UM2767

User manual

Getting started with the SBSFU of STM32CubeWL

Introduction

This user manual describes how to get started with the STM32CubeWL SBSFU (Secure Boot and Secure Firmware Update).

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

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

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

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

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

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

Regarding LoRaWAN® firmware update over-the-air with STM32CubeWL, more details are provided in application note *LoRaWAN® firmware update over the air with STM32CubeWL* (AN5554).

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

For a complex system (such as protocol stack, middleware and user application), the firmware image configuration can be extended up to three firmware images. In this application, one firmware image is used for the single-core configuration while two firmware images are available for the dual-core configuration. Examples can be configured to use asymmetric or symmetric cryptographic schemes with or without firmware encryption.

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

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

1 General information

The STM32CubeWL SBSFU comes with examples running on the STM32WL Series.

STM32CubeWL SBSFU is provided as a reference code for standalone STM32 system solution examples demonstrating the best use of STM32 protections to protect assets against unauthorized external and internal access.

STM32CubeWL SBSFU is a starting point for OEMs to develop their SBSFU as a function of their product security requirement levels.

The STM32CubeWL Secure Boot and Secure Firmware Update run on STM32 32-bit microcontrollers based on the Arm® Cortex®-M processor.

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

1.1 Terms and definitions

The following table presents the definition of acronyms that are relevant for a better understanding of this document.

Table 1. List of acronyms

Acronym	Description
AAD	Additional authenticated data
AES	Advanced encryption standard
BFU	Boot and Firmware Update
CBC	AES cipher block chaining
DMA	Direct memory access
DSA	Digital signature algorithm
ECC	Elliptic curve cryptography
ECCN	Export control classification number
ECDSA	Elliptic curve digital signature algorithm
FSM	Finite-state machine
GCM	AES Galois/counter mode
GTZC	Global security controller
GUI	Graphical user interface
HAL	Hardware abstraction layer
HDP	Hide protection
IDE	Integrated development environment
IV	Initialization vector
IWDG	Independent watchdog
FW	Firmware
KMS	Key management services
MAC	Message authentication code
MCU	Microcontroller unit
MPU	Memory protection unit
NONCE	Number used only once
PEM	Privacy enhanced mail
RDP	Readout protection
SB	Secure Boot
SE	Secure Engine
SBSFU	Secure Boot and Secure Firmware Update
SFU	Secure Firmware Update
SM	State machine
TZIC	Security Illegal access controller
TZSC	Security controller
UART	Universal asynchronous receiver/transmitter
UUID	Universally unique identifier
WRP	Write protection

The following table presents the definition of terms that are relevant for a better understanding of this document.

Table 2. **List of terms**

Term	Description
Firmware image	A binary image (executable) run by the device (can be a user application or a protocol stack).
Firmware header	Bundle of meta-data describing the firmware image to be installed. It contains firmware information and cryptographic information.
mbed-crypto	Mbed™ implementation of the cryptographic algorithms.
sfb file	Binary file packing the firmware header and the firmware image.

1.2 References

STMicroelectronics related documents

Public documents are available online from the STMicroelectronics website at www.st.com. Contact STMicroelectronics when more information is needed.

- Application note *Integration guide of SBSFU on STM32CubeWL (including KMS)* (AN5544)
- Application note *Introduction to STM32 microcontrollers security* (AN5156)
- Application note *LoRaWAN® firmware update over the air with STM32CubeWL* (AN5554)
- User manual *STM32CubeProgrammer software description* (UM2237)

Other documents

- *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40 Plus Errata*

2 STM32Cube overview

What is STM32Cube

STM32Cube is an STMicroelectronics original initiative to significantly improve designer's productivity by reducing development effort, time, and cost. STM32Cube covers the whole STM32 portfolio.

STM32Cube includes:

- A set of user-friendly software development tools to cover project development from conception to realization, among which are:
  - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards
  - STM32CubeIDE, an all-in-one development tool with peripheral configuration, code generation, code compilation, and debug features
  - STM32CubeProgrammer (STM32CubeProg), a programming tool available in graphical and command-line versions
  - STM32CubeMonitor (STM32CubeMonitor, STM32CubeMonPwr, STM32CubeMonRF, STM32CubeMonUCPD) powerful monitoring tools to fine-tune the behavior and performance of STM32 applications in real-time
- STM32Cube MCU and MPU Packages, comprehensive embedded-software platforms specific to each microcontroller and microprocessor series (such as STM32CubeWL for the STM32WL Series), which include:
  - STM32Cube hardware abstraction layer (HAL), ensuring maximized portability across the STM32 portfolio
  - STM32Cube low-layer APIs, ensuring the best performance and footprints with a high degree of user control over hardware
  - A consistent set of middleware components such as FAT file system (FatFS), RTOS, USB Host and Device, TCP/IP, Touch library, and Graphics
  - All embedded software utilities with full sets of peripheral and applicative examples
- STM32Cube Expansion Packages, which contain embedded software components that complement the functionalities of the STM32Cube MCU and MPU Packages with:
  - Middleware extensions and applicative layers
  - Examples running on some specific STMicroelectronics development boards

How does this software integrate in STM32Cube?

The proposed software is based on the STM32Cube HAL, the hardware abstraction layer for the STM32 microcontroller. In addition, STM32CubeWL provides middleware components:

- Secure Engine for managing all critical data and operations, such as cryptography operations accessing firmware encryption key and others
- Key management services offering cryptographic services via PKCS#11 APIs

The package includes different sample applications to provide a complete SBSFU solution:

- SE_CoreBin application: provides a binary including all the ‘trusted’ code.
- Cortex®-M4 Secure Boot (SB) application:
  - Secure Boot (Root of Trust)
  - Local download via UART Virtual COM port in single-core configurations and in the dual-core dual-slot configuration
- Cortex®-M0+Secure Boot and Secure Firmware Update (SBSFU) application:
  - Secure Boot (Root of Trust)
  - FW installation management
  - Local download via UART Virtual COM port in the dual-core single-slot configuration

- Cortex®-M0+ user application:
  - Downloads a new firmware in the dual-slot mode of operation
  - Provides examples of testing protection mechanisms
  - Provides examples using KMS APIs
- Cortex®-M4 user application: frame for the user's unsecured code

The sample applications are delivered in dual-slot and single-slot modes of operation and can be configured in different cryptographic schemes.

Note: The single-slot configuration is demonstrated in the examples named BFU_1_Slot (single-core configuration) and SBSFU_1_Slot_DualCore.

Note: The dual-slot configuration is demonstrated in the examples named BFU_2_Slots (single-core configuration) and SBSFU_2_Slots_DualCore.

The examples BFU_1_Slot and BFU_2_Slots are named BFU because they run on the non-secure Cortex®-M4 and do not integrate the main security features like secure mode, secure memory, GTZC and HDP. In this configuration, only attack-surface reduction is ensured. The sample applications are then:

- SE_CoreBin application
- Boot and Firmware Upgrade (BFU) application
- User application

This user manual describes the typical use of the package:

- Based on the NUCLEO-WL55JC board
- With sample applications operating in dual-slot mode and configured with asymmetric authentication and symmetric firmware encryption

More information about the configuration options and the single-slot mode of operation are provided in the appendices of this document.

3 Secure Boot and Secure Firmware Update (SBSFU)

This section provides details on the product security introduction, the Secure Boot, the Secure Firmware Update and the cryptography operations on the SBSFU environment.

3.1 Product security introduction

A device deployed in the field operates in an untrusted environment and it is therefore subject to threats and attacks. To mitigate the risk of attack, the goal is to allow only authentic firmware to run on the device. Allowing the update of firmware images to fix bugs, or introduce new features or countermeasures, is commonplace for connected devices, but it is prone to attacks if not executed securely.

Consequences may be damaging such as firmware cloning, malicious software download, or device corruption. Security solutions have to be designed to protect sensitive data (potentially even the firmware itself) and critical operations.

Typical countermeasures are based on cryptography (with an associated secret key) and memory protections:

- Cryptography ensures integrity (the assurance that data has not been corrupted), authentication (the assurance that a certain entity is who it claims to be) and confidentiality (the assurance that only authorized users can read sensitive data) during firmware transfer.
- Memory protection mechanisms prevent external attacks (for example by accessing the device physically through JTAG) and internal attacks from other embedded processes.

The following chapters describe solutions implementing confidentiality, integrity, and authentication services to address the most common threats for an IoT end-node device.

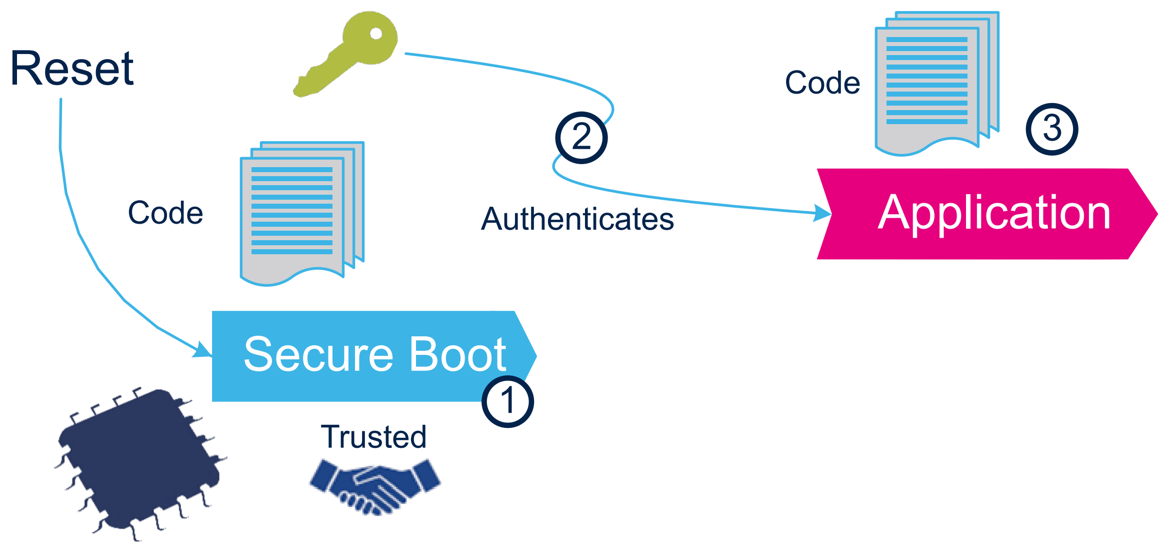
3.2 Secure Boot

Secure Boot (SB) asserts the integrity and authenticity of the Cortex®-M4 and Cortex®-M0+ user application images that are executed: cryptographic checks are used to prevent any unauthorized or maliciously modified software from running. The Secure Boot process implements a Root of Trust (refer to the following figure): starting from this trusted component (1), every other component is authenticated (2) before its execution (3).

The **integrity** is verified to be sure that the image that is going to be executed has not been corrupted or maliciously modified.

An **authenticity** check aims to verify that the firmware image is coming from a trusted and known source to prevent unauthorized entities to install and execute code.

Figure 1. **Secure Boot Root of Trust**



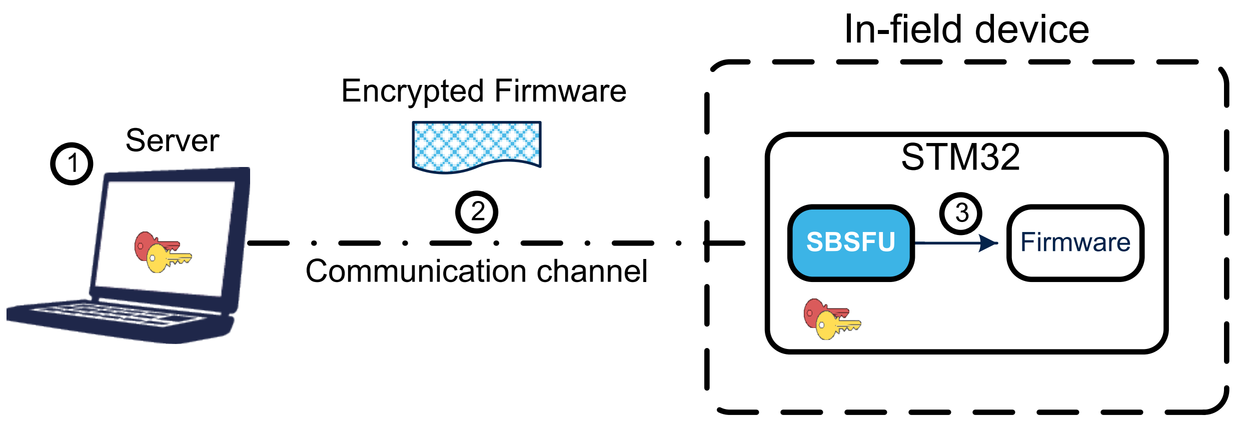
3.3 Secure Firmware Update

Secure Firmware Update (SFU) provides a secure implementation of in-field firmware updates, enabling the download of new firmware images to a device in a secure way.

As shown in the following figure, two entities are typically involved in a firmware update process:

- Server
  - OEM manufacturer server/web service
  - Stores the new version of device firmware
  - Communicates with the device and sends the new image version in an encrypted form if it is available
- Device
  - Deployed in the field
  - Embeds a code running firmware update process
  - Communicates with the server and receives a new firmware image
  - Authenticates, decrypts, and installs the new firmware images then executes it

Figure 2. **Typical in-field device update scenario**



Firmware update runs through the following steps:

1. If a firmware update is needed, a new encrypted firmware image is created and stored in the server.
2. The new encrypted firmware image is sent to the device deployed in the field through an untrusted channel.
3. The new image is downloaded, checked, and installed.

The firmware update can be done on the complete firmware image, or only on a portion of the firmware image (only for dual-slot configuration).

The firmware update is vulnerable to the threats presented in Section 3.1 Product security introduction: cryptography is used to ensure confidentiality, integrity, and authentication.

The **confidentiality** is implemented to protect the firmware image, which may be a key asset for the manufacturer. The firmware image sent over the untrusted channel is encrypted so that only devices having access to the encryption key can decrypt the firmware package.

The **integrity** is verified to be sure that the received image is not corrupted.

The **authenticity** check aims to verify that the firmware image is coming from a trusted and known source, to prevent unauthorized entities to install and execute code.

3.4 Cryptography operations

The STM32CubeWL SBSFU is delivered with three cryptographic schemes using both asymmetric and symmetric cryptography.

The default cryptographic scheme demonstrates ECDSA asymmetric cryptography for firmware verification and AES-CBC symmetric cryptography for firmware decryption. Thanks to asymmetric cryptography, the firmware verification can be performed with public key operations so that no secret information is required in the device.

The alternative cryptographic schemes provided in the STM32CubeWL SBSFU are:

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

The following table presents the various security features associated with each of the cryptographic schemes.

Table 3. Cryptographic scheme comparison

Features	Asymmetric with AES encryption	Asymmetric without encryption	Symmetric (AES-GCM)[1]
Confidentiality	AES-CBC encryption (FW binary)	No, the user FW is in a clear format.	AES-GCM encryption (FW binary)
Integrity	SHA256 (FW header and FW binary)		AES-GCM Tag (FW header and FW binary)
Authentication	• SHA256 of the FW header is ECDSA signed • SHA256 of the FW binary stored in FW header		
Cryptographic keys in device	Private AES-CBC key (secret) Public ECDSA key	Public ECDSA key	Private AES-GCM key (secret)

1. *For the symmetric cryptographic scheme, it is highly recommended to configure a unique symmetric key for each product.*

4 Key management services

Key management services (KMS) middleware provides cryptographic services through the standard PKCS#11 APIs (specified by OASIS) allowing to abstract the key value to the caller (using object ID and not directly the key-value). In the dual-core configuration, KMS is executed inside a protected/isolated environment to ensure that key value cannot be accessed by an unauthorized code running outside the protected/isolated environment. In the single-core configuration, the same services are offered but they are not executed inside a protected and isolated environment.

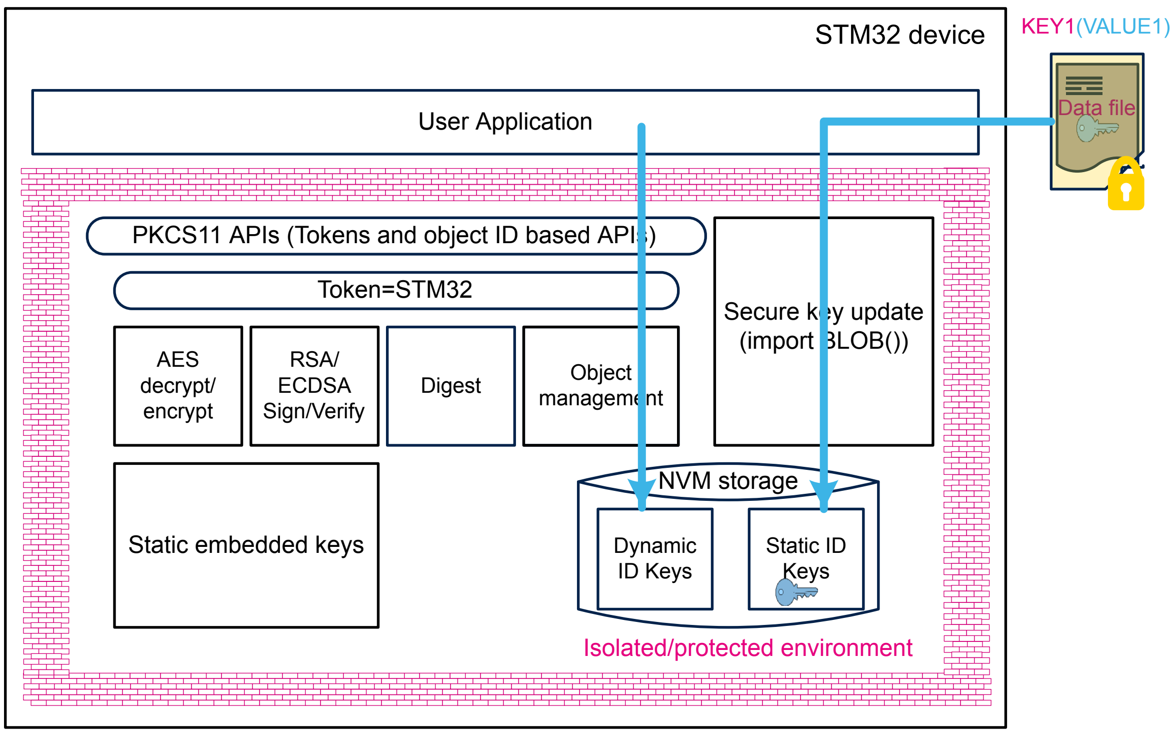
KMS only supports a subset of PKCS#11 APIs:

- Object management functions: creation / update / deletion
- AES encryption functions
- AES decryption functions
- Digesting functions
- RSA and ECDSA signing/verifying functions
- Key management functions: key generation/derivation

KMS manages three types of keys:

- Static embedded keys:
  - Predefined keys are embedded within the code. Such keys cannot be modified.
- Updatable keys with static ID:
  - Keys IDs are predefined in the system
  - The key value can be updated in an NVM storage via a secure procedure using static embedded root keys (authenticity check, data integrity check, and data decryption)
  - Key cannot be deleted
- Updatable keys with dynamic ID:
  - Key IDs are defined when creating the keys
  - The key value is created using internal functions. Typically, the `DeriveKey()` function creates dynamic objects
  - Key can be deleted

Figure 3. **KMS functions overview**



For more details regarding the OASIS PKCS#11 standard, refer to document *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40 Plus Errata*

5 Protection measures and security strategy

Cryptography ensures integrity, authentication, and confidentiality. However, the use of cryptography alone is not enough: a set of measures and system-level strategies are needed for protecting critical operations and sensitive data (such as a secret key), and the execution flow, to resist possible attacks.

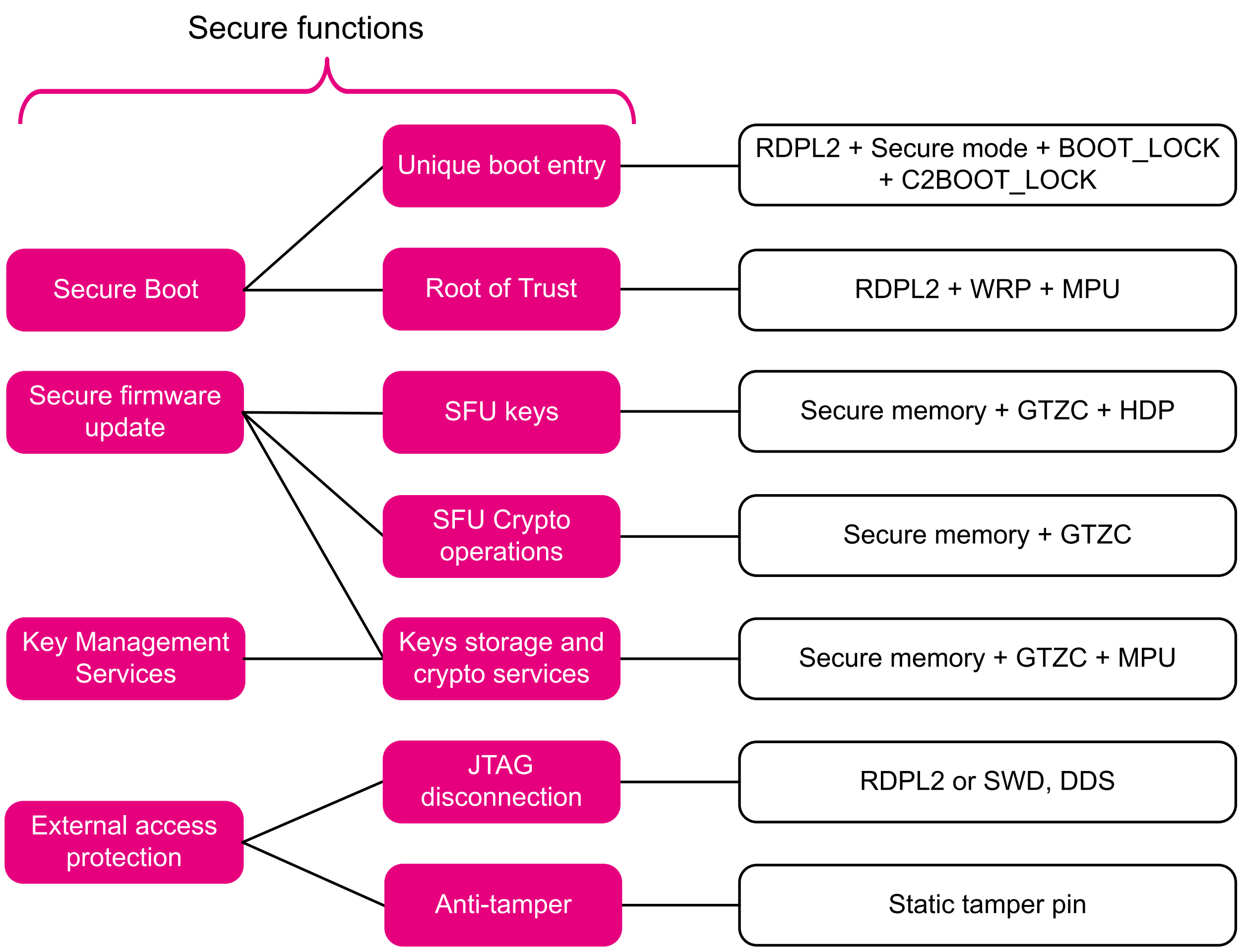
Secure software coding techniques such as doubling critical tests, doubling critical actions, checking parameters values, and testing a flow control mechanism, are implemented to resist basic fault-injection attacks.

