 covers only 68 Kbytes (up to 0x0801 **3**FFF), and <RF_App>_CM0PLUS raises to 92 Kbytes (from 0x0801 **4**000).

Intermediate solutions are possible but require an additional MPU region with subregion size smaller than 16 Kbytes.

Remember:
The STM32WLxx MPU is divided in eight regions of definable size. Each MPU region includes eight subregions of equal size.

On STM32WL devices, both LoRaWAN_SBSFU_1_Slot and Sigfox_SBSFU_1_Slot use six MPU regions at boot. One of the two remaining MPU regions at boot can be used to define an additional MPU region. This refines the memory split between the cores.

Example:

Allocating 88 Kbytes to the Cortex-M4 and 7288 Kbytes to the Cortex-M0+ can be achieved by adding a new 4-Kbyte region to

`2_Images_SBSFU\CM4\Inc\sfu_low_level_security.h`:

```
/**
  * @brief Region 6 - Forbid all access to the active slot.
  *                   From 0x08018000 ==> 0x08018FFF (4 kbytes)
  */
#define SFU_PROTECT_MPU_APP_FLASHEXE_RGNV  MPU_REGION_NUMBER6
#define SFU_PROTECT_MPU_APP_FLASHEXE_START SLOT_ACTIVE_2_LAST
#define SFU_PROTECT_MPU_APP_FLASHEXE_SIZE  MPU_REGION_SIZE_4KB
#define SFU_PROTECT_MPU_APP_FLASHEXE_SREG  0x00U  /*!< All subregions activated */
#define SFU_PROTECT_MPU_APP_FLASHEXE_PERM  MPU_REGION_NO_ACCESS
```

The linker file must be adapted accordingly. For EWARM, `Linker_Common\EWARM\mapping_fwimg.icf` is modified as follows:

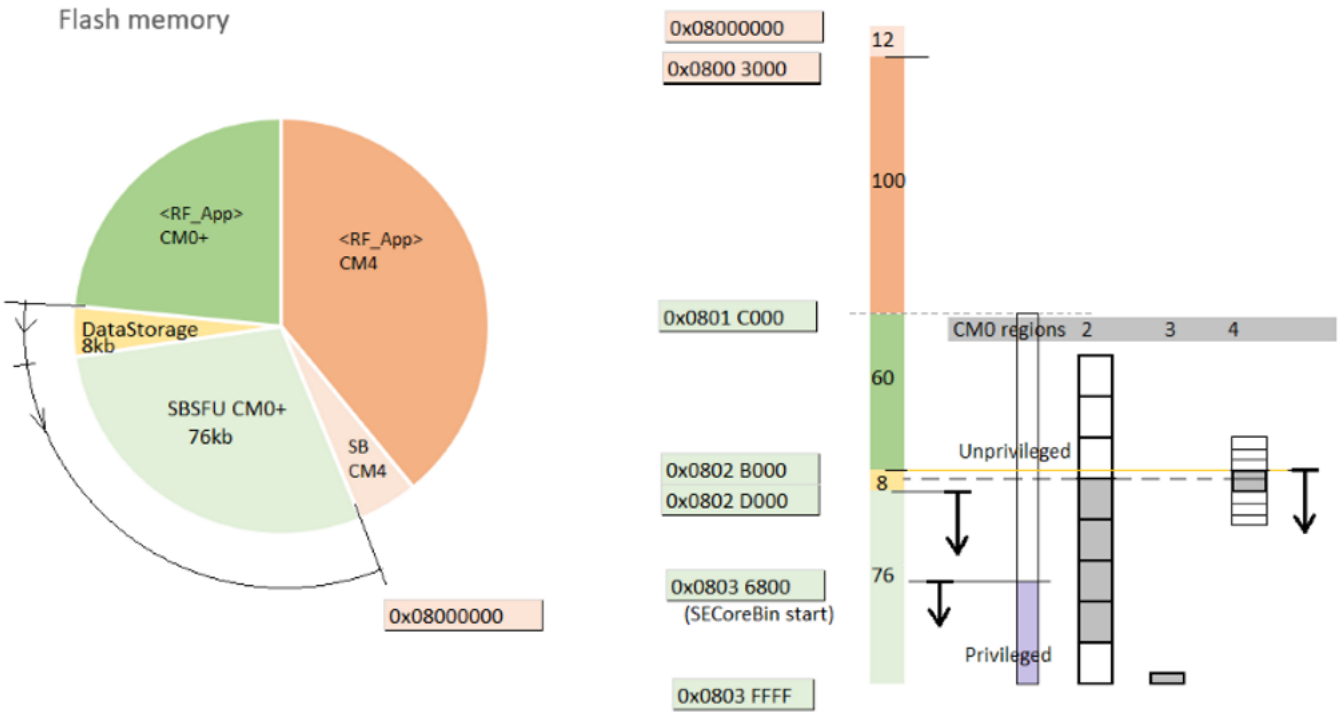
```
/* Active slot #2 (88 kbytes) */
define exported symbol   __ICFEDIT_SLOT_Active_2_header__= 0x0803F000;
define exported symbol   __ICFEDIT_SLOT_Active_2_start__ = 0x08003000;
define exported symbol   __ICFEDIT_SLOT_Active_2_end__   = 0x08018FFF;

/* Active slot #1 (72 kbytes) */
define exported symbol   __ICFEDIT_SLOT_Active_1_header__= 0x0803F800;
define exported symbol   __ICFEDIT_SLOT_Active_1_start__ = 0x08019000;
define exported symbol   __ICFEDIT_SLOT_Active_1_end__   = 0x0802AFFF;
```

7.3 How to reduce the SBSFU footprint and remap the memory accordingly

As explained in Section 7.1 , there is a mismatch between the SBSFU memory footprint and its memory allocation.

This section gives some guidelines to reduce further the SBSFU footprint, which means reducing the SBSFU feature to increase the slot active area (currently 100 + 60 = 160 Kbytes). The main objective is to remap the memory to reduce the SBSFU allocation.

Figure 38. SBSFU memory optimization



The achievable results depend on the IDE. Footprints are calculated with IAR Embedded Workbench in this section, but the principle can be extended to other compilers.

Remember:
Some constraints on memory alignment have to be taken into account when calculating the mapping (refer to the documents [1], [2], and [6]).

In order to optimize the number of MPU regions used, the examples provided in this section focus on reducing the SBSFU memory by blocks of at least 4 Kbytes or multiple of 4 Kbytes. This limits the changes to enable/disable only few subregions or to change the start address of some regions. There is no need to involve other MPU parameters and restrictions (such as adding regions, changing size, or alignment constraints).

The following modules can be reduced in the scope of the global SBSFU CM0+:

- NVM KMS Data Storage
- SBSFU CM0+
- SECoreBin

From the SBSFU footprint compiled with IAR Embedded Workbench version 8.30.1, the following features has been identified as good example candidates to be removed:

- trace and tamper features concerning the SBSFU area
- RSA in the SECoreBin area

Y-MODEM and MPU/GTZC can also be removed. Refer to section 'how to reduce SBSFU footprint' in document [1] for a general presentation . This section is a complement that shows in few examples which code lines can be modified.