The security strategy is based on the following concepts:

- Ensure single entry point at reset: force code execution to start with Cortex®-M4 Secure Boot code
- Make Cortex®-M4 Secure Boot, Cortex®-M0+ SBSFU code and Cortex®-M0+ SBSFU secrets immutable: no possibility to modify or alter them once security is fully activated
- Create a protected enclave isolated from SBSFU application and user applications to store secrets such as keys, and to run critical operations such as cryptographic algorithms
- Limit surface execution to SBSFU code during SBSFU application execution
- Remove JTAG access to the device
- Monitor the system: intrusion detection and SBSFU execution time

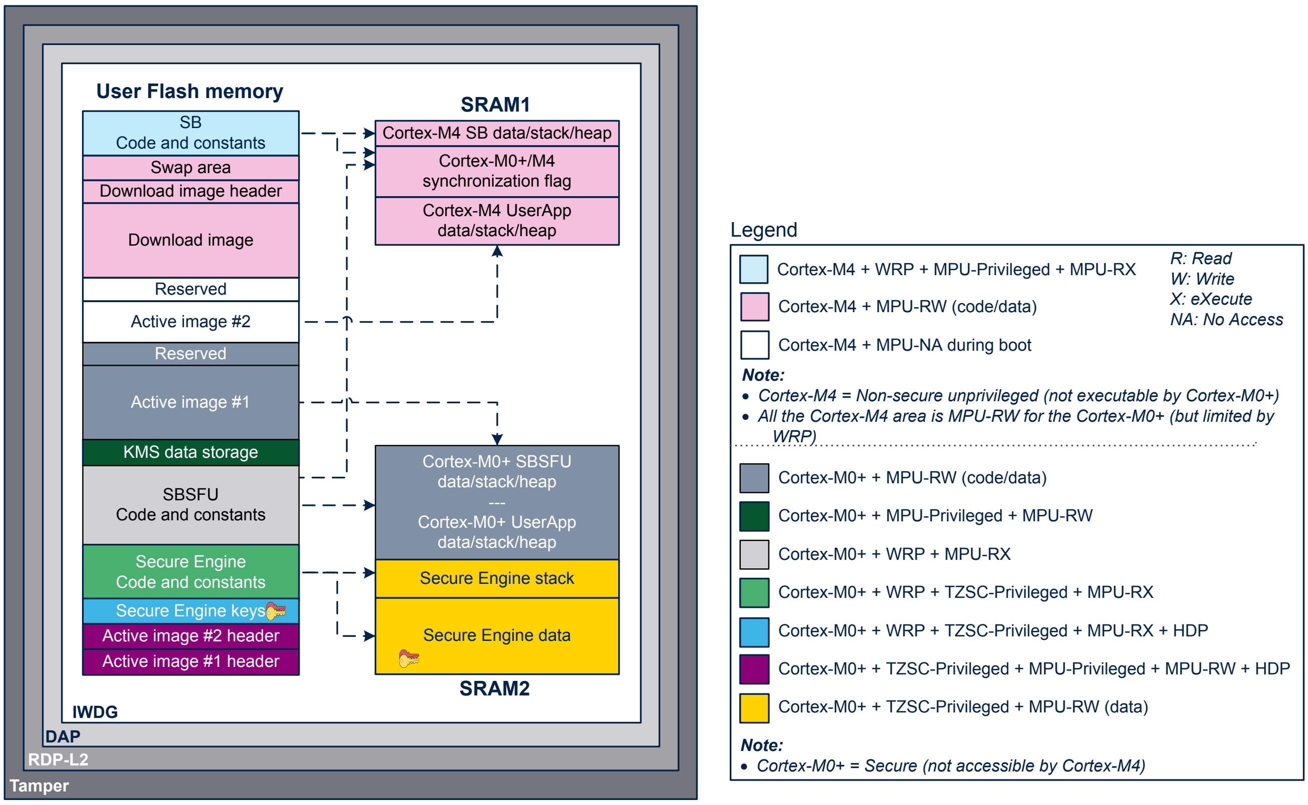
The following figure gives a high-level view of the security mechanisms activated on STM32WL Series.

Figure 4. **SBSFU security on STM32WL Series**



The figure below illustrates how the system, the code, and the data are protected in the dual-core dual-slot STM32CubeWL SBSFU application example.

Figure 5. STM32WL protection overview during SBSFU execution


User Flash memory
SB Code and constants
Swap area
Download image header
Download image
Reserved
Active image #2
Reserved
Active image #1
KMS data storage
SBSFU Code and constants
Secure Engine Code and constants
Secure Engine keys
Active image #2 header
Active image #1 header
SRAM1
Cortex-M4 SB data/stack/heap
Cortex-M0+/M4 synchronization flag
Cortex-M4 UserApp data/stack/heap
Cortex-M0+ SBSFU data/stack/heap --- Cortex-M0+ UserApp data/stack/heap
Secure Engine stack
Secure Engine data
SRAM2
IWDG
DAP
RDP-L2
Tamper
Legend
Cortex-M4 + WRP + MPU-Privileged + MPU-RX
Cortex-M4 + MPU-RW (code/data)
Cortex-M4 + MPU-NA during boot
R: Read
W: Write
X: eXecute
NA: No Access
Note:
Cortex-M4 = Non-secure unprivileged (not executable by Cortex-M0+)
All the Cortex-M4 area is MPU-RW for the Cortex-M0+ (but limited by WRP)
Cortex-M0+ + MPU-RW (code/data)
Cortex-M0+ + MPU-Privileged + MPU-RW
Cortex-M0+ + WRP + MPU-RX
Cortex-M0+ + WRP + TZSC-Privileged + MPU-RX
Cortex-M0+ + WRP + TZSC-Privileged + MPU-RX + HDP
Cortex-M0+ + TZSC-Privileged + MPU-Privileged + MPU-RW + HDP
Cortex-M0+ + TZSC-Privileged + MPU-RW (data)
Note:
Cortex-M0+ = Secure (not accessible by Cortex-M4)

Protections against outer attacks

Outer attacks refer to attacks triggered by external tools such as debuggers or probes, trying to access the device. In the SBSFU application examples, RDP, tamper, DAP, and IWDG protections are used to protect the product against outer attacks:

- **RDP** (read protection): read protection level 2 is mandatory to achieve the highest level of protection and to implement a Root of Trust:
  - External access via the JTAG hardware interface to RAM and Flash memory is forbidden. This prevents attacks aiming to change SBSFU code and therefore mining the Root of Trust.
  - Option bytes can only be changed by secure Cortex®-M0+ (use case not supported by STM32CubeWL SBSFU package).

Note: RDP level 1 is not proposed for the following reasons:

- *Secure Boot / Root of Trust (single entry point and immutable code) cannot be ensured, because option bytes (WRP, BOOT_LOCK, C2BOOT_LOCK) can be modified in RDP L1 by non-secure Cortex®-M4 or with tools like STM32CubeProgrammer.*
- *Device internal Flash memory can be fully reprogrammed (after Flash memory mass erase via RDP L0 regression) with a new FW without any security.*
- *Secrets in RAM protected by secure mode and GTZC can be accessed by attaching the debugger via the JTAG hardware interface on a system reset.*

In case JTAG hardware interface access is not possible at customer product, and in case the customer uses a trusted and reliable user application code, then the above highlighted risks are not valid.

Note: Details about the configuration of RDP level 2 (production mode configuration) are provided in application note Integration guide of SBSFU on STM32CubeWL (including KMS) (AN5544).

- **Tamper**: the anti-tamper protection is used to detect physical tampering actions on the device and to take related countermeasures. In the case of tampering detection, the SBSFU application example forces a reboot.
- **DAP** (debug access port): the DAP protection consists of de-activating the DAP (debug access port). Once de-activated, JTAG pins are no longer connected to the STM32 device internal bus. DAP is automatically disabled with RDP Level 2.
- **IWDG** (independent watchdog): IWDG is a free-running down-counter. Once running, it cannot be stopped. It must be refreshed periodically before it causes a reset. This mechanism allows the control of SBSFU execution duration.

Protections against inner attacks

Inner attacks refer to attacks triggered by code running in the STM32 device. Attacks may be due to either malicious firmware exploiting bugs or security breaches, or unwanted operations. In the SBSFU application examples, secure mode, WRP, GTZC (TZSC and TZIC), HDP, and MPU protections preserve the product from inner attacks:

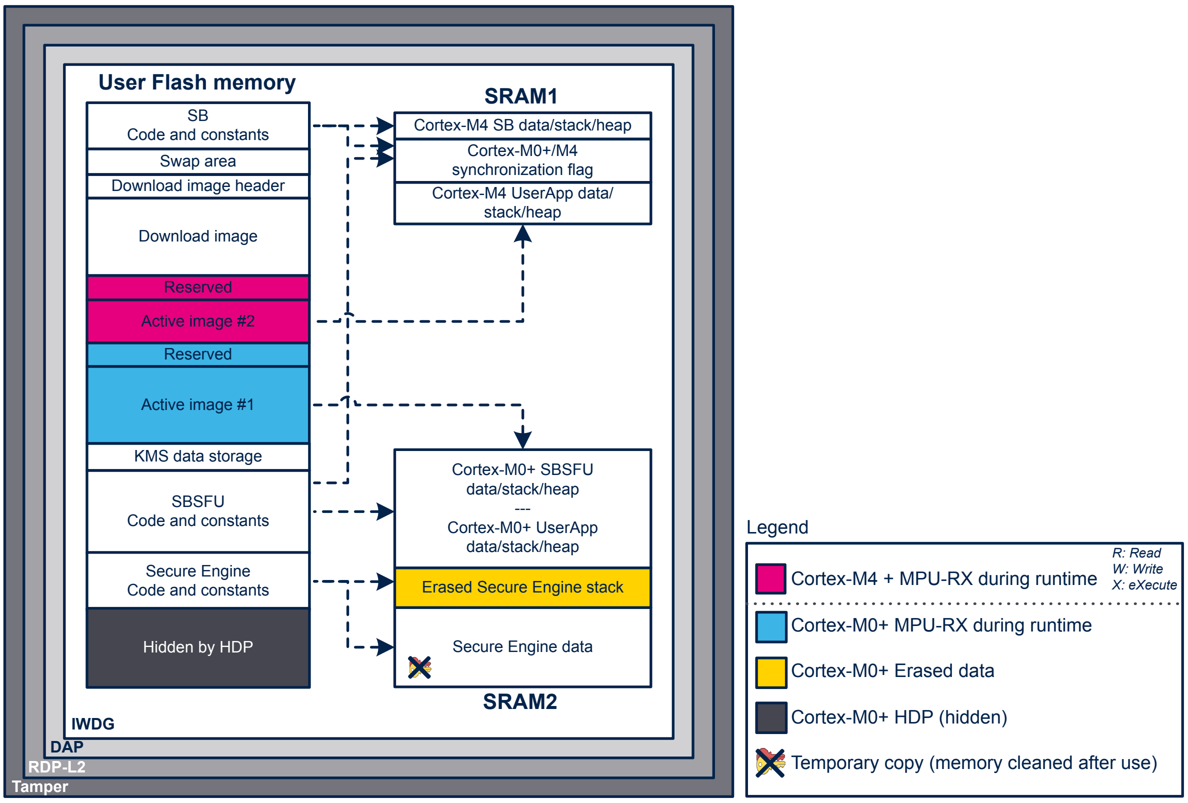
- **Secure mode (ESE = 1)**: the secure Flash memory and RAM are dedicated to the Cortex®-M0+ core and therefore cannot be accessed from the Cortex®-M4 core. The protection restrains the access to some peripherals like AES, RNG, PKA, PWR, FLASHIF and DMA (DMA1, DMA2, DMAMUX) to the secure and/or privileged mode. By default, SUBGHZSPI is accessible by secure-unprivileged applications; it is possible to restrain the access to secure-privileged applications.
- **WRP** (write protection): write protection is used to protect trusted code from external attacks or even internal modifications such as unwanted writings/erase operations on critical code/data. Moreover, WRP allows protecting the SBSFU public key.
- **TZSC** (security controller): the protected environment managing all critical data and operations (Secure Engine) is isolated from the other software components by leveraging TZSC. The Secure Engine code and data can be accessed only through a secure privileged level of software execution. Therefore, software running at a non-secure and/or non-privileged level cannot call the Secure Engine services nor access the critical data. This strict access control to Secure Engine services and resources is implemented by defining specific GTZC regions for the permissions and MPU regions for the execution capabilities as described in the table below.

Table 4. **GTZC and MPU regions**

Region content	Secure Privileged permission	Non-Secure and/or unprivileged permission
Secure Engine code and constants	Read only (execution allowed)	No access
Secure Engine stack and data	Read write (not executable)	No access

- **TZIC** (security illegal access-controller) configures interrupts for each detected illegal access, thus making possible to implement countermeasures.
- **MPU** (memory protection unit): the MPU is used to make an embedded system more robust by splitting the memory map for Flash memory and SRAMs into regions having their access rights. In the SBSFU application examples, the MPU is configured to ensure that no other code is executed from any memory during SBSFU code execution. When leaving the SBSFU application, the MPU configuration is updated to also authorize the execution of the user application code. Nevertheless, the Secure Engine isolation settings and supervisor call mechanisms (TZSC related) still apply when running the user applications (not only when running the SBSFU code).
- **HDP** (hide protection): once the user applications are running, the SBSFU keys and the FW headers are hidden and cannot be accessed anymore.

Figure 6. STM32WL Series devices protection overview during UserApp execution


User Flash memory
SB Code and constants
Swap area
Download image header
Download image
Reserved
Active image #2
Reserved
Active image #1
KMS data storage
SBSFU Code and constants
Secure Engine Code and constants
Hidden by HDP
SRAM1
Cortex-M4 SB data/stack/heap
Cortex-M0+/M4 synchronization flag
Cortex-M4 UserApp data/ stack/heap
Cortex-M0+ SBSFU data/stack/heap

Cortex-M0+ UserApp data/stack/heap
Erased Secure Engine stack
Secure Engine data
SRAM2
IWDG
DAP
RDP-L2
Tamper
Legend
R: Read
W: Write
X: eXecute
Cortex-M4 + MPU-RX during runtime
Cortex-M0+ MPU-RX during runtime
Cortex-M0+ Erased data
Cortex-M0+ HDP (hidden)
Temporary copy (memory cleaned after use)

6 Package description

This section details the STM32CubeWL SBSFU application content and the way to use it.

6.1 General description

SBSFU is a software application delivered in STM32CubeWL.

It provides a complete solution to build Secure Boot and Secure Firmware Update applications:

- Support of symmetric and asymmetric cryptography approaches with the AES-GCM, AES-CBC, and ECDSA algorithms for decryption, verification, or both with the use of hardware components accessed through HAL
- Two modes of operation:
  - The dual-slot configuration, which enables safe image programming, with resume capability in the case of an interruption of the installation procedure. It is based on:
    - one active slot and one download slot for the single-core configuration
    - two active slots and one download slot for the dual-core configuration
  - The single-slot configuration, which maximizes the user application size, with:
    - one active slot for the single-core configuration
    - two active slots for the dual-core configuration
- Integration of security peripherals and mechanisms to implement a SBSFU Root of Trust: RDP, WRP, secure mode, GTZC (TZSC, TZIC), MPU, HDP, tamper, and IWDG are combined to achieve the highest security level
- Use of a Secure Engine (SE) module as part of the middleware to provide a protected environment managing all critical data and operations such as secure key storage, cryptographic operations, and others
- Integration of secure Key Management Services (KMS) offering symmetric and asymmetric cryptographic services via the PKCS#11 APIs and offering secure key storage, update services
- Availability of the Cortex®-M4 and Cortex®-M0+ user application example source code
- The firmware image configuration can be extended up to three images for a complex system with multiple firmware (such as protocol stack, middleware, and user application.)
- Availability of the firmware image preparation tool provided both as executable and source code.

The package includes sample applications that the developer can use to start experimenting with the code.

The package is provided as a zip archive containing the source code.

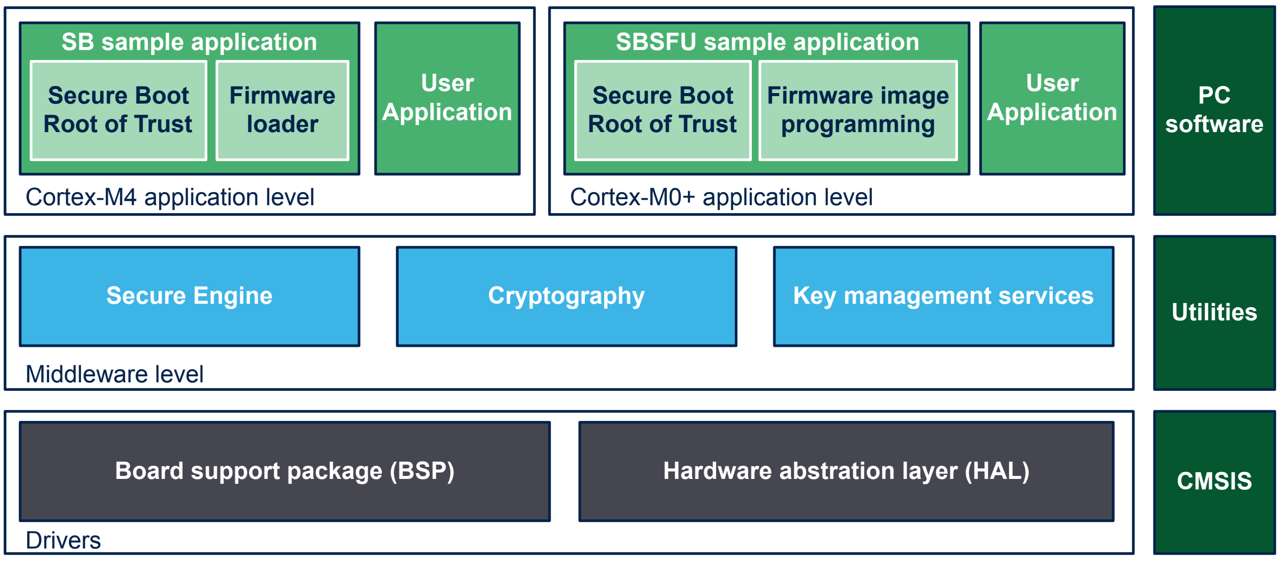
The following integrated development environments are supported:

- STM32CubeIDE
- IAR Embedded Workbench® for Arm® (EWARM)
- Keil® Microcontroller Development Kit (MDK-ARM).

6.2 Architecture

This section describes the software components of the STM32CubeWL SBSFU package illustrated in the figure below.

Figure 7. Software architecture overview (dual-slot configuration)


SB sample application
Secure Boot Root of Trust
Firmware loader
User Application
Cortex-M4 application level
SBSFU sample application
Secure Boot Root of Trust
Firmware image programming
User Application
Cortex-M0+ application level
PC software
Secure Engine
Cryptography
Key management services
Middleware level
Utilities
Board support package (BSP)
Hardware abstration layer (HAL)
Drivers
CMSIS

Note: *In the dual-core single-slot configuration, the software component "Firmware loader" is integrated within the "SBSFU sample application" and not within the "SB sample application". With the single-slot configuration, the downloaded firmware is written directly in its matching active slot: only Cortex®-M0+ has the access rights to update both the non-secure application and the secure application.*

6.2.1 STM32Cube HAL

The HAL driver layer provides a generic multi-instance simple set of APIs (application programming interface) to interact with the upper layers (application, libraries and stacks). It is composed of generic and extension APIs. It is directly built around a generic architecture and allows the layers that are built upon, such as the middleware layer, implementing their functionalities without dependencies on the specific hardware configuration for a given microcontroller unit (MCU). This structure improves the library code re-usability and guarantees an easy portability onto other devices.

6.2.2 Board support package (BSP)

The software package needs to support the peripherals on the STM32 boards apart from the MCU. This software is included in the board support package (BSP). This is a limited set of APIs which provides a programming interface for certain board-specific peripherals such as the LED and the *user* button.

6.2.3 Cryptographic library

The mbed-crypto cryptographic middleware is supported. It delivers cryptographic services as open-source code. STM32CubeWL SBSFU uses this middleware for its SHA256 cryptography API.

Note: *The other cryptographic algorithms use HAL to access to the related hardware components (AES, PKA).*

6.2.4 Secure Engine (SE) middleware

The Secure Engine middleware provides a protected environment to manage all critical data and operations (such as cryptography operations accessing firmware encryption key and others). Protected code and data are accessible through a single entry-point (call-gate mechanism) and it is therefore not possible to run or access any SE code or data without passing through it, otherwise, a system reset is generated (refer to Section Appendix A Secure Engine protected environment to get details about call-gate mechanism).

Note: Secure Engine critical operations can be extended with other functions depending on user application needs. Only trusted code is to be added to the Secure Engine environment because it has access to the secrets.

6.2.5 Key management services (KMS) middleware

The secure key management services provide cryptographic services to the user application through the PKCS#11 APIs (KEY ID-based APIs) that are executed inside the secure enclave in the dual-core configuration. User application keys are stored in the secure enclave and can be updated securely (authenticity check, decryption, and integrity check before the update).

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

6.2.6 Secure Boot and Secure Firmware Update (SBSFU) application

Secure Boot (Root of Trust)

- Is compounded of two parts: Cortex®-M4 Secure Boot and Cortex®-M0+ SBSFU
- Checks and applies the security mechanisms of the STM32 platform to protect critical operations and secrets from attacks
- Authenticates and verifies the user application before each execution

Local download via UART Virtual COM port

- Detects firmware download requests
- Downloads in STM32 Flash memory the new encrypted firmware image (header and encrypted firmware) via the UART Virtual COM port using Ymodem protocol and the Tera Term tool

Firmware installation management

- Detects new firmware version to install
  - From local download service via the UART interface
  - Downloaded via the user application (dual-slot variant only)
- Secures firmware upgrade:
  - Authentication and integrity check
  - Firmware decryption
  - Firmware installation
  - Anti-rollback mechanisms to avoid reinstallation of previous firmware version
- Supports multiple images:
  - Up to three active slots and three download slots for a complex system with multiple firmware, such as protocol stack, middleware, and user application
  - Specific cryptographic keys per slot
  - Simultaneous image installation to ensure compatibility between firmware
- Supports single-slot configuration for maximizing the user application size
- Supports dual-slot configuration for safe image programming
  - Resume firmware installation: in the case of power off during the installation process, installation is resumed at the next power on
  - Installation with the SWAP area to limit needed memory overhead. Refer to Section Appendix B Dual-slot configuration to get details about multiple slots management)
  - Installation without the SWAP area to keep the download area encrypted
  - Partial update: flexibility to update the complete firmware image or a portion of it

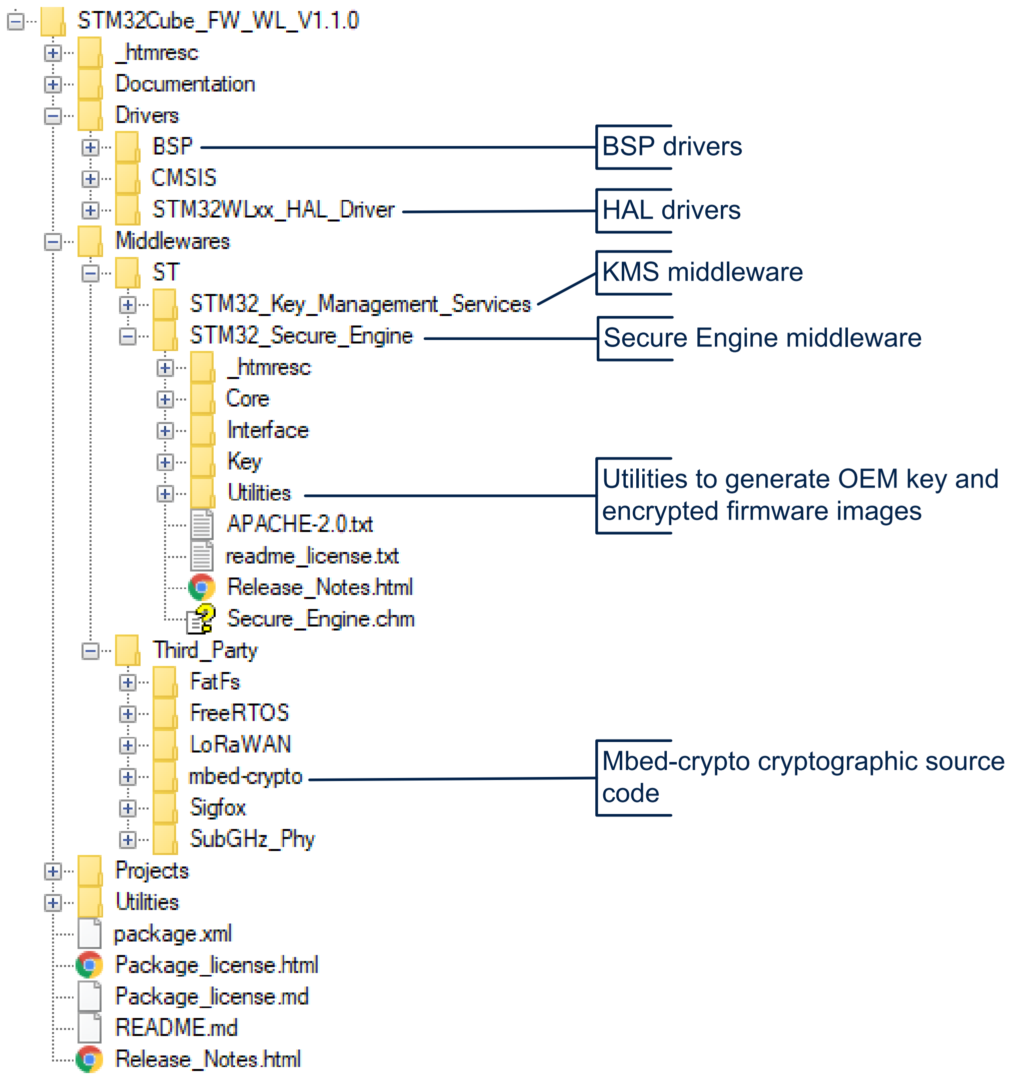
6.2.7 User application

- Provides an example for downloading the user application via Ymodem protocol over a UART (the application “LoRaWAN_FUOTA” illustrates *over-the-air* download mechanism)
- Provides examples testing the protection mechanisms
- Provides an example for using some of the functionalities exported by SE such as getting information about the current firmware image
- Provides examples using KMS exported services through a standard PKCS#11 interface: AES-GCM/CBC encryption/decryption, RSA signature/verification, key provisioning, and AES ECB key derivation

6.3 Folder structure

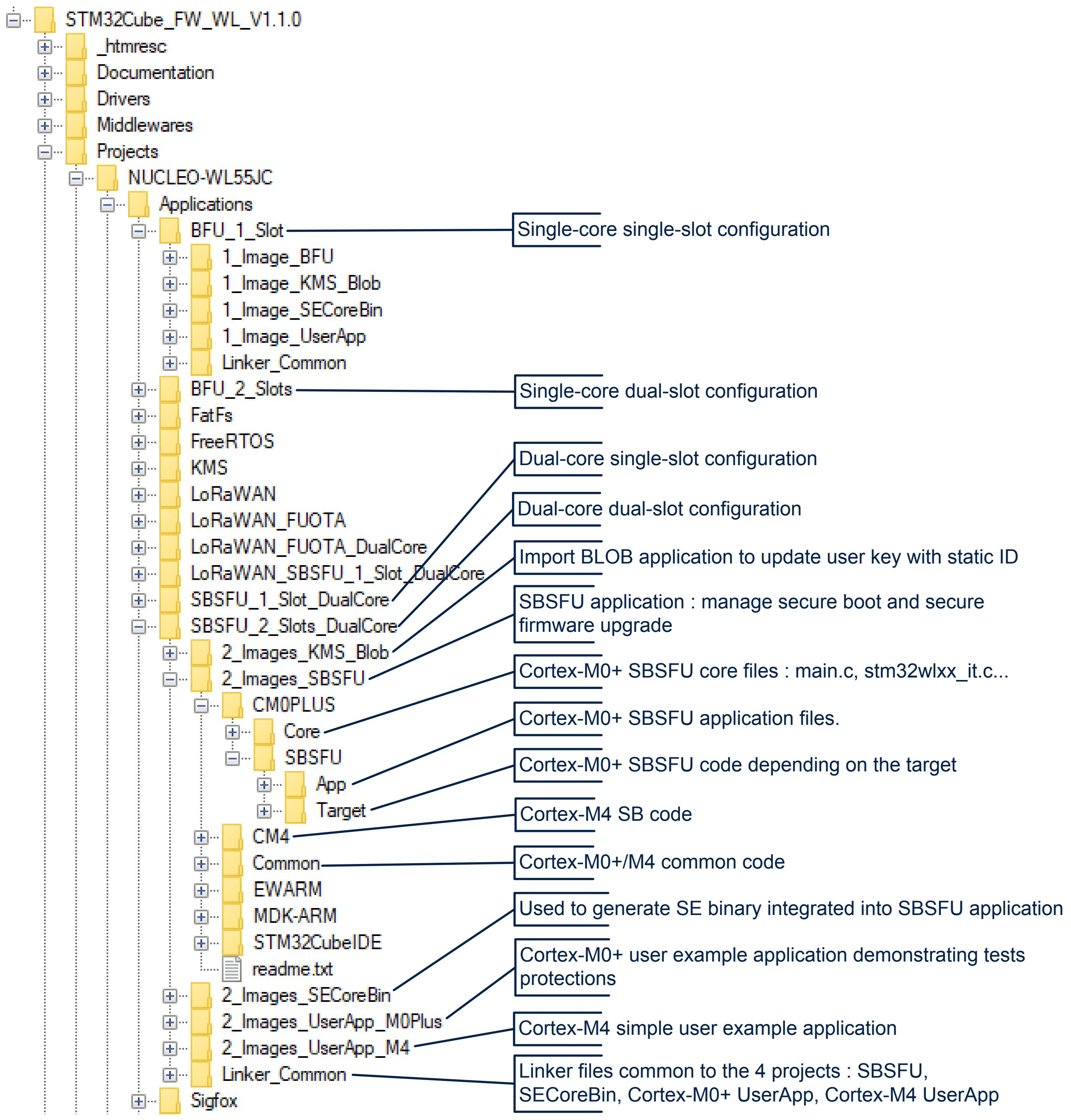
A top-level view of the folder structure is shown in the following two figures.

Figure 8. **Project folder structure (1 of 2)**



```
STM32Cube_FW_WL_V1.1.0
  _htmresc
  Documentation
  Drivers
    BSP                                   BSP drivers
    CMSIS
    STM32WLxx_HAL_Driver                  HAL drivers
  Middlewares
    ST
      STM32_Key_Management_Services       KMS middleware
      STM32_Secure_Engine                 Secure Engine middleware
        _htmresc
        Core
        Interface
        Key
        Utilities                         Utilities to generate OEM key and
                                          encrypted firmware images
        APACHE-2.0.txt
        readme_license.txt
        Release_Notes.html
        Secure_Engine.chm
    Third_Party
      FatFs
      FreeRTOS
      LoRaWAN
      mbed-crypto                         Mbed-crypto cryptographic source
                                          code
      Sigfox
      SubGHz_Phy
  Projects
  Utilities
  package.xml
  Package_license.html
  Package_license.md
  README.md
  Release_Notes.html
```

Figure 9. Project folder structure (2 of 2)


STM32Cube_FW_WL_V1.1.0
_htmresc
Documentation
Drivers
Middlewares
Projects
NUCLEO-WL55JC
Applications
BFU_1_Slot
1_Image_BFU
1_Image_KMS_Blob
1_Image_SECoreBin
1_Image_UserApp
Linker_Common
BFU_2_Slots
FatFs
FreeRTOS
KMS
LoRaWAN
LoRaWAN_FUOTA
LoRaWAN_FUOTA_DualCore
LoRaWAN_SBSFU_1_Slot_DualCore
SBSFU_1_Slot_DualCore
SBSFU_2_Slots_DualCore
2_Images_KMS_Blob
2_Images_SBSFU
CM0PLUS
Core
SBSFU
App
Target
CM4
Common
EWARM
MDK-ARM
STM32CubeIDE
readme.txt
2_Images_SECoreBin
2_Images_UserApp_M0Plus
2_Images_UserApp_M4
Linker_Common
Sigfox
Single-core single-slot configuration
Single-core dual-slot configuration
Dual-core single-slot configuration
Dual-core dual-slot configuration
Import BLOB application to update user key with static ID
SBSFU application : manage secure boot and secure firmware upgrade
Cortex-M0+ SBSFU core files : main.c, stm32wlxx_it.c...
Cortex-M0+ SBSFU application files.
Cortex-M0+ SBSFU code depending on the target
Cortex-M4 SB code
Cortex-M0+/M4 common code
Used to generate SE binary integrated into SBSFU application
Cortex-M0+ user example application demonstrating tests protections
Cortex-M4 simple user example application
Linker files common to the 4 projects : SBSFU, SECoreBin, Cortex-M0+ UserApp, Cortex-M4 UserApp

Note:
- *Single-slot configuration is demonstrated in the applications named BFU_1_Slot and SBSFU_1_Slot_DualCore*
- *Dual-slot configuration is demonstrated in the applications named BFU_2_Slots and SBSFU_2_Slots_DualCore*

6.4 APIs

Detailed technical information about the APIs is provided in a compiled HTML file located in the *STM32_Secure_Engine*, and *STM32_Key_Management_Services* folders of the software package where all the functions and parameters are described.

6.5 Application compilation process with IAR Embedded Workbench® toolchain

The figure below outlines the steps needed to build the application and to demonstrate Secure Boot and Secure Firmware Update:

1. Core binaries preparation

   This step is needed to create the Secure Engine core binary including all the ‘trusted’ code and keys mapped in the secure code section. The SE call-gate function is specified as the entry point for the binary. The binary is linked with the Cortex®-M0+ SBSFU code in step 2.
2. Cortex®-M0+ SBSFU

   This step compiles the SBSFU source code implementing the state machine and configuring the protections. It also links the code with the SE core binary generated at step 1 to generate a single SBSFU binary including the SE trusted code. It also generates a file including symbols for the user application to call the SE interface methods, a set of user-friendly APIs wrapping the single SE call gate API.
3. Cortex®-M4 Secure Boot

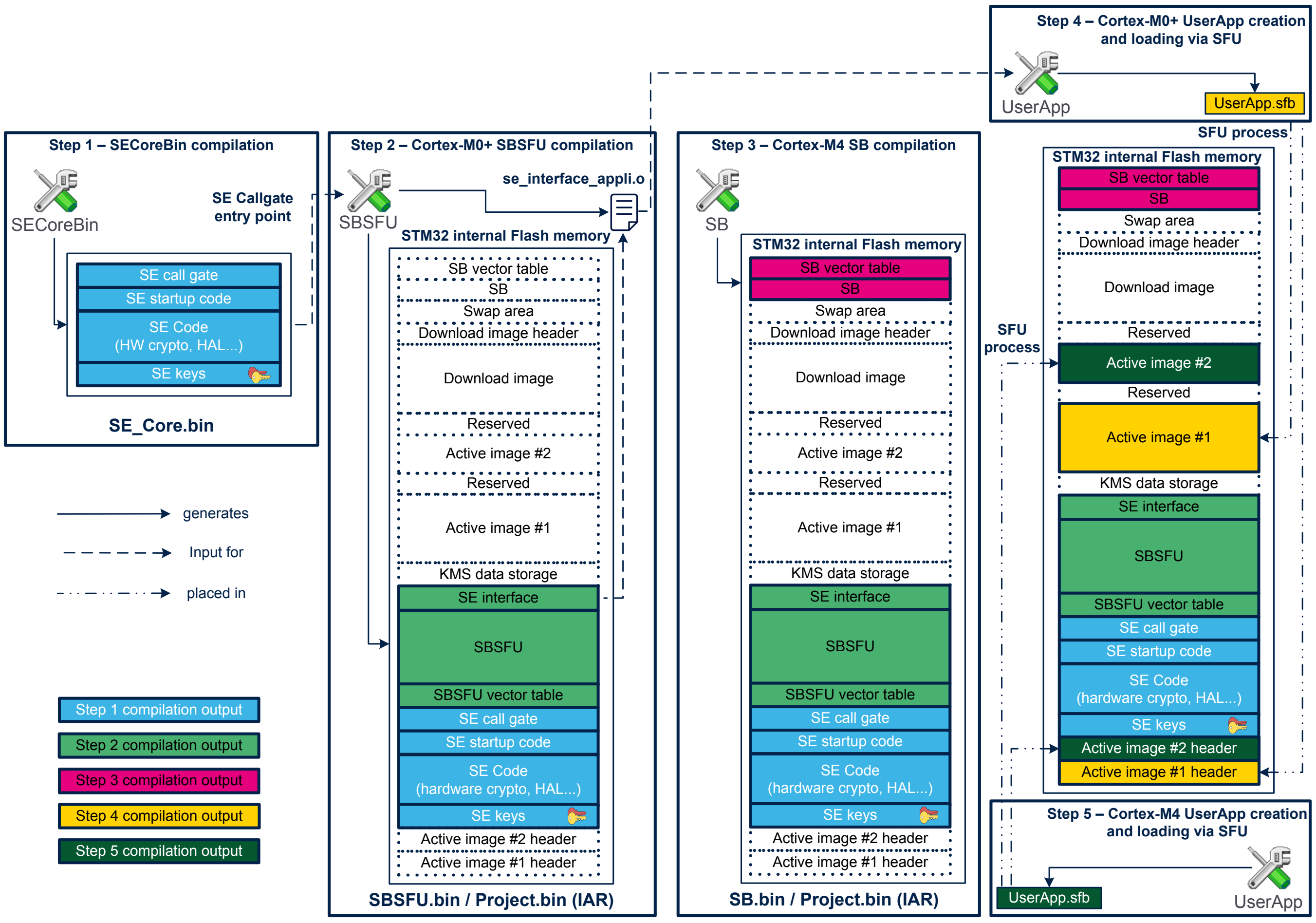
   This step compiles the Secure Boot source code configuring the protections on the Cortex®-M4 side.
4. Cortex®-M0+ user application example

   It generates:
   – The user application binary file that is uploaded to the device using the Secure Firmware Update process (*UserApp.sfb*)
5. Cortex®-M4 user application example

   It generates:
   – The user application binary file that is uploaded to the device using the Secure Firmware Update process (*UserApp.sfb*)
   – A binary file concatenating the SBSFU binary, the Secure Boot binary, both Cortex®-M4 and Cortex®-M0+ user application binaries in clear format, and the corresponding firmware headers

     These four elements are placed properly for the Cortex®-M4 Secure Boot, the Cortex®-M0+ SBSFU and both user applications to run when the binary file is flashed into the device with a flasher tool. Hence, no firmware installation procedure is required for Cortex®-M4 Secure Boot and Cortex®-M0+ SBSFU to start and boot the user applications. This is a convenient way to test the user applications with a single flashing stage.

Figure 10. Application compilation steps


Step 1 – SECoreBin compilation
SECoreBin
SE Callgate entry point
SE call gate
SE startup code
SE Code (HW crypto, HAL...)
SE keys
SE_Core.bin
Step 2 – Cortex-M0+ SBSFU compilation
SBSFU
se_interface_appli.o
STM32 internal Flash memory
SB vector table
SB
Swap area
Download image header
Download image
Reserved
Active image #2
Reserved
Active image #1
KMS data storage
SE interface
SBSFU
SBSFU vector table
SE call gate
SE startup code
SE Code (hardware crypto, HAL...)
SE keys
Active image #2 header
Active image #1 header
SBSFU.bin / Project.bin (IAR)
Step 3 – Cortex-M4 SB compilation
SB
STM32 internal Flash memory
SB.bin / Project.bin (IAR)
Step 4 – Cortex-M0+ UserApp creation and loading via SFU
UserApp
UserApp.sfb
SFU process
Step 5 – Cortex-M4 UserApp creation and loading via SFU
generates
Input for
placed in
Step 1 compilation output
Step 2 compilation output
Step 3 compilation output
Step 4 compilation output
Step 5 compilation output

Note: *The figure above presents the mapping of the dual-core dual-slot configuration. The application compilation steps are the same for the dual-core single-slot configuration.*

7 Hardware and software environment setup

This section describes the hardware and software setup procedures.

7.1 Hardware setup

To set up the hardware environment, a NUCLEO-WL55JC board must be connected to a personal computer via a USB cable. This connection with the PC allows the user to:

- Flash the board
- Interact with the board via a UART console
- Debug when the protections are disabled

7.2 Software setup

This section lists the minimum requirements for the developer to set up the SDK (software development kit), run the sample scenario, and customize applications.

7.2.1 Development toolchains and compilers

Select one of the integrated development environments supported by STM32CubeWL SBSFU.

The system requirements and setup information provided by the selected IDE provider must be taken into consideration.

Note: For instance, when using STM32CubeIDE on macOS®, it must be checked that the default shell is bash and not Zsh to avoid compilation issues.

7.2.2 Software tools for programming STM32 microcontrollers

ST-LINK utility

STM32 ST-LINK Utility (STSW-LINK004) is a full-featured software interface for programming STM32 microcontrollers. It provides an easy-to-use and efficient environment for reading, writing, and verifying a memory device.

Refer to the STSW-LINK004 STM32 ST-LINK Utility software on www.st.com.

Note: Make sure to use an up-to-date version of ST-LINK.

STM32CubeProgrammer

STM32CubeProgrammer (STM32CubeProg) is an all-in-one multi-OS software tool for programming STM32 microcontrollers. It provides an easy-to-use and efficient environment for reading, writing, and verifying device memory through both the debug interface (JTAG and SWD) and the bootloader interface (UART and USB).