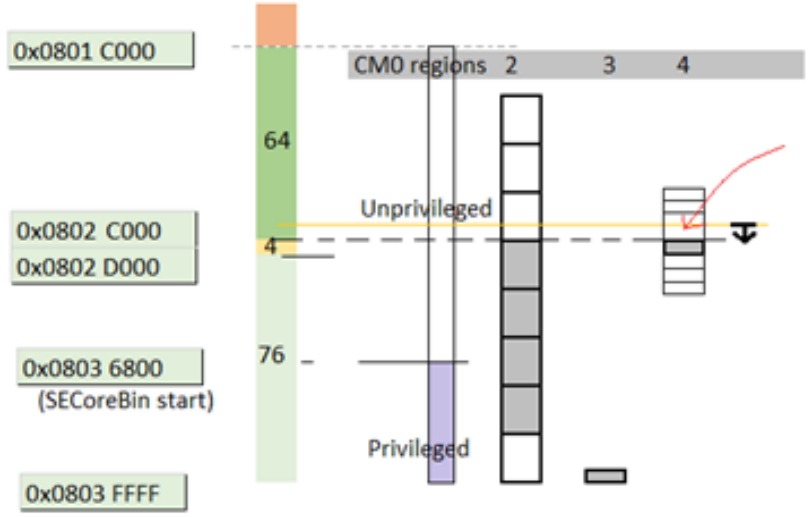
7.3.1 NVM KMS Data Storage

The NVM KMS Data Storage is the simplest to modify because it is positioned just after the slot active area, the <RF_App> area (see Table 8).

KMS_DataStorage in <Secure_RF_App> is 8 Kbytes and contains LoRaWAN or Sigfox derived keys. This can be reduced to 4 Kbytes without any impact on the functionality of current projects. A larger place than necessary has been provided for derived key because it implies longer “estimated live time” for the memory. The KMS Data Storage is filled sequentially each time a key is derived. Once a page reaches its end, the full page is erased, and keys are written in a FLOP page. The expected memory lifetime is 10.000 page erase. Sigfox derives one key each time it sends data. LoRaWAN derives keys only at the JOIN procedure, but the number of keys depends on the configuration (such as OTAA/APB or multicast). See the documents [3] and [4] for a deeper explanation with precise examples about key size, or number of requested keys.

The NVM is organized by 2-Kbyte pages. Due to the double buffering (flip/flop EEPROM emulation mechanism), each page needs a 'twin'. The minimum to be allocated for KMS_DataStorage is then 4 Kbytes. Reducing KMS Data Storage from 8 Kbytes to 4 Kbytes can be simply done by changing the correspondent MPU execution region.

Figure 39. KMS_DataStorage reduction



These are the configurations to be changed in

`2_Images_SBSFU\CM0PLUS\SBSFU\Target\sfu_low_level_security.h`:

```
/**
  * @brief Region 4 - Enable the rw operation in privileged mode for KMS_DataStorage
  *                   Execution capability disabled
  *                   Inner region inside the Region 0
  */
#define SFU_PROTECT_MPU_KMS_RGNV  MPU_REGION_NUMBER4
#define SFU_PROTECT_MPU_KMS_START 0x08028000UL
#define SFU_PROTECT_MPU_KMS_SREG  0xE8U    /*!< 32 Kbytes / 8 * 1 ==> 4 Kbytes */
#define SFU_PROTECT_MPU_KMS_SIZE  MPU_REGION_SIZE_32KB
#define SFU_PROTECT_MPU_KMS_PERM  MPU_REGION_PRIV_RW
#define SFU_PROTECT_MPU_KMS_EXECV MPU_INSTRUCTION_ACCESS_DISABLE
```

The linker file example below is given for EWARM, but the same principle applies for other IDEs.

Update in `Linker_Common\EWARM\mapping_sbsfu.icf`:

```
/* 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__       = 0x0802C000;
define exported symbol __ICFEDIT_KMS_DataStorage_end__         = 0x0802CFFF;
```

Slot Active 1 is then increased from 60 Kbytes to 64 Kbytes in `Linker_Common\EWARM\mapping_fwimg.icf`:

```
/* Active slot #1 (64 kbytes) */
define exported symbol  __ICFEDIT_SLOT_Active_1_header__= 0x0803F800;
define exported symbol  __ICFEDIT_SLOT_Active_1_start__ = 0x0801C000;
define exported symbol  __ICFEDIT_SLOT_Active_1_end__   = 0x0802BFFF;
```

7.3.2 Trace and tamper

Trace and tamper are two different features that belong to the SBSFU CM0+ memory area. The SBSFU CM0+ footprint can be reduced by disabling the definition `SFU_DEBUG_MODE` in

`2_Images_SBSFU\CM0PLUS\SBSFU\App\app_sfu.h`:

```
/* #define SFU_DEBUG_MODE         */
```

The SBSFU CM0+ footprint is then reduced of around 13 Kbytes (depending on the compiler), but there are no more logs printed on the terminal during the SBSFU execution.