STM32CubeProgrammer offers a wide range of features to program STM32 microcontroller internal memories (such as Flash memory, RAM, and OTP) as well as external memories (when supported). STM32CubeProgrammer also allows option programming and upload, programming content verification, and microcontroller programming automation through scripting.

STM32CubeProgrammer is delivered in GUI (graphical user interface) and CLI (command line interface) versions.

Refer to the STM32CubeProgrammer software (STM32CubeProg) on www.st.com.

7.2.3 Terminal emulator

A terminal emulator software is needed to run the demonstration.

The example in this document is based on Tera Term, an open-source free software terminal emulator. Any other similar tool can be used instead (Ymodem protocol support is required).

7.2.4 STM32CubeWL SBSFU firmware image preparation tool

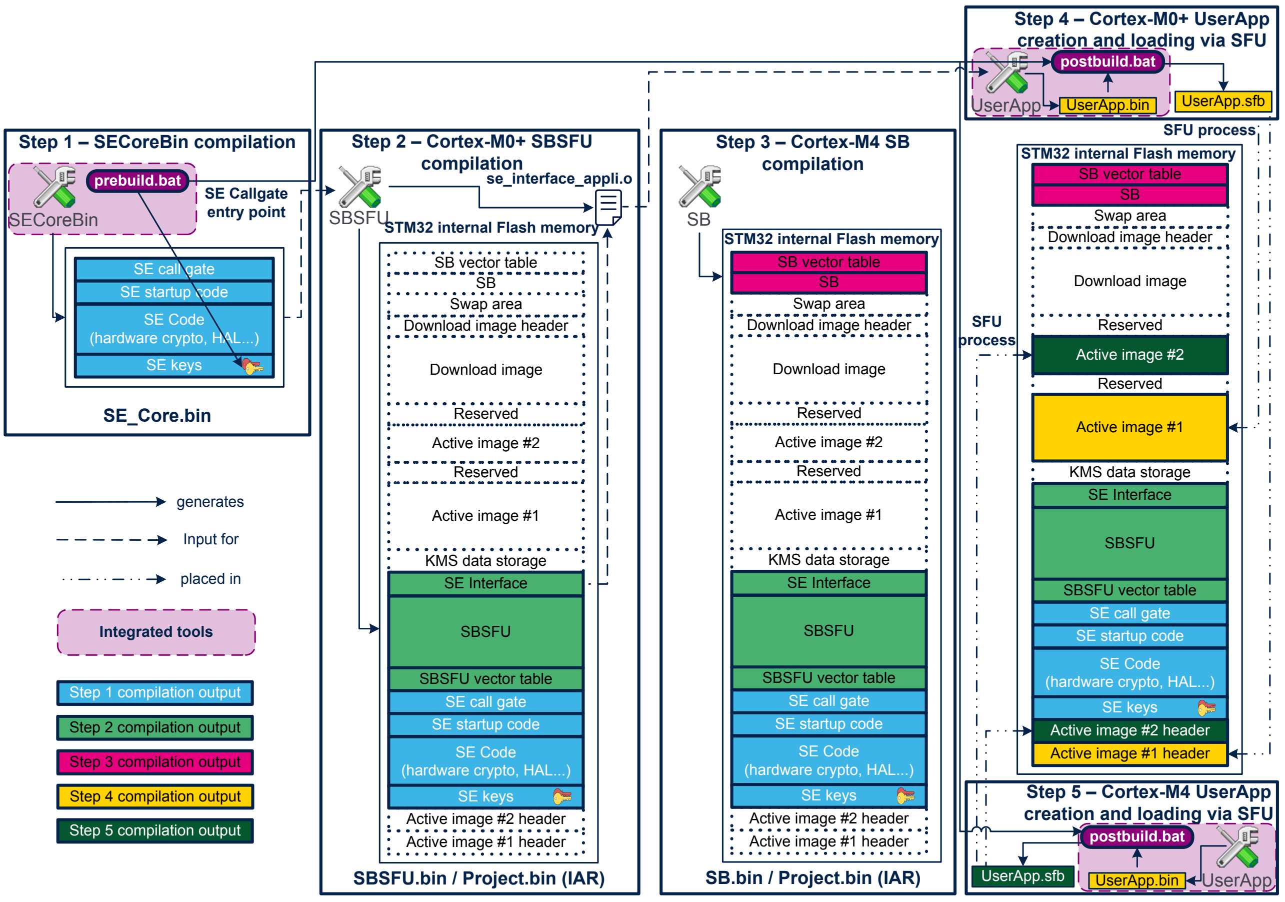
The STM32CubeWL SBSFU is delivered with the *prepareimage* tool which handles the cryptographic keys and firmware image preparation.

The *prepareimage* tool is delivered in two formats:

- Windows® executable: the standard Windows® command interpreter is required
- Python™ scripts: a Python™ interpreter as well as the elements listed in *Middlewares\ST\STM32_Secure_Engine\Utilities\KeysAndImages\readme.txt* are required

The Windows® executable is fully integrated into the supported IDEs and compilation process as shown in the figure below.

Figure 11. Firmware image preparation tool IDE integration


Step 1 – SECoreBin compilation
prebuild.bat
SECoreBin
SE Callgate entry point
SE call gate
SE startup code
SE Code (hardware crypto, HAL...)
SE keys
SE_Core.bin
Step 2 – Cortex-M0+ SBSFU compilation
se_interface_appli.o
SBSFU
STM32 internal Flash memory
SB vector table
SB
Swap area
Download image header
Download image
Reserved
Active image #2
Active image #1
KMS data storage
SE Interface
SBSFU vector table
Active image #2 header
Active image #1 header
SBSFU.bin / Project.bin (IAR)
Step 3 – Cortex-M4 SB compilation
SB.bin / Project.bin (IAR)
Step 4 – Cortex-M0+ UserApp creation and loading via SFU
postbuild.bat
UserApp
UserApp.bin
UserApp.sfb
SFU process
Step 5 – Cortex-M4 UserApp creation and loading via SFU
generates
Input for
placed in
Integrated tools
Step 1 compilation output
Step 2 compilation output
Step 3 compilation output
Step 4 compilation output
Step 5 compilation output

Note: The figure above presents the mapping of the dual-core dual-slot configuration. The steps are the same for the dual-core single-slot configuration.

More information about the preparation tool is provided in Section Appendix E Firmware image preparation tool.

8 Step-by-step execution

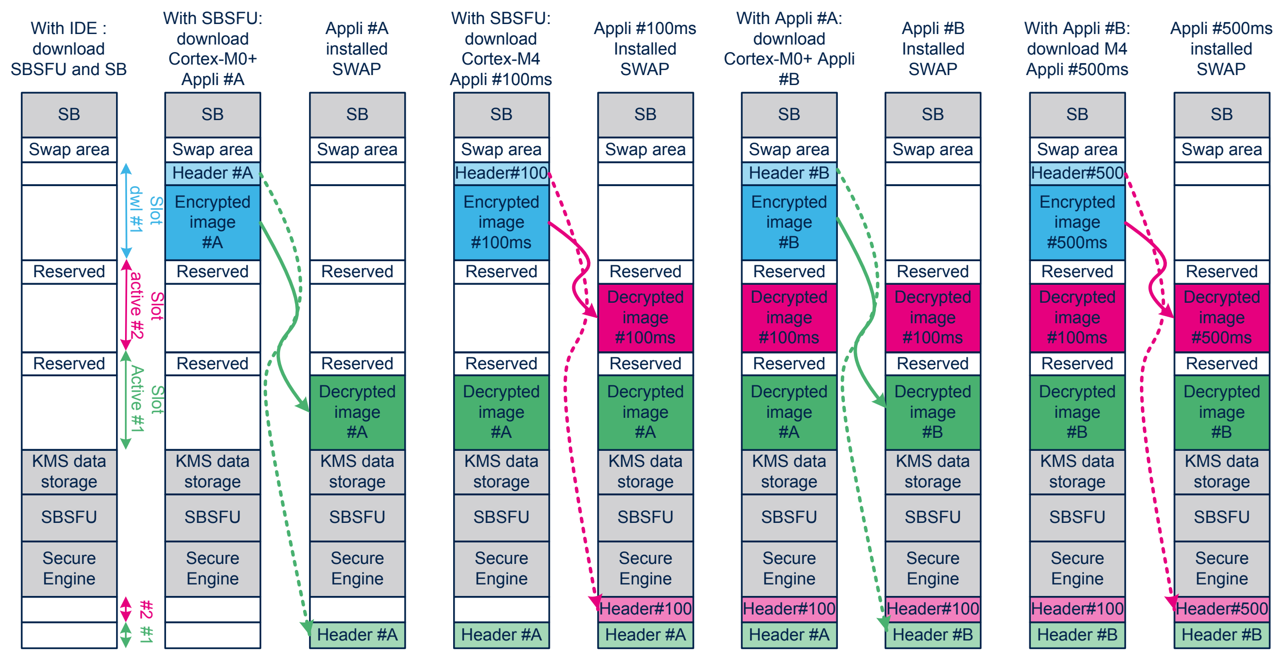
The following steps describe a dual-slot SBSFU scenario executed on the NUCLEO-WL55JC board with the default cryptographic scheme, further illustrated in the figure below:

1. Download Secure Boot and SBSFU applications
2. Secure Boot and SBSFU are running: download Cortex®-M0+ UserApp #A
3. Cortex®-M0+ UserApp #A is installed
4. Secure Boot and SBSFU are running: download Cortex®-M4 UserApp (100 ms LED blinking)
5. Cortex®-M4 UserApp (100 ms LED blinking) is installed then Cortex®-M0+ UserApp #A and Cortex®-M4 UserApp (100 ms LED blinking) are running
6. Download Cortex®-M0+ UserApp #B
7. UserApp #B is installed then Cortex®-M0+ UserApp #B and Cortex®-M4 UserApp (100 ms LED blinking) are running
8. Download Cortex®-M4 UserApp (500 ms LED blinking)
9. Cortex®-M4 UserApp (500 ms LED blinking) is installed then Cortex®-M0+ UserApp #B and Cortex®-M4 UserApp (500 ms LED blinking) are running

The Cortex®-M0+ UserApp #A and Cortex®-M0+ UserApp #B binaries are generated based on the Cortex®-M0+ user application example project. Defining the application as #A or #B is done by changing the value of the *UserAppId* variable declared in the *main.c* of the Cortex®-M0+ application.

The Cortex®-M4 UserApp (100 ms LED blinking) and Cortex®-M4 UserApp (500 ms LED blinking) binaries are generated based on the Cortex®-M4 user application example project. Defining the delay 100 ms or 500 ms is done by changing the value used by "HAL_Delay" in the *main.c* of the Cortex®-M4 application.

Figure 12. Step-by-step execution



8.1 STM32 board preparation

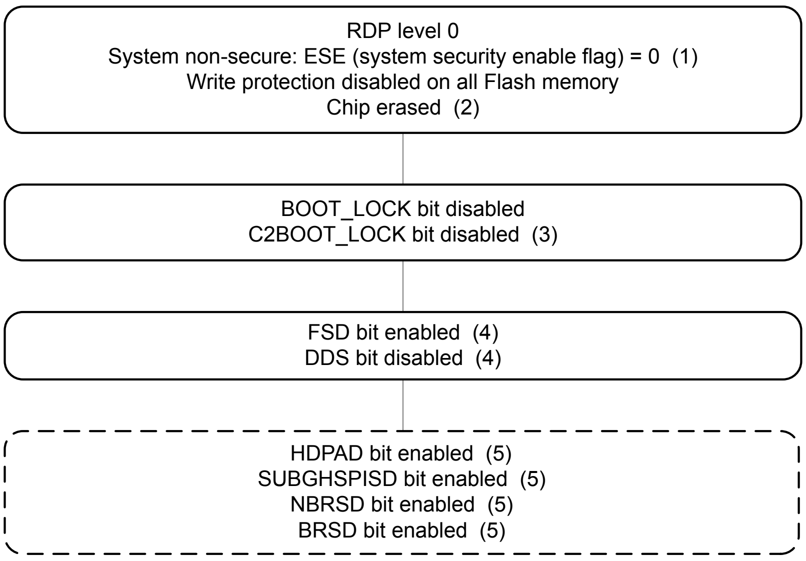
As illustrated in the figure below, the target option bytes setting is the following for the NUCLEO-WL55JC board:

- RDP Level 0 set
- ESE disabled[1]
- BOOT_LOCK disabled
- C2BOOT_LOCK disabled
- FSD enabled and SFSA = 0x7f
- DDS disabled
- HDPAD enabled and HDPSA = 0x7f
- SUBGHSPISD enabled
- C2OPT enabled and SBRV = 0x8000
- NBRSD enabled and SNBRSA = 0x1f
- BRSD enabled and SBRSA = 0x1f
- Write protection disabled
  - WRP1A_STRT = WRP1B_STRT= 0x7f
  - WRP1A_END = WRP1B_END = 0x0
- Chip is erased[2]

1. *ESE can only be set to 0 when regressing the RDP from level 1 to level 0 at the same time.*
2. *Automatically done when switching from RDP level 1 to RDP level 0.*

Note: *FSD, HDPAD, SUBGHSPISD, NBRSD and BRD disable the security features when set.*

Figure 13. **STM32 board preparation**

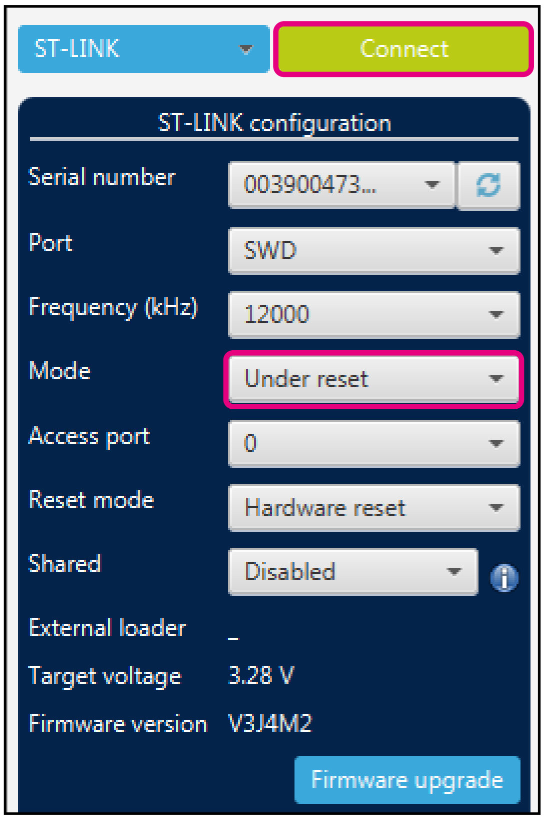

RDP level 0
System non-secure: ESE (system security enable flag) = 0 (1)
Write protection disabled on all Flash memory
Chip erased (2)
BOOT_LOCK bit disabled
C2BOOT_LOCK bit disabled (3)
FSD bit enabled (4)
DDS bit disabled (4)
HDPAD bit enabled (5)
SUBGHSPISD bit enabled (5)
NBRSD bit enabled (5)
BRSD bit enabled (5)

(1) ESE can only be set to 0 when switching from RDP level 1 to RDP level 0
(2) Automatically done when switching from RDP level 1 to RDP level 0
(3) When this bit set, it is impossible to change the Cortex-M0+ boot address
(4) Value automatically set when ESE is set from 1 to 0
(5) When FSD = 0, the related feature is disabled whichever this bit value

Option bytes setting is verified using the STM32CubeProgrammer through the following four steps:

1. **Connect the board in "Under reset" mode**

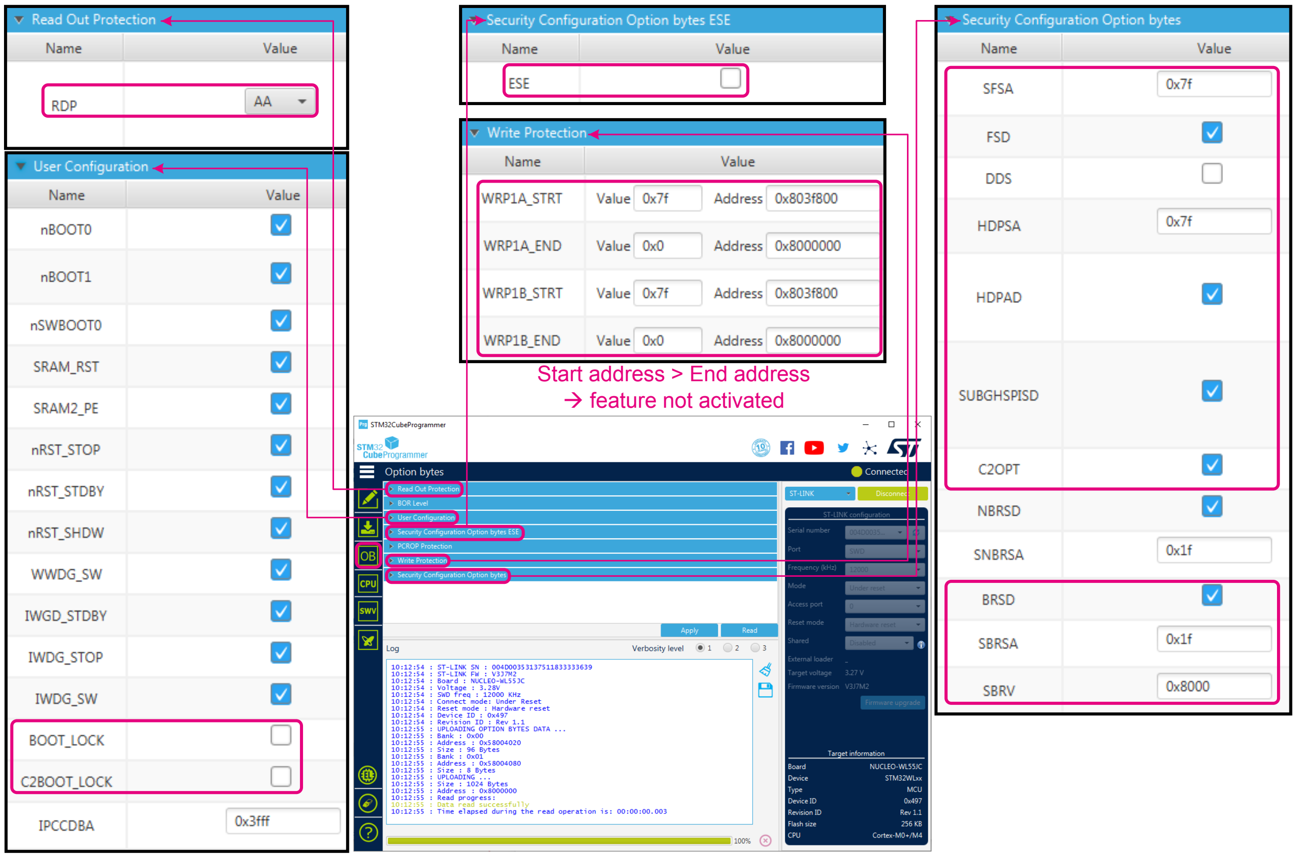
Figure 14. STM32CubeProgrammer connection menu



Serial number	003900473...
Port	SWD
Frequency (kHz)	12000
Mode	Under reset
Access port	0
Reset mode	Hardware reset
Shared	Disabled
External loader	_
Target voltage	3.28 V
Firmware version	V3J4M2

It is recommended to upgrade the firmware version of ST-LINK through the button ‘Firmware upgrade’.

2. Verify "option bytes" configuration



Read Out Protection

Name	Value
RDP	AA

User Configuration

Name	Value
nBOOT0	✓
nBOOT1	✓
nSWBOOT0	✓
SRAM_RST	✓
SRAM2_PE	✓
nRST_STOP	✓
nRST_STDBY	✓
nRST_SHDW	✓
WWDG_SW	✓
IWGD_STDBY	✓
IWDG_STOP	✓
IWDG_SW	✓
BOOT_LOCK	☐
C2BOOT_LOCK	☐
IPCCDBA	0x3fff

Security Configuration Option bytes ESE

Name	Value
ESE	☐

Write Protection

Name	Value
WRP1A_STRT	Value 0x7f Address 0x803f800
WRP1A_END	Value 0x0 Address 0x8000000
WRP1B_STRT	Value 0x7f Address 0x803f800
WRP1B_END	Value 0x0 Address 0x8000000

Security Configuration Option bytes

Name	Value
SFSA	0x7f
FSD	✓
DDS	☐
HDPSA	0x7f
HDPAD	✓
SUBGHSPISD	✓
C2OPT	✓
NBRSD	✓
SNBRSA	0x1f
BRSD	✓
SBRSA	0x1f
SBRV	0x8000

ST-LINK configuration

Serial number	004D0035...
Port	SWD
Frequency (kHz)	12000
Mode	Under reset
Access port	0
Reset mode	Hardware reset
Shared	Disabled
External loader	_
Target voltage	3.27 V
Firmware version	V3J7M2

Target information

Board	NUCLEO-WL55JC
Device	STM32WLxx
Type	MCU
Device ID	0x497
Revision ID	Rev 1.1
Flash size	256 KB
CPU	Cortex-M0+/M4

Log Verbosity level [✓ 1] 2 3

```
10:12:54 : ST-LINK SN : 004D00353137511833333639
10:12:54 : ST-LINK FW : V3J7M2
10:12:54 : Board : NUCLEO-WL55JC
10:12:54 : Voltage : 3.28V
10:12:54 : SWD freq : 12000 KHz
10:12:54 : Connect mode: Under Reset
10:12:54 : Reset mode : Hardware reset
10:12:54 : Device ID : 0x497
10:12:54 : Revision ID : Rev 1.1
10:12:55 : UPLOADING OPTION BYTES DATA ...
10:12:55 : Bank : 0x00
10:12:55 : Address : 0x58004020
10:12:55 : Size : 96 Bytes
10:12:55 : Bank : 0x01
10:12:55 : Address : 0x58004080
10:12:55 : Size : 8 Bytes
10:12:55 : UPLOADING ...
10:12:55 : Size : 1024 Bytes
10:12:55 : Address : 0x8000000
10:12:55 : Read progress:
10:12:55 : Data read successfully
10:12:55 : Time elapsed during the read operation is: 00:00:00.003
```

100%

Figure 15. **Details of the STM32CubeProgrammer Option Bytes screen**

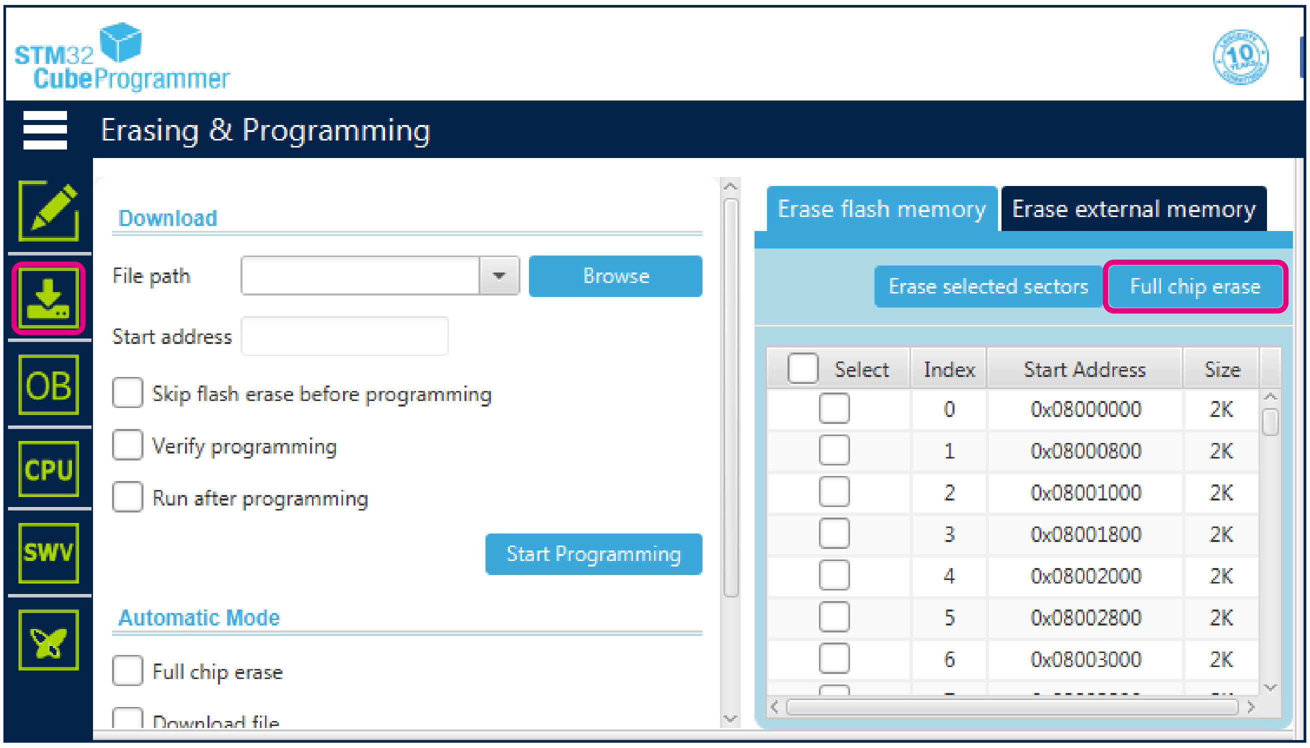
The framed option bytes are the ones used by the dual-slot SBSFU.

The STM32CubeProgrammer *option bytes* screen is specific to the STM32 microcontroller series.

3. Erase chip: Menu Target / Erase Chip

In case of memory-mapping change, this step makes sure there is no unexpected data in the NVM KMS data storage.

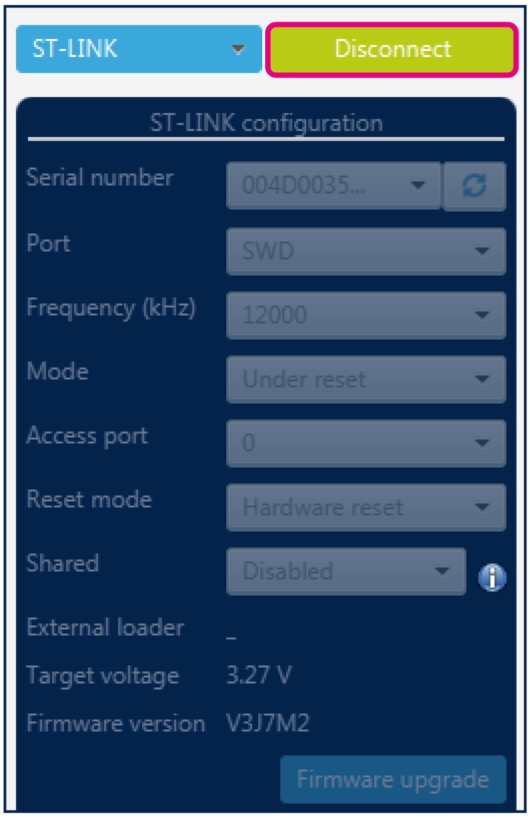
Figure 16. **STM32CubeProgrammer erasing**



Select	Index	Start Address	Size
☐	0	0x08000000	2K
☐	1	0x08000800	2K
☐	2	0x08001000	2K
☐	3	0x08001800	2K
☐	4	0x08002000	2K
☐	5	0x08002800	2K
☐	6	0x08003000	2K

4. Disconnect

Figure 17. **STM32CubeProgrammer disconnect menu**



Serial number 004D0035...
Port SWD
Frequency (kHz) 12000
Mode Under reset
Access port 0
Reset mode Hardware reset
Shared Disabled
External loader _
Target voltage 3.27 V
Firmware version V3J7M2

8.2 Application compilation

With the selected toolchain (IAR Embedded Workbench®, Keil®, or STM32CubeIDE) rebuild all the projects as explained in Section 6.5 Application compilation process with IAR Embedded Workbench toolchain.

Download the SBSFU project software (Cortex®-M4 Secure Boot binary and Cortex®-M0+ SBSFU binary) to the target without starting a debug session (security protections managed by SBSFU forbid JTAG connection as it is interpreted as an external attack).

Note: *With most toolchains, Cortex®-M4 Secure Boot binary must be running to make it possible to download Cortex®-M0+ SBSFU binary. However, as Cortex®-M4 Secure Boot binary sets some security protections when started, it is then impossible to download Cortex®-M0+ SBSFU binary. Most of the time, both binaries must be downloaded using STM32CubeProgrammer.*

8.3 Tera Term connection

Tera Term connection is achieved by applying in sequence the steps described in the following sections.

8.3.1 ST-LINK disable

The security mechanisms managed by SBSFU forbid JTAG connection (interpreted as an external attack). The ST-LINK must be disabled to establish a Tera Term connection. The following procedure applies from ST-LINK firmware version V2J29 onwards:

1. Power cycle the board after flashing SBSFU (unplug/plug the USB cable).
2. The SBSFU application starts and configures the security mechanisms in development mode. In product mode, security mechanisms are only checked to be at the correct values.

3. Power cycle the board a second time (unplug/plug the USB cable): the SBSFU application starts with the configured securities turned on and the Tera Term connection is possible.

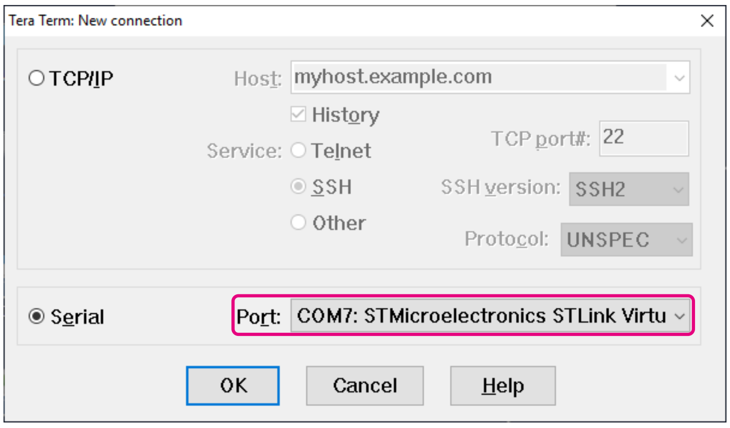
Note: Make sure the ST-LINK debugger/programmer embedded on the board runs the proper firmware version. If this is not the case, please upgrade this firmware first.

8.3.2 Tera Term launch

The Tera Term launch requires that the port is selected as *COMxx: STMicroelectronics STLink Virtual COM port*.

The following figure illustrates an example based on the selection of port COM7.

Figure 18. Tera Term connection screen



Tera Term: New connection

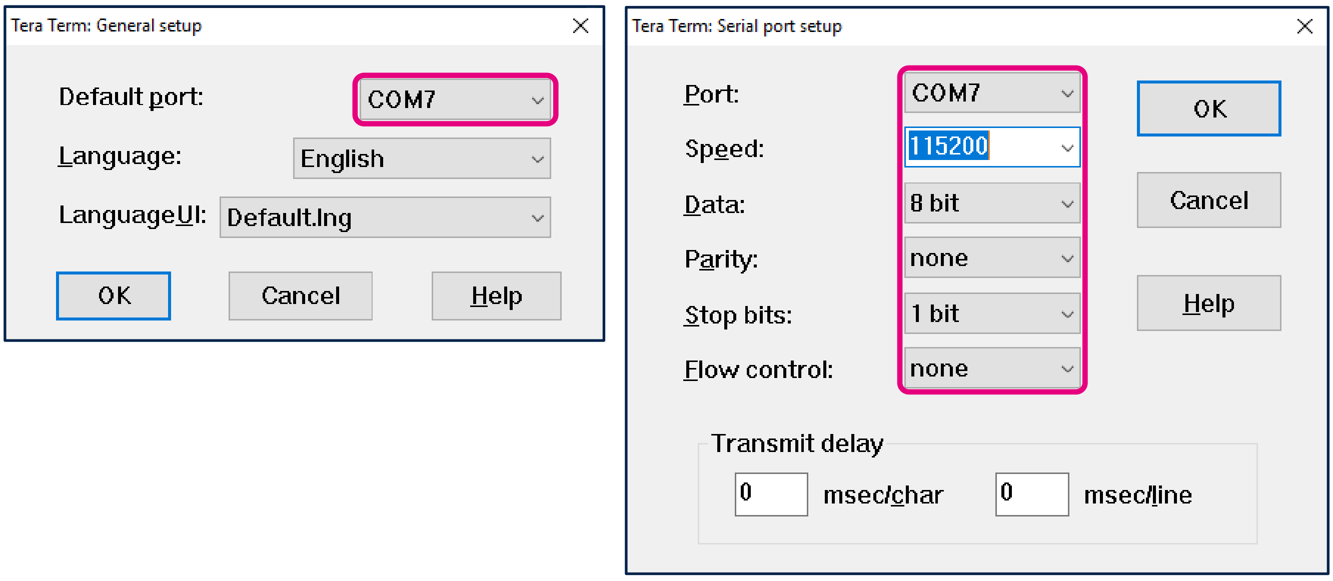
TCP/IP — Host: myhost.example.com; History; Service: Telnet / SSH / Other; TCP port#: 22; SSH version: SSH2; Protocol: UNSPEC

Serial — Port: COM7: STMicroelectronics STLink Virtu

8.3.3 Tera Term configuration

The Tera Term configuration is performed through the *General* and *Serial port* setup menus as illustrated on the figures below.

Figure 19. Tera Term setup screen



Tera Term: General setup

Default port: COM7
Language: English
LanguageUI: Default.lng

Tera Term: Serial port setup

Port: COM7
Speed: 115200
Data: 8 bit
Parity: none
Stop bits: 1 bit
Flow control: none

Transmit delay
0 msec/char 0 msec/line

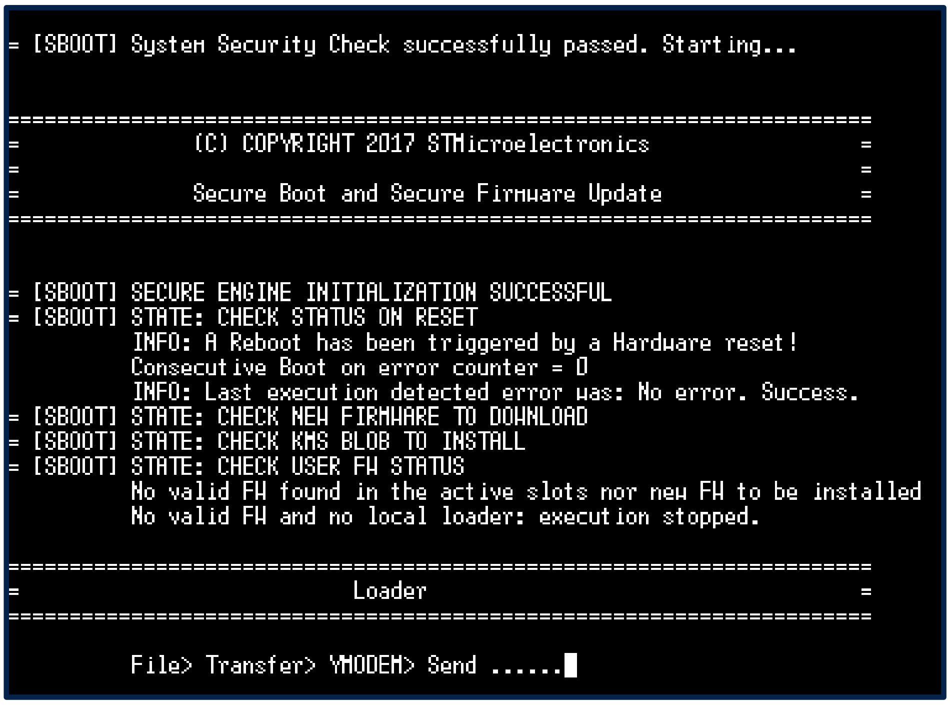
A configuration is saved using *Menu Setup / Save Setup*.

Note: After each plug / unplug of the USB cable, the Tera Term Serial port setup menu may have to be validated again to restart the connection. ***Press the Reset button to display the welcome screen.***

8.3.4 Welcome screen display

The welcome screen is displayed on the Tera Term as illustrated in the figure below.

Figure 20. SBSFU welcome screen display



```
= [SBOOT] System Security Check successfully passed. Starting...


======================================================================
=              (C) COPYRIGHT 2017 STMicroelectronics                 =
=                                                                    =
=              Secure Boot and Secure Firmware Update                =
======================================================================


= [SBOOT] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SBOOT] STATE: CHECK STATUS ON RESET
          INFO: A Reboot has been triggered by a Hardware reset!
          Consecutive Boot on error counter = 0
          INFO: Last execution detected error was: No error. Success.
= [SBOOT] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SBOOT] STATE: CHECK KMS BLOB TO INSTALL
= [SBOOT] STATE: CHECK USER FW STATUS
          No valid FW found in the active slots nor new FW to be installed
          No valid FW and no local loader: execution stopped.

======================================================================
=                            Loader                                  =
======================================================================

          File> Transfer> YMODEM> Send ......
```

8.4 SBSFU application execution

At each reboot, the application checks if the user has requested a new firmware download by keeping the User button pressed.

If there is no download request, the application checks the status of the user firmware

- Since the board was erased, no firmware is available
- The application cannot jump to firmware and goes back to check if there is a download request

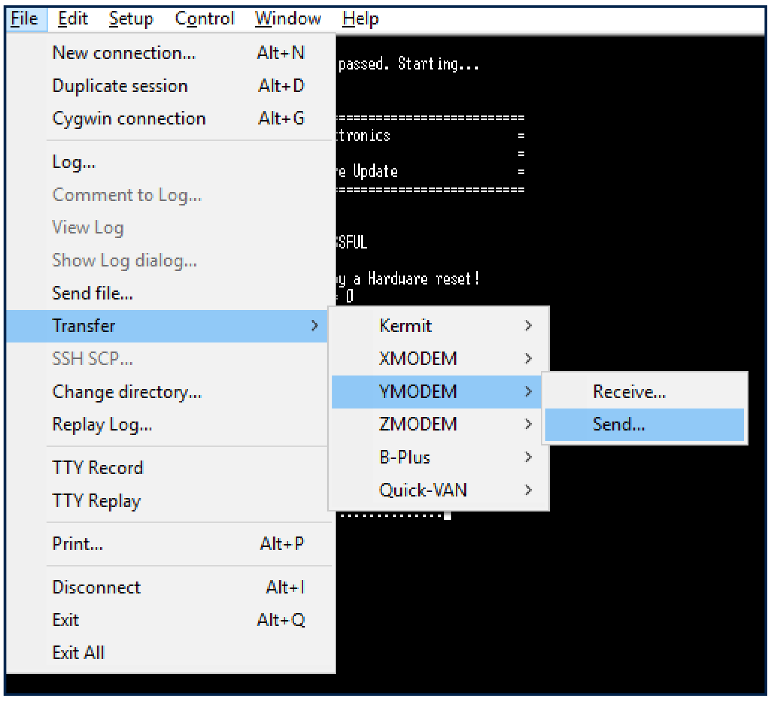
8.4.1 First download request

When no user firmware is present, SBSFU automatically waits for the download procedure to start. Otherwise, the forced download request is obtained by holding the User button on the STM32 Nucleo board during reset.

8.4.2 Send the first firmware

For sending the first firmware (**.sfb*), use the *File* > *Transfer* > *YMODEM* > *Send* menu in Tera Term as shown in the following figure. It can be the Cortex®-M4 or the Cortex®-M0+ firmware (no order required).

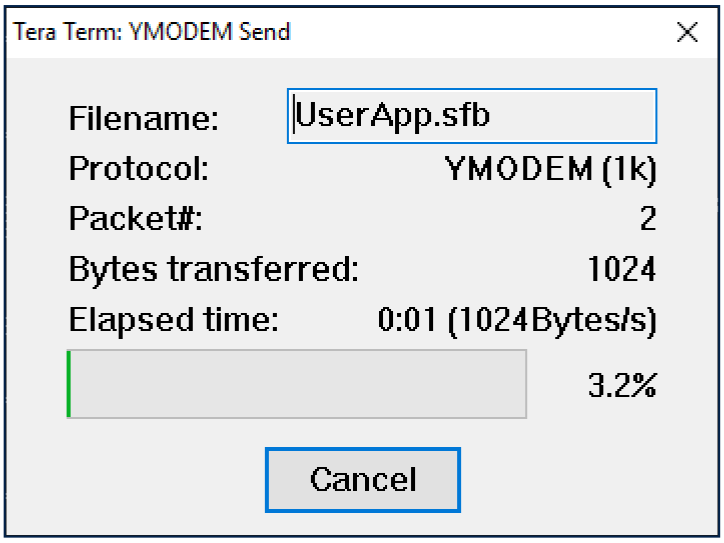
Figure 21. **First SBSFU encrypted firmware transfer start**



```
passed. Starting...

========================
tronics                =
                       =
e Update               =
========================


SFUL

y a Hardware reset!
 0
```

Once the *UserApp.sfb* file is selected, the Ymodem transfer starts. Transfer progress is reported as shown in the next figure.

Figure 22. **First SBSFU encrypted firmware transfer in progress**



Tera Term: YMODEM Send

Filename:	UserApp.sfb
Protocol:	YMODEM (1k)
Packet#:	2
Bytes transferred:	1024
Elapsed time:	0:01 (1024Bytes/s)
	3.2%

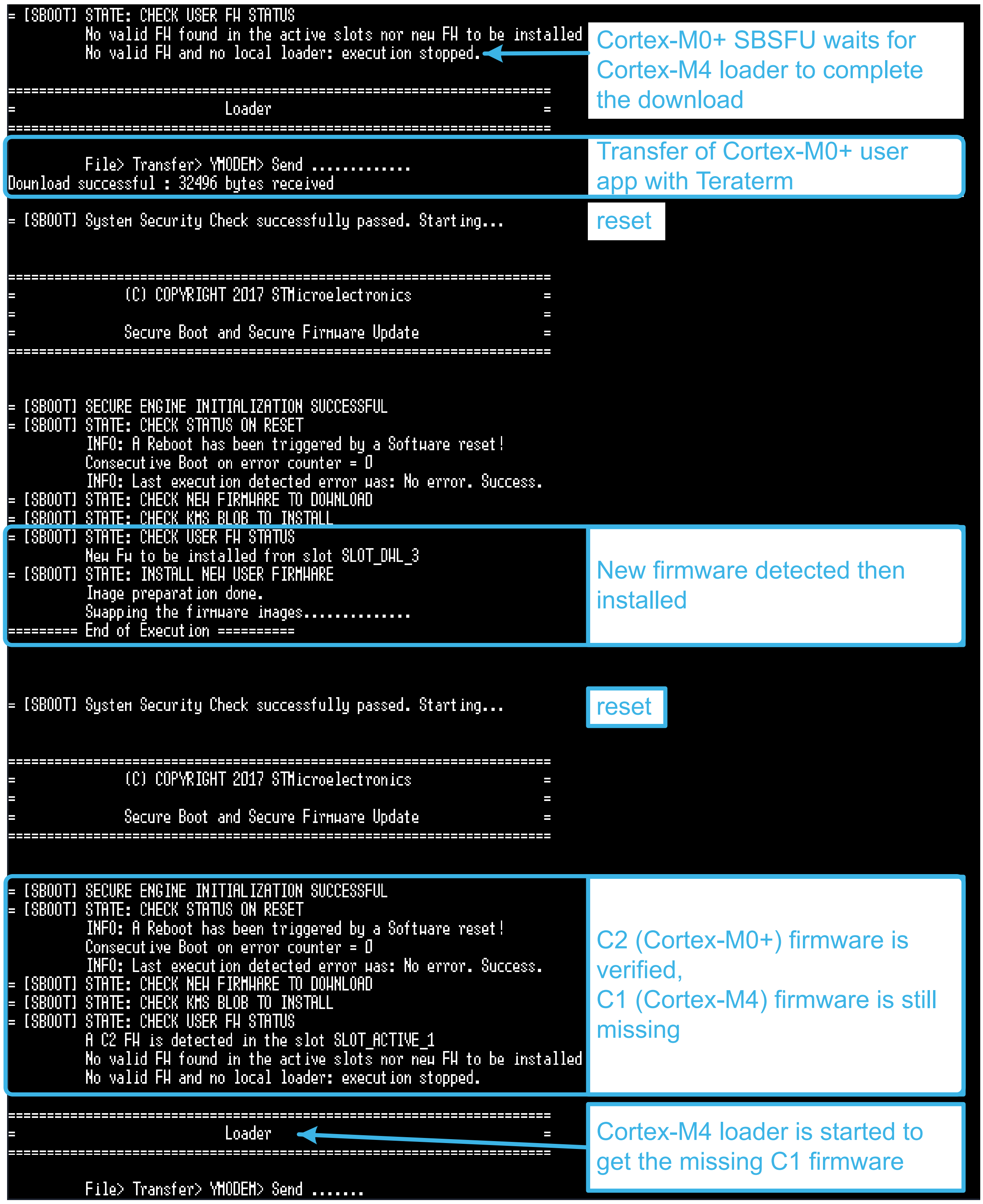
The progress gauge stalls for a short time at the beginning of the procedure while SBSFU verifies the firmware header validity and erases the Flash memory slot where the firmware image is downloaded.