The size of the SBSFU is now reduced but all the code placed above must be shifted down to give more space to the application.

```
/* 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__    = 0x0802xxxx;
define exported symbol __ICFEDIT_KMS_DataStorage_end__      = 0x0802xxxx;

/* SE IF ROM: used to locate Secure Engine interface code out of MPU isolation */
Define exported symbol __SE_IF_region_ROM_start__ = __ KMS_DataStorage_end__ + 1;
Define exported symbol __SE_IF_region_ROM_end__ = __SE_IF_region_ROM_start__ + 0x13FF;

/* SBSFU Code region */
define exported symbol __SB_region_ROM_start__ = __ SE_IF_region_ROM_end__ + 1;
define exported symbol __ICFEDIT_SB_region_ROM_end__ = 0x080367FF;
```

`__SB_region_ROM_start__` can start lower, `SE_IF` is shifted lower, and the `KMS_DataStorage` as well. The MPU of the KMS, but also the MPU that protects the SBSFU has to be shifted.

In the STM32CubeWL example, the shift down makes sense if the region size is multiple the 4 Kbytes (even better 16 Kbytes), otherwise MPU regions may be not sufficient (taking into account the constraints when changing the MPU).

Disabling the trace reduces of 13 Kbytes (for IAR Embedded Workbench version 8.30.1). The mapping can be modified easily to gain 12 Kbytes for the application. To gain 16 Kbytes, the user must find other 3 Kbytes of available memory.

In the RF STM32CubeWL examples, the allocation have been made to assign the same mapping between different compilers and to respect alignment constraints. Some memory parts remain then unused (depending on the compiler) and can be exploited to reach the 16 Kbytes.

Some empty space may be reserved to allow users to enable features that are disabled by default in the RF STM32CubeWL examples. This is the case for the tamper feature: often disabled during development (because Nucleo boards are sensitive: strong movements can be interpreted as hardware attacks) but useful during production. The tamper code uses 3,2 Kbytes (for IAR Embedded Workbench 8.30.1). If this space is used to reach the 16 Kbytes, the tamper cannot be enabled in production.

The figures below show two solutions that do not need additional MPU regions, where `KMS_DataStorage` is sized 4 Kbytes.

Figure 40. Example to reduce SBSFU by 12 Kbytes

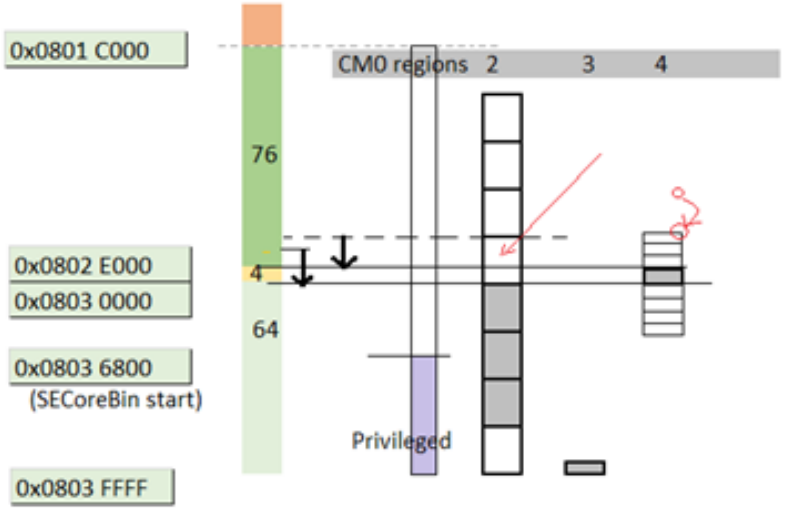
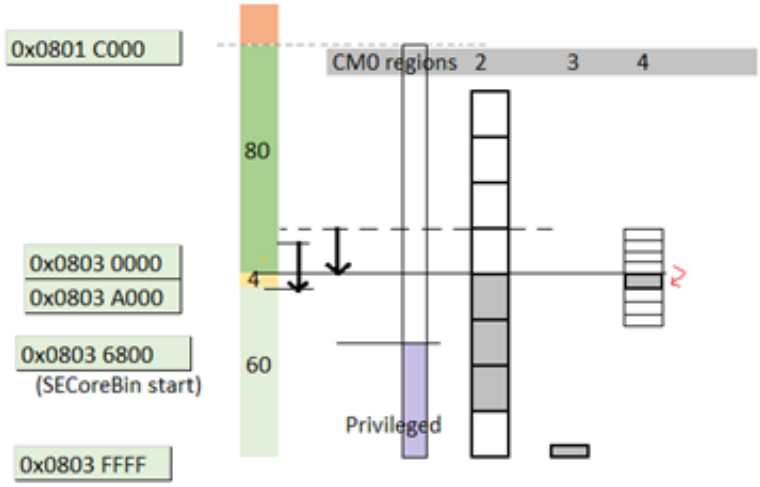