8.4.3 First file transfer completion

This section presents the use case when there is no other firmware available in the Flash memory. Otherwise, the behavior is the one described in Section 8.4.6 Second file transfer completion.

After the file transfer is completed, the system forces a reboot as shown in the figure below.

Figure 23. **SBSFU reboot after first encrypted firmware transfer (dual-slot configuration)**



```
= [SBOOT] STATE: CHECK USER FW STATUS
        No valid FW found in the active slots nor new FW to be installed
        No valid FW and no local loader: execution stopped.

======================================================================
=                           Loader                                   =
======================================================================

        File> Transfer> YMODEM> Send .............
Download successful : 32496 bytes received

= [SBOOT] System Security Check successfully passed. Starting...


======================================================================
=              (C) COPYRIGHT 2017 STMicroelectronics                 =
=                                                                    =
=              Secure Boot and Secure Firmware Update                =
======================================================================


= [SBOOT] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SBOOT] STATE: CHECK STATUS ON RESET
        INFO: A Reboot has been triggered by a Software reset!
        Consecutive Boot on error counter = 0
        INFO: Last execution detected error was: No error. Success.
= [SBOOT] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SBOOT] STATE: CHECK KMS BLOB TO INSTALL
= [SBOOT] STATE: CHECK USER FW STATUS
        New Fw to be installed from slot SLOT_DWL_3
= [SBOOT] STATE: INSTALL NEW USER FIRMWARE
        Image preparation done.
        Swapping the firmware images..............
========= End of Execution ==========


= [SBOOT] System Security Check successfully passed. Starting...


======================================================================
=              (C) COPYRIGHT 2017 STMicroelectronics                 =
=                                                                    =
=              Secure Boot and Secure Firmware Update                =
======================================================================

= [SBOOT] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SBOOT] STATE: CHECK STATUS ON RESET
        INFO: A Reboot has been triggered by a Software reset!
        Consecutive Boot on error counter = 0
        INFO: Last execution detected error was: No error. Success.
= [SBOOT] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SBOOT] STATE: CHECK KMS BLOB TO INSTALL
= [SBOOT] STATE: CHECK USER FW STATUS
        A C2 FW is detected in the slot SLOT_ACTIVE_1
        No valid FW found in the active slots nor new FW to be installed
        No valid FW and no local loader: execution stopped.

======================================================================
=                           Loader                                   =
======================================================================

        File> Transfer> YMODEM> Send .......
```

Cortex-M0+ SBSFU waits for Cortex-M4 loader to complete the download
Transfer of Cortex-M0+ user app with Teraterm
reset
New firmware detected then installed
reset
C2 (Cortex-M0+) firmware is verified,
C1 (Cortex-M4) firmware is still missing
Cortex-M4 loader is started to get the missing C1 firmware

Note: *With the dual-core single-slot configuration, the loader is run directly by the Cortex$^{®}$-M0+.*

The system status that is printed as shown in the previous figure consequently provides the following information:

- There is no firmware to download
- The firmware is detected as encrypted. The user firmware is decrypted
- If the decryption is OK, the user firmware is installed
- If the installation is OK, the user firmware signature is verified
- If the verification is OK, the availability of the Cortex®-M4 firmware and of Cortex®-M0+ firmware is checked
- Here, one firmware is missing, a second download procedure is started.

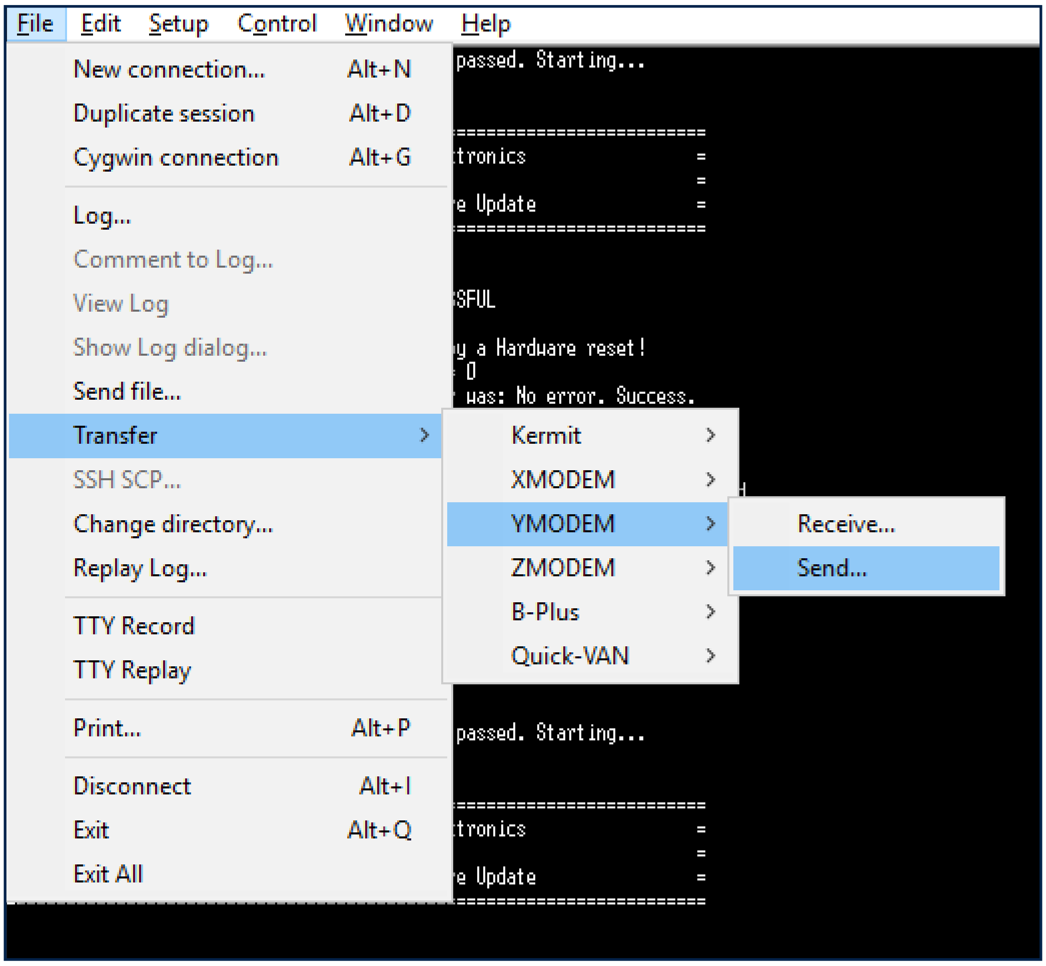
8.4.4 Second download request

When only one user firmware is present (Cortex®-M4 or Cortex®-M0+), SBSFU automatically waits for the download procedure to start. Otherwise, the forced download request is obtained by holding the User button on the STM32 Nucleo board during reset.

8.4.5 Send the second firmware

For sending the second firmware (**.sfb*), use the *File* > *Transfer* > *YMODEM* > *Send* menu in Tera Term as shown in the next figure. It can be the Cortex®-M4 or the Cortex®-M0+ firmware (no order required).

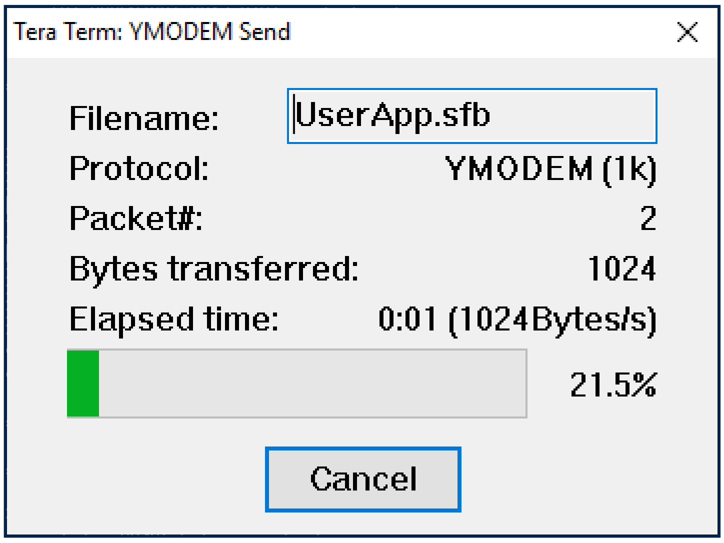
Figure 24. **Second SBSFU encrypted firmware transfer start**



```
passed. Starting...

========================
tronics               =
                      =
e Update              =
========================


SFUL

y a Hardware reset!
 0
 was: No error. Success.




passed. Starting...

========================
tronics               =
                      =
e Update              =
========================
```

Once the *UserApp.sfb* file is selected, the Ymodem transfer starts. Transfer progress is reported as shown in the next figure.

Figure 25. **Second SBSFU encrypted firmware transfer in progress**



Tera Term: YMODEM Send

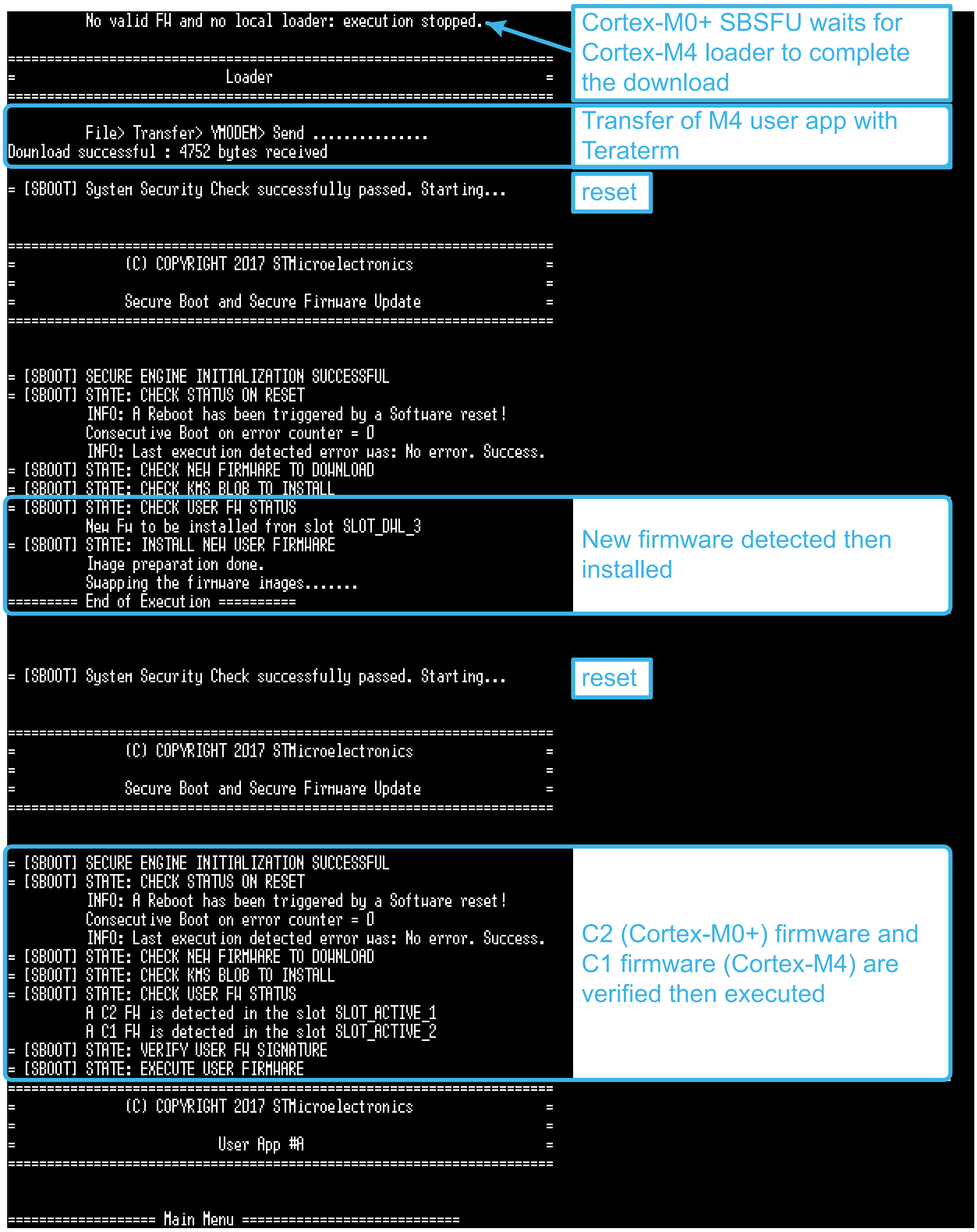
Filename: UserApp.sfb
Protocol: YMODEM (1k)
Packet#: 2
Bytes transferred: 1024
Elapsed time: 0:01 (1024Bytes/s)
21.5%

The progress gauge stalls for a short time at the beginning of the procedure while SBSFU verifies the firmware header validity and erases the Flash memory slot where the firmware image is downloaded.

8.4.6 Second file transfer completion

After the second file transfer is completed, the system forces a reboot as shown in the next figure.

Figure 26. **SBSFU reboot after encrypted firmware transfer (dual-slot configuration)**



```
          No valid FW and no local loader: execution stopped.

======================================================================
=                           Loader                                   =
======================================================================

          File> Transfer> YMODEM> Send ...............
Download successful : 4752 bytes received

= [SBOOT] System Security Check successfully passed. Starting...


======================================================================
=              (C) COPYRIGHT 2017 STMicroelectronics                 =
=                                                                    =
=              Secure Boot and Secure Firmware Update                =
======================================================================


= [SBOOT] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SBOOT] STATE: CHECK STATUS ON RESET
          INFO: A Reboot has been triggered by a Software reset!
          Consecutive Boot on error counter = 0
          INFO: Last execution detected error was: No error. Success.
= [SBOOT] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SBOOT] STATE: CHECK KMS BLOB TO INSTALL
= [SBOOT] STATE: CHECK USER FW STATUS
          New Fw to be installed from slot SLOT_DWL_3
= [SBOOT] STATE: INSTALL NEW USER FIRMWARE
          Image preparation done.
          Swapping the firmware images.......
========= End of Execution =========


= [SBOOT] System Security Check successfully passed. Starting...


======================================================================
=              (C) COPYRIGHT 2017 STMicroelectronics                 =
=                                                                    =
=              Secure Boot and Secure Firmware Update                =
======================================================================


= [SBOOT] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SBOOT] STATE: CHECK STATUS ON RESET
          INFO: A Reboot has been triggered by a Software reset!
          Consecutive Boot on error counter = 0
          INFO: Last execution detected error was: No error. Success.
= [SBOOT] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SBOOT] STATE: CHECK KMS BLOB TO INSTALL
= [SBOOT] STATE: CHECK USER FW STATUS
          A C2 FW is detected in the slot SLOT_ACTIVE_1
          A C1 FW is detected in the slot SLOT_ACTIVE_2
= [SBOOT] STATE: VERIFY USER FW SIGNATURE
= [SBOOT] STATE: EXECUTE USER FIRMWARE
======================================================================
=              (C) COPYRIGHT 2017 STMicroelectronics                 =
=                                                                    =
=                          User App #A                               =
======================================================================


================= Main Menu ===========================
```

Cortex-M0+ SBSFU waits for Cortex-M4 loader to complete the download
Transfer of M4 user app with Teraterm
reset
New firmware detected then installed
reset
C2 (Cortex-M0+) firmware and C1 firmware (Cortex-M4) are verified then executed

Note: With the dual-core single-slot configuration, the loader is run directly by the Cortex$^{®}$*-M0+.*

The system status that is printed as shown in the above figure, consequently provides the following information:

- There is no firmware to download
- The firmware is detected as encrypted. The user firmware is decrypted
- If the decryption is OK, the user firmware is installed
- If the installation is OK, the user firmware signature is verified
- If the verification is OK, the availability of the Cortex®-M4 firmware and of Cortex®-M0+ firmware is checked
- If the Cortex®-M4 firmware and the Cortex®-M0+ firmware are both available, they are executed

8.4.7 System restart

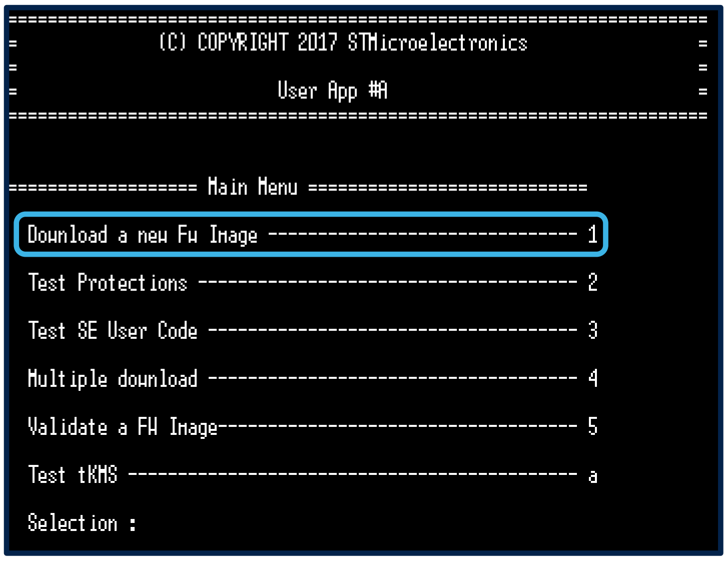
Pressing the Reset button forces the system to restart: the user applications are started by Cortex®-M4 SB and Cortex®-M0+ SBSFU.

Note: Holding the User button during reset, triggers the forced download state instead of the user application execution.

8.5 User application execution

The Cortex®-M0+ user application is executed according to the selection illustrated in the following figure and further described in the following sections.

Figure 27. **Cortex-M0+ user application execution**



```
======================================================================
=              (C) COPYRIGHT 2017 STMicroelectronics                 =
=                                                                    =
=                          User App #A                               =
======================================================================


================== Main Menu ============================

  Download a new Fw Image ------------------------------ 1

  Test Protections ------------------------------------- 2

  Test SE User Code ------------------------------------ 3

  Multiple download ------------------------------------ 4

  Validate a FW Image----------------------------------- 5

  Test tKMS -------------------------------------------- a

  Selection :
```

The Cortex®-M4 user application makes the blue LED blink.

8.5.1 Download a new firmware image (dual-slot configuration only)

The download of a new firmware image is performed through the same steps as those presented for SBSFU in Section 8.4 SBSFU application execution:

1. Send firmware
   - In Tera Term, click on *File>Transfer>YMODEM>Send*
   - Select *UserApp.sfb* (compiled as *UserApp#B*)
2. The system reboots
3. The Secure Boot state machine handles the new image
   - Firmware header is verified
   - Firmware is decrypted
   - Firmware is installed
   - Firmware signature is verified
   - Firmware is executed

The above procedure is also valid for the Cortex®-M4 *UserApp.sfb*.

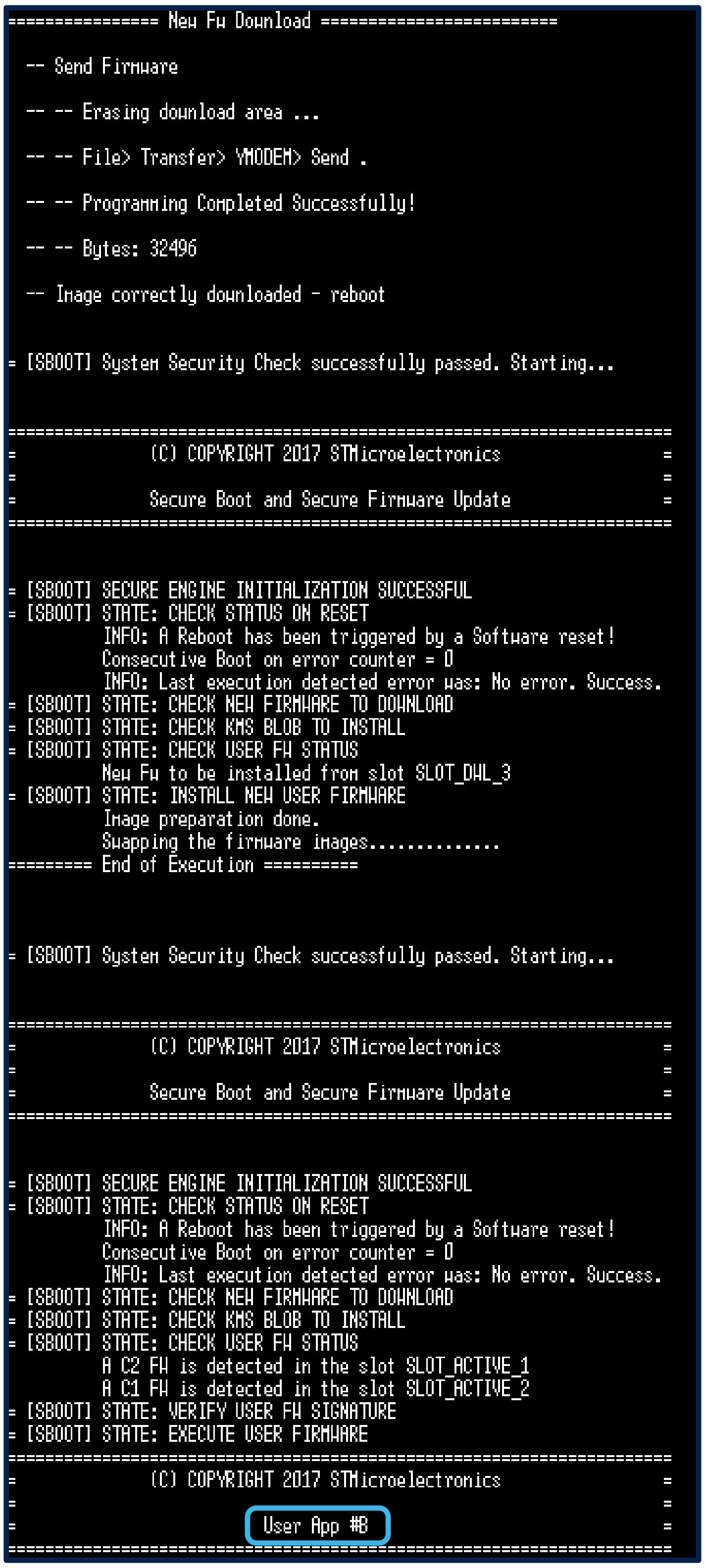


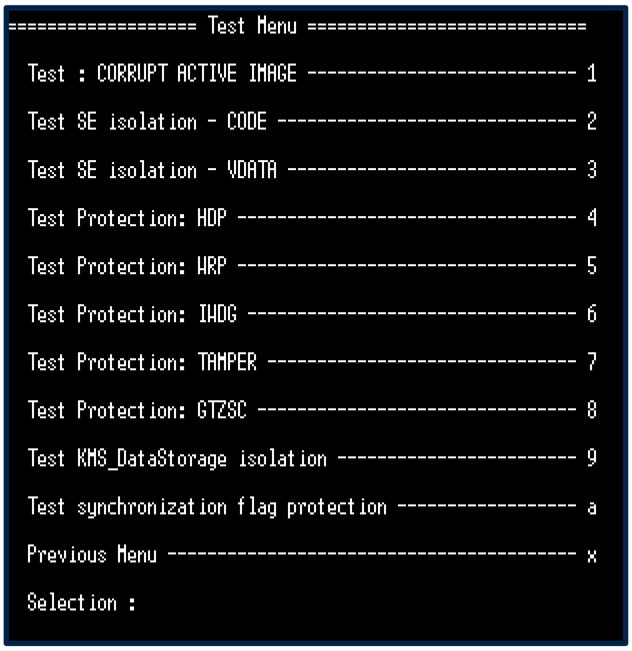
Figure 28. Encrypted firmware download via a user application

```
=============== New Fw Download ========================

  -- Send Firmware

  -- -- Erasing download area ...

  -- -- File> Transfer> YMODEM> Send .

  -- -- Programming Completed Successfully!

  -- -- Bytes: 32496

  -- Image correctly downloaded - reboot


= [SBOOT] System Security Check successfully passed. Starting...


======================================================================
=              (C) COPYRIGHT 2017 STMicroelectronics                 =
=                                                                    =
=              Secure Boot and Secure Firmware Update                =
======================================================================


= [SBOOT] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SBOOT] STATE: CHECK STATUS ON RESET
          INFO: A Reboot has been triggered by a Software reset!
          Consecutive Boot on error counter = 0
          INFO: Last execution detected error was: No error. Success.
= [SBOOT] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SBOOT] STATE: CHECK KMS BLOB TO INSTALL
= [SBOOT] STATE: CHECK USER FW STATUS
          New Fw to be installed from slot SLOT_DWL_3
= [SBOOT] STATE: INSTALL NEW USER FIRMWARE
          Image preparation done.
          Swapping the firmware images.............
========= End of Execution ==========



= [SBOOT] System Security Check successfully passed. Starting...


======================================================================
=              (C) COPYRIGHT 2017 STMicroelectronics                 =
=                                                                    =
=              Secure Boot and Secure Firmware Update                =
======================================================================


= [SBOOT] SECURE ENGINE INITIALIZATION SUCCESSFUL
= [SBOOT] STATE: CHECK STATUS ON RESET
          INFO: A Reboot has been triggered by a Software reset!
          Consecutive Boot on error counter = 0
          INFO: Last execution detected error was: No error. Success.
= [SBOOT] STATE: CHECK NEW FIRMWARE TO DOWNLOAD
= [SBOOT] STATE: CHECK KMS BLOB TO INSTALL
= [SBOOT] STATE: CHECK USER FW STATUS
          A C2 FW is detected in the slot SLOT_ACTIVE_1
          A C1 FW is detected in the slot SLOT_ACTIVE_2
= [SBOOT] STATE: VERIFY USER FW SIGNATURE
= [SBOOT] STATE: EXECUTE USER FIRMWARE
======================================================================
=              (C) COPYRIGHT 2017 STMicroelectronics                 =
=                                                                    =
=                          User App #B                               =
======================================================================
```

8.5.2 Test protections

The test protection menu is shown in the following figure:

Figure 29. Cortex-M0+ user application test protection menu



```
================== Test Menu ===========================

 Test : CORRUPT ACTIVE IMAGE -------------------------- 1

 Test SE isolation - CODE ----------------------------- 2

 Test SE isolation - VDATA ---------------------------- 3

 Test Protection: HDP --------------------------------- 4

 Test Protection: WRP --------------------------------- 5

 Test Protection: IWDG -------------------------------- 6

 Test Protection: TAMPER ------------------------------ 7

 Test Protection: GTZSC ------------------------------- 8

 Test KMS_DataStorage isolation ----------------------- 9

 Test synchronization flag protection ----------------- a

 Previous Menu ---------------------------------------- x

 Selection :
```

The test protection menu is printed at each test attempt of a prohibited operation or error injection as a function of the test run:

- CORRUPT ACTIVE IMAGE test (#1)
  - Causes a signature verification failure at the next boot.
- SE isolation tests (#2, #3)
  - Causes a reset trying to access protected code or data (either in RAM or Flash memory)
- HDP test (#4)
  - Causes an error trying to access the HDP region protecting the keys
- WRP test (#5)
  - Causes an error trying to erase write protected code
- IWDG test (#6)
  - Causes a reset simulating a deadlock by not refreshing the watchdog
- TAMPER test (#7)
  - Causes a reset if a tamper event is detected
  - To generate a tamper event, the user must connect PC13 (CN7.23) to GND (It may be enough to put a finger close to CN7.23).
- GTZC tests (#8)
  - Causes a reset trying to access protected resources (RAM, Flash memory, or IPs)
- KMS_DataStorage protection test (#9)
  - Causes an IWDG reset trying to access protected KMS data (Flash memory)
- Synchronization flag protection test (#a)
  - Causes a reset trying to access protected data (in RAM)

Returning to the previous menu is obtained by pressing the *x* key.

8.5.3 Test Secure Engine user code

The version and size of the current user firmware are retrieved using a Secure Engine service and printed in the console.

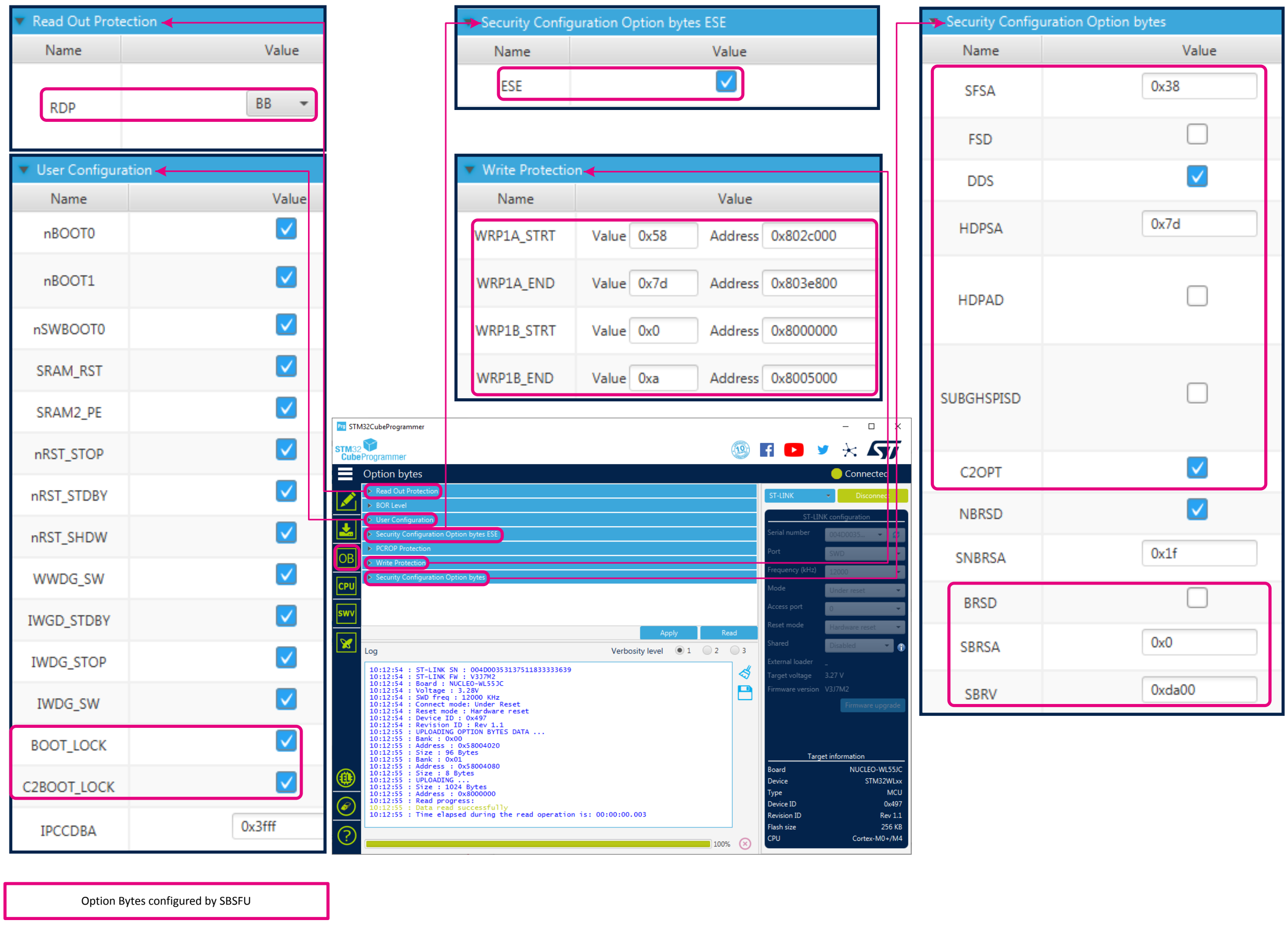
8.6 Programming a new software when the securities are activated

After flashing binary in Flash memory with all securities enabled it is not possible anymore to update SBSFU directly. You need first to disable the securities preventing the erasure of the SBSFU software.

Launch the STM32CubeProgrammer application and open the *option bytes* menu.

The *option bytes* menu looks like this (RDP Level 1, WRP protection and secure mode):

Figure 30. Option Bytes menu



Read Out Protection

Name	Value
RDP	BB

User Configuration

Name	Value
nBOOT0	✓
nBOOT1	✓
nSWBOOT0	✓
SRAM_RST	✓
SRAM2_PE	✓
nRST_STOP	✓
nRST_STDBY	✓
nRST_SHDW	✓
WWDG_SW	✓
IWGD_STDBY	✓
IWDG_STOP	✓
IWDG_SW	✓
BOOT_LOCK	✓
C2BOOT_LOCK	✓
IPCCDBA	0x3fff

Security Configuration Option bytes ESE

Name	Value
ESE	✓

Write Protection

Name	Value			
WRP1A_STRT	Value	0x58	Address	0x802c000
WRP1A_END	Value	0x7d	Address	0x803e800
WRP1B_STRT	Value	0x0	Address	0x8000000
WRP1B_END	Value	0xa	Address	0x8005000

Security Configuration Option bytes

Name	Value
SFSA	0x38
FSD	☐
DDS	✓
HDPSA	0x7d
HDPAD	☐
SUBGHSPISD	☐
C2OPT	✓
NBRSD	✓
SNBRSA	0x1f
BRSD	☐
SBRSA	0x0
SBRV	0xda00

ST-LINK configuration

Serial number	004D0035...
Port	SWD
Frequency (kHz)	12000
Mode	Under reset
Access port	0
Reset mode	Hardware reset
Shared	Disabled
External loader	_
Target voltage	3.27 V
Firmware version	V3J7M2

Target information

Board	NUCLEO-WL55JC
Device	STM32WLxx
Type	MCU
Device ID	0x497
Revision ID	Rev 1.1
Flash size	256 KB
CPU	Cortex-M0+/M4

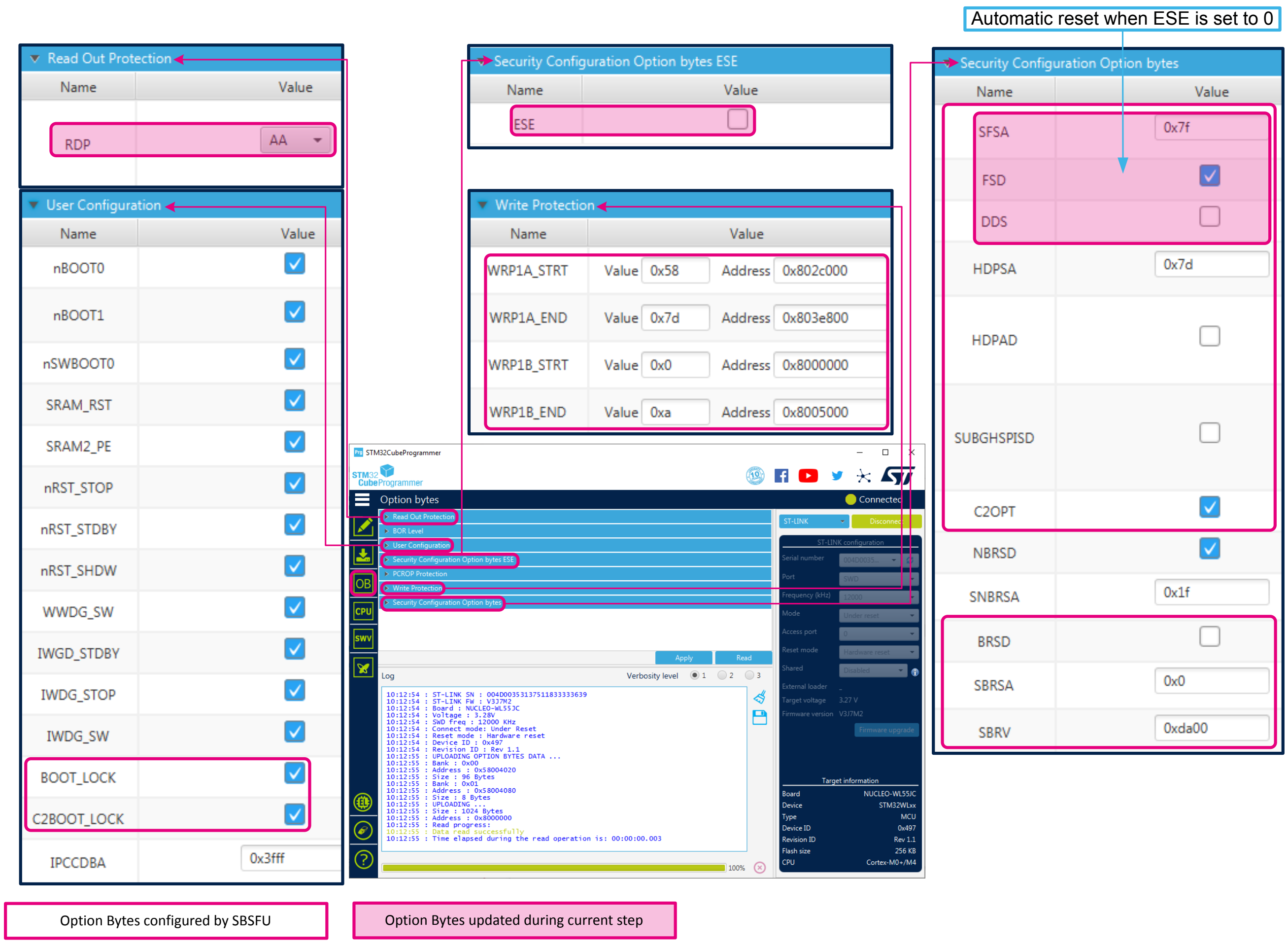
Log — Verbosity level [✓ 1] 2 3

```
10:12:54 : ST-LINK SN : 004D00353137511833333639
10:12:54 : ST-LINK FW : V3J7M2
10:12:54 : Board : NUCLEO-WL55JC
10:12:54 : Voltage : 3.28V
10:12:54 : SWD freq : 12000 KHz
10:12:54 : Connect mode: Under Reset
10:12:54 : Reset mode : Hardware reset
10:12:54 : Device ID : 0x497
10:12:54 : Revision ID : Rev 1.1
10:12:55 : UPLOADING OPTION BYTES DATA ...
10:12:55 : Bank : 0x00
10:12:55 : Address : 0x58004020
10:12:55 : Size : 96 Bytes
10:12:55 : Bank : 0x01
10:12:55 : Address : 0x58004080
10:12:55 : Size : 8 Bytes
10:12:55 : UPLOADING ...
10:12:55 : Size : 1024 Bytes
10:12:55 : Address : 0x8000000
10:12:55 : Read progress:
10:12:55 : Data read successfully
10:12:55 : Time elapsed during the read operation is: 00:00:00.003
```

100%

Update the fields as follows. Set RDP to level 0 and uncheck ESE. Then click on *Apply* for mass deletion of the Flash memory to occur:

Figure 31. Secure mode disable and FLASH memory mass delete


Automatic reset when ESE is set to 0
Option Bytes configured by SBSFU
Option Bytes updated during current step
STM32CubeProgrammer
Option bytes
Connected
Read Out Protection
BOR Level
User Configuration
Security Configuration Option bytes ESE
PCROP Protection
Write Protection
Security Configuration Option bytes
Apply
Read
Log
Verbosity level
ST-LINK
Disconnect
Firmware upgrade

Read Out Protection

Name	Value
RDP	AA

User Configuration

Name	Value
nBOOT0	✓
nBOOT1	✓
nSWBOOT0	✓
SRAM_RST	✓
SRAM2_PE	✓
nRST_STOP	✓
nRST_STDBY	✓
nRST_SHDW	✓
WWDG_SW	✓
IWGD_STDBY	✓
IWDG_STOP	✓
IWDG_SW	✓
BOOT_LOCK	✓
C2BOOT_LOCK	✓
IPCCDBA	0x3fff

Security Configuration Option bytes ESE

Name	Value
ESE	☐

Write Protection

Name	Value			
WRP1A_STRT	Value	0x58	Address	0x802c000
WRP1A_END	Value	0x7d	Address	0x803e800
WRP1B_STRT	Value	0x0	Address	0x8000000
WRP1B_END	Value	0xa	Address	0x8005000

Security Configuration Option bytes

Name	Value
SFSA	0x7f
FSD	✓
DDS	☐
HDPSA	0x7d
HDPAD	☐
SUBGHSPISD	☐
C2OPT	✓
NBRSD	✓
SNBRSA	0x1f
BRSD	☐
SBRSA	0x0
SBRV	0xda00

Verbosity level [✓ 1] 2 3

```
10:12:54 : ST-LINK SN : 004D00353137511833333639
10:12:54 : ST-LINK FW : V3J7M2
10:12:54 : Board : NUCLEO-WL55JC
10:12:54 : Voltage : 3.28V
10:12:54 : SWD freq : 12000 KHz
10:12:54 : Connect mode: Under Reset
10:12:54 : Reset mode : Hardware reset
10:12:54 : Device ID : 0x497
10:12:54 : Revision ID : Rev 1.1
10:12:55 : UPLOADING OPTION BYTES DATA ...
10:12:55 : Bank : 0x00
10:12:55 : Address : 0x58004020
10:12:55 : Size : 96 Bytes
10:12:55 : Bank : 0x01
10:12:55 : Address : 0x58004080
10:12:55 : Size : 8 Bytes
10:12:55 : UPLOADING ...
10:12:55 : Size : 1024 Bytes
10:12:55 : Address : 0x8000000
10:12:55 : Read progress:
10:12:55 : Data read successfully
10:12:55 : Time elapsed during the read operation is: 00:00:00.003
```