Figure 41. Example to reduce SBSFU by a 16-Kbyte subregion



7.3.3 RSA/ SECoreBin

The <Secure_RF_App> projects do not use RSA, which saves 2 Kbytes in the SECoreBin privileged area (see documents [1] and [2] for details about privileged/unprivileged memory settings). Changing the memory mapping for 2 Kbytes is complex and not described here. But some memory remain unused in the provided RF STM32CubeWL examples. Depending on the compiler, the user can check if SECoreBin can be reduced by 4 Kbytes, or how to combine with the SBSFU alignment in order to limit the impact on the MPU configuration. Combinations are multiples.

The Secure Engine is about the bottom of the memory mapping.

Changing `__ICFEDIT_SE_Code_region_ROM_start__` means changing all the above mappings up to `Slot Active region`:

- changing related Cortex-M0+ MPU (regions 2 and 4) as for the above SBSFU
- changing the privileged boundaries, as shown in Section 7.3

These changes can be done by repeating the principle already detailed previously.

7.3.4 Summary

The table below lists the SBSFU features that can be removed/disabled to increase the user application memory. Further gain can be obtained by reducing memory loss due to alignment constraint, or by tailoring the memory mapping for a specific IDE.

Table 18. SBSFU code size reduction

Option	Description	Gain
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 'import blob' feature is also disabled (no ECDSA, no RSA)
Disable SFU_DEBUG_MODE.	No more information displayed on the terminal during the SBSFU execution	~ 13 Kbytes
Remove Y-MODEM, UART	No more possible to update the firmware (make sense to remove UART only if SFU_DEBUG_MODE is disabled.)	~ 3 Kbytes
Remove SE internal isolation based on MPU/GTZC (only when all STM32 code is fully trusted and robust).	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, or short memory expected life-time.	4 Kbytes
Configure the system clock with LL interface.	The code is a bit more complex and the tamper must not be used as the removed HAL dependencies are restored.	~ 2 Kbytes

The relation between footprint and allocation gain is not always proportional. Reducing the footprint ~1,5 Kbytes can lead to zero gain in allocation, or to 4-Kbyte gain in allocation.

Reallocating boundaries for less than 4 Kbytes is technically possible but requires reworking the full MPU mapping provided in the STM32CubeWL (eight maximum regions allowed for each STM32WL5x core, Flash memory and RAM included) . The examples in this application note avoid changing MPU region sizes, and avoid adding MPU regions.

Remapping can reduce the Flash memory efficiently when tailored to a specific IDE compiler.

Note: The memory mapping may change with each STM32CubeWL revision .

Revision history

Table 19. **Document revision history**

Date	Version	Changes
26-July-2021	1	Initial release.

Contents

List of tables

List of figures