100%

ST-LINK configuration

Serial number	004D0035...
Port	SWD
Frequency (kHz)	12000
Mode	Under reset
Access port	0
Reset mode	Hardware reset
Shared	Disabled
External loader	_
Target voltage	3.27 V
Firmware version	V3J7M2

Target information

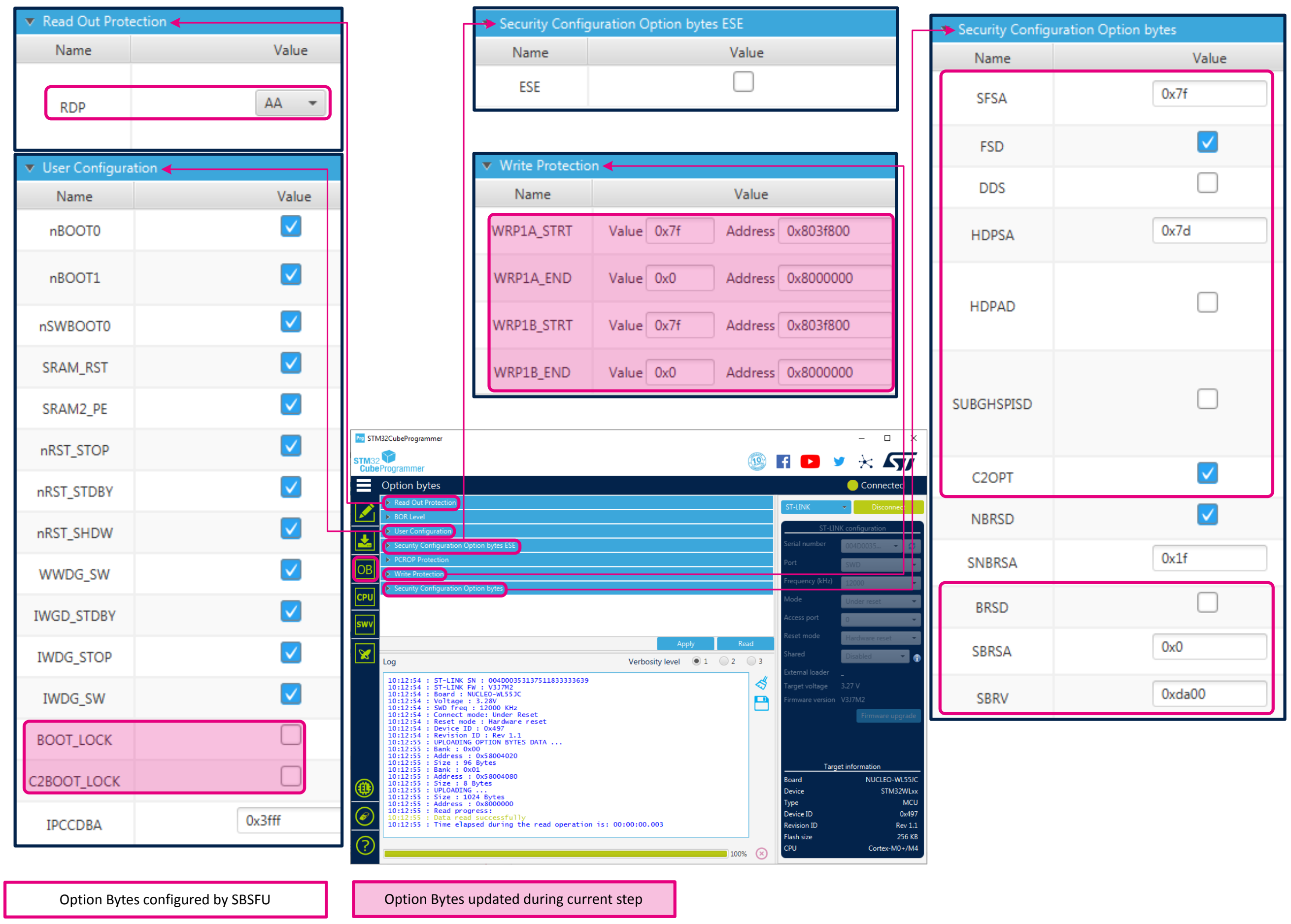
Board	NUCLEO-WL55JC
Device	STM32WLxx
Type	MCU
Device ID	0x497
Revision ID	Rev 1.1
Flash size	256 KB
CPU	Cortex-M0+/M4

After this step, the Flash memory is fully erased but the write protection is still enabled.

Update the fields as follows. Set the WRP start address to `0x7f` and the WRP end address to `0x0`.

Uncheck C2_BOOT_LOCK if the Cortex$^{®}$-M0+ boot address changes and BOOT_LOCK to return to a default configuration. Then click on *Apply*:

Figure 32. Boot lock and Write protection removal



Read Out Protection

Name	Value
RDP	AA

User Configuration

Name	Value
nBOOT0	✓
nBOOT1	✓
nSWBOOT0	✓
SRAM_RST	✓
SRAM2_PE	✓
nRST_STOP	✓
nRST_STDBY	✓
nRST_SHDW	✓
WWDG_SW	✓
IWGD_STDBY	✓
IWDG_STOP	✓
IWDG_SW	✓
BOOT_LOCK	☐
C2BOOT_LOCK	☐
IPCCDBA	0x3fff

Security Configuration Option bytes ESE

Name	Value
ESE	☐

Write Protection

Name	Value			
WRP1A_STRT	Value	0x7f	Address	0x803f800
WRP1A_END	Value	0x0	Address	0x8000000
WRP1B_STRT	Value	0x7f	Address	0x803f800
WRP1B_END	Value	0x0	Address	0x8000000

Security Configuration Option bytes

Name	Value
SFSA	0x7f
FSD	✓
DDS	☐
HDPSA	0x7d
HDPAD	☐
SUBGHSPISD	☐
C2OPT	✓
NBRSD	✓
SNBRSA	0x1f
BRSD	☐
SBRSA	0x0
SBRV	0xda00

```
10:12:54 : ST-LINK SN : 004D00353137511833333639
10:12:54 : ST-LINK FW : V3J7M2
10:12:54 : Board : NUCLEO-WL55JC
10:12:54 : Voltage : 3.28V
10:12:54 : SWD freq : 12000 KHz
10:12:54 : Connect mode: Under Reset
10:12:54 : Reset mode : Hardware reset
10:12:54 : Device ID : 0x497
10:12:54 : Revision ID : Rev 1.1
10:12:55 : UPLOADING OPTION BYTES DATA ...
10:12:55 : Bank : 0x00
10:12:55 : Address : 0x58004020
10:12:55 : Size : 96 Bytes
10:12:55 : Bank : 0x01
10:12:55 : Address : 0x58004080
10:12:55 : Size : 8 Bytes
10:12:55 : UPLOADING ...
10:12:55 : Size : 1024 Bytes
10:12:55 : Address : 0x8000000
10:12:55 : Read progress:
10:12:55 : Data read successfully
10:12:55 : Time elapsed during the read operation is: 00:00:00.003
```

ST-LINK configuration

Serial number	004D0035...
Port	SWD
Frequency (kHz)	12000
Mode	Under reset
Access port	0
Reset mode	Hardware reset
Shared	Disabled
External loader	_
Target voltage	3.27 V
Firmware version	V3J7M2

Target information

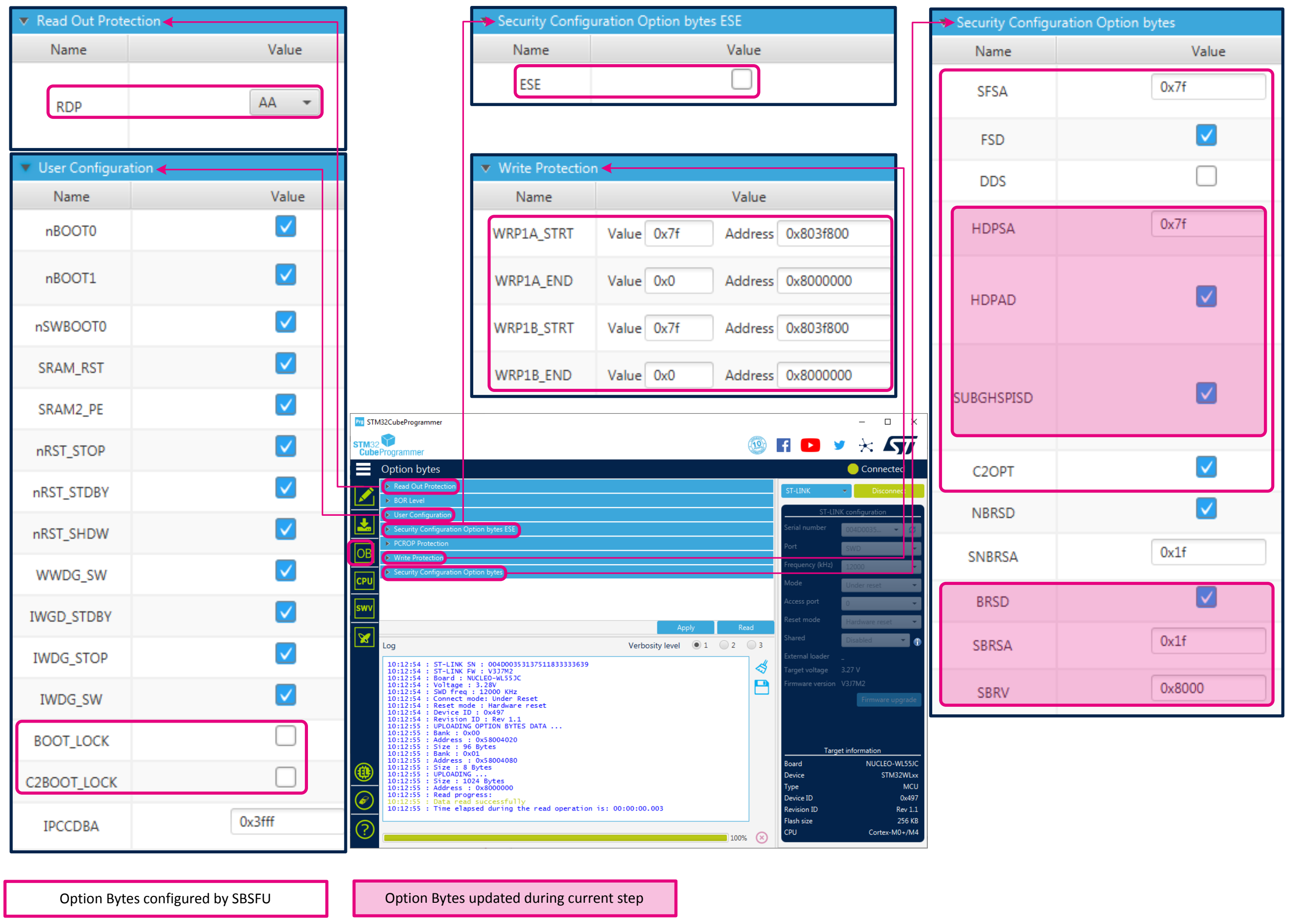
Board	NUCLEO-WL55JC
Device	STM32WLxx
Type	MCU
Device ID	0x497
Revision ID	Rev 1.1
Flash size	256 KB
CPU	Cortex-M0+/M4

Option Bytes configured by SBSFU

Option Bytes updated during current step

Optionally, to get back to the default configuration, all the highlighted option bytes must be set to the values shown in the figure below. Then click on *Apply*:

Figure 33. Optional changes



Read Out Protection

Name	Value
RDP	AA

User Configuration

Name	Value
nBOOT0	✓
nBOOT1	✓
nSWBOOT0	✓
SRAM_RST	✓
SRAM2_PE	✓
nRST_STOP	✓
nRST_STDBY	✓
nRST_SHDW	✓
WWDG_SW	✓
IWGD_STDBY	✓
IWDG_STOP	✓
IWDG_SW	✓
BOOT_LOCK	☐
C2BOOT_LOCK	☐
IPCCDBA	0x3fff

Security Configuration Option bytes ESE

Name	Value
ESE	☐

Write Protection

Name	Value			
WRP1A_STRT	Value	0x7f	Address	0x803f800
WRP1A_END	Value	0x0	Address	0x8000000
WRP1B_STRT	Value	0x7f	Address	0x803f800
WRP1B_END	Value	0x0	Address	0x8000000

Security Configuration Option bytes

Name	Value
SFSA	0x7f
FSD	✓
DDS	☐
HDPSA	0x7f
HDPAD	✓
SUBGHSPISD	✓
C2OPT	✓
NBRSD	✓
SNBRSA	0x1f
BRSD	✓
SBRSA	0x1f
SBRV	0x8000

Serial number	004D0035...
Port	SWD
Frequency (kHz)	12000
Mode	Under reset
Access port	0
Reset mode	Hardware reset
Shared	Disabled
External loader	-
Target voltage	3.27 V
Firmware version	V3J7M2

Board	NUCLEO-WL55JC
Device	STM32WLxx
Type	MCU
Device ID	0x497
Revision ID	Rev 1.1
Flash size	256 KB
CPU	Cortex-M0+/M4

```
10:12:54 : ST-LINK SN : 004D00353137511833333639
10:12:54 : ST-LINK FW : V3J7M2
10:12:54 : Board : NUCLEO-WL55JC
10:12:54 : Voltage : 3.28V
10:12:54 : SWD freq : 12000 KHz
10:12:54 : Connect mode: Under Reset
10:12:54 : Reset mode : Hardware reset
10:12:54 : Device ID : 0x497
10:12:54 : Revision ID : Rev 1.1
10:12:55 : UPLOADING OPTION BYTES DATA ...
10:12:55 : Bank : 0x00
10:12:55 : Address : 0x58004020
10:12:55 : Size : 96 Bytes
10:12:55 : Bank : 0x01
10:12:55 : Address : 0x58004080
10:12:55 : Size : 8 Bytes
10:12:55 : UPLOADING ...
10:12:55 : Size : 1024 Bytes
10:12:55 : Address : 0x8000000
10:12:55 : Read progress:
10:12:55 : Data read successfully
10:12:55 : Time elapsed during the read operation is: 00:00:00.003
```

☐	Option Bytes configured by SBSFU
▨	Option Bytes updated during current step

After this step, the software can be programmed again.

9 Understanding the last execution status message at boot-up

The following table lists the main error messages together with their explanation.

Table 5. **Error messages at boot-up**

Error message	Meaning
No error. Success.	No problem is encountered.
Watchdog error.	Watchdog expiry: the processing is too long and the watchdog has not been reloaded in due time.
Memory fault.	Memory fault is reported by the MPU fault handler.
Hard fault.	Arm® Cortex®-M hard fault exception.
Tampering fault.	Tamper-detection report.
Check protections error.	Not used in the example code. This can be used to log errors when doing the periodic verification of applied protection mechanisms.
Check status on reset error.	Error encountered while checking the status at boot (generic error).
Check new user firmware to download error.	Error encountered while checking if there is a local download request (generic error).
Download new user firmware error.	Error encountered while performing a local download (generic error).
Verify user firmware status error.	Error encountered while verifying the status of the user firmware. This error is reached when the Flash memory state does not allow determining the firmware status (generic error).
Decrypt user firmware error.	Not used in the current example code. This can be used to log generic errors related to the decrypt of a firmware. In the example, a more specific error is used: “Decrypt failure.”
Install user firmware error.	Error encountered during the installation of new firmware (generic error).
Verify user FW signature.	Error encountered while verifying the signature of the active firmware. In the example code, the signature is already checked during the firmware status check so this error is not supposed to be reported.
Resume firmware installation error.	Error encountered during the firmware installation procedure.
Execute user firmware error.	Error encountered while trying to launch the active firmware (generic error).
SE lock cannot be set.	Error encountered while trying to configure Secure Engine in “Firmware execution” mode (unprivileged mode) before starting the active firmware.
Firmware too big.	This error means that during a local download procedure the header indicated a firmware size bigger than the capacity of the download slot.
Ymodem com failure.	During a local download procedure, the download operation did not complete successfully (Ymodem protocol issue).
Header authentication failed.	During a local download procedure, the header could not be authenticated successfully. This error is reached only if the header stored in RAM is altered (otherwise the download is bypassed without triggering a critical failure).
Decrypt failure.	Error encountered while decrypting the content of the download slot. This error reports a decryption or authentication issue as the final stage of the decryption is a check of the signature.
Signature check failure.	Error encountered while verifying the signature of the decrypted firmware during an installation procedure. In the example code, this error may not be reached as a signature issue may be captured at the decrypt stage (reporting “Decrypt failure.”).
Flash memory error.	Flash memory error is encountered during an installation procedure.
FWIMG pattern issue.	Error encountered during an installation procedure: internal issue while writing some SBSFU patterns.

Error message	Meaning
Error while swapping the images in active and download slots.	Error encountered during an installation procedure: failure while swapping the images (previous firmware and decrypted firmware).
Firmware version rejected by anti rollback check.	Error encountered during an installation procedure: the firmware version cannot be accepted (newer firmware already installed or lower version than min. allowed version).
Unknown error.	Undocumented error (unexpected exception or unexpected state machine issue).

Appendix A Secure Engine protected environment

The Secure Engine (SE) concept defines a protected enclave exporting a set of secure functions executed in a trusted environment.

The following functionalities are provided by SE to the SBSFU application example:

- Secure Engine initialization function
- Secure cryptographic functions
  - AES-GCM and AES-CBC decryption
  - SHA256 HASH and ECDSA verification
  - Sensitive data (secret key, AES context) never leaves the protected environment and cannot be accessed from unprotected code
- Secure read/write access to firmware image Information
  - Read and write operation on a protected Flash memory area
  - Access to this area is allowed only to protected code
- Secure service to lock some functions in Secure Engine
  - One-way lock mechanism: once locked, no way to unlock it except via a system reset
  - Once locked, functions execution is no more possible via the call gate mechanism
  - Functionalities that are locked via the lock mechanism in Secure Engine example:
    - Secure Engine initialization function
    - Secure Encryption functions with OEM key
    - Secure read/write access to firmware image information
    - Secure service to lock some functions in Secure Engine

Note: Functionalities exported by SE can be extended depending on final user applications needs.

With KMS, SE functionalities are extended with the secure key management services providing cryptographic services to the user application through the PKCS#11 APIs (KEY ID-based APIs) are executed inside the Secure Engine protected environment in the dual-core configuration. User application keys are stored in this protected/ isolated environment. In the single-core configuration, the same services are offered but they are not executed inside a protected and isolated environment.

To deal with the GTZC call gate mechanism and to provide the user with a set of secure APIs, SE is designed with a two-level architecture, composed of Secure Engine Core and SE interface.

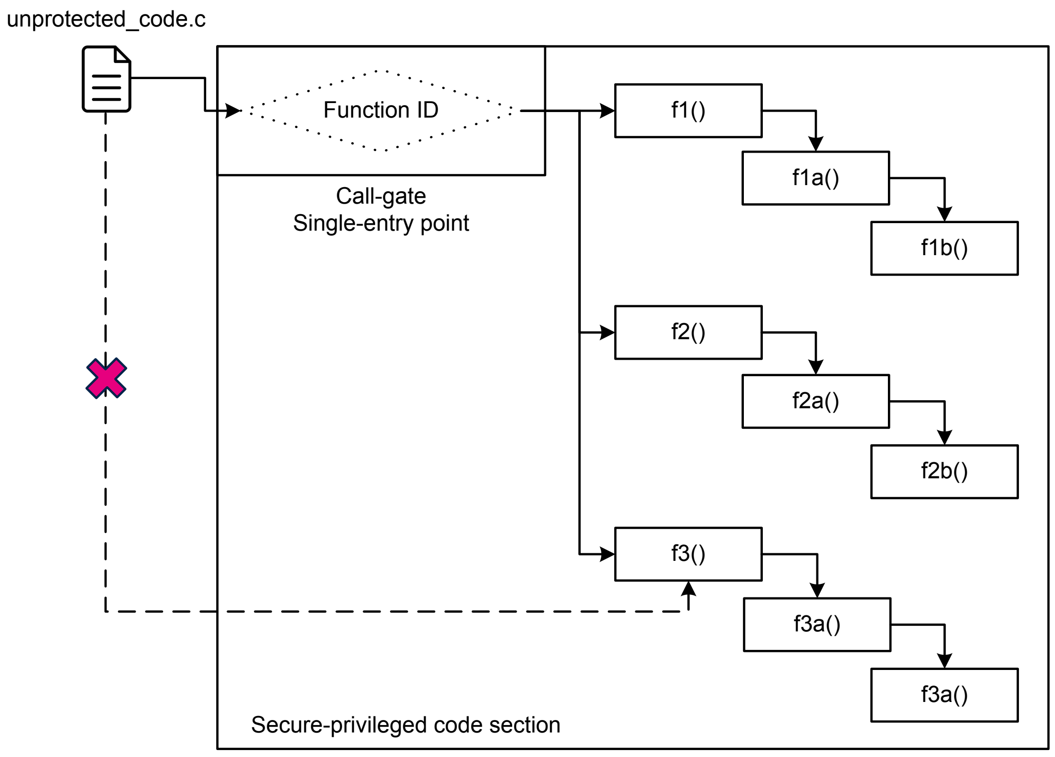
A.1 Overview of Secure Engine isolation

A.1.1 Secure Engine Core call-gate mechanism

The SE secure privileged world is entered using a specific "call gate" mechanism: a single entry point (placed at the second word of the code segment base address) must be used to open the gate and to execute the code protected by the security mechanisms. If the secure privileged code is accessed without passing through the call-gate mechanism then a system reset is generated.

As the only way to respect the call-gate sequence is to pass through the single call-gate entry point, therefore, if the application requires to have multiple functions located in the secure privileged area called from secure unprivileged code (such as encrypt and decrypt functions), a way to select which of the internal functions to execute is needed. A solution is to use a parameter to specify which function to execute, for instance, CallGate(F1_ID), CallGate(F2_ID), and so on. According to the parameter, the right function is internally called.

Figure 34. **Call-gate mechanism**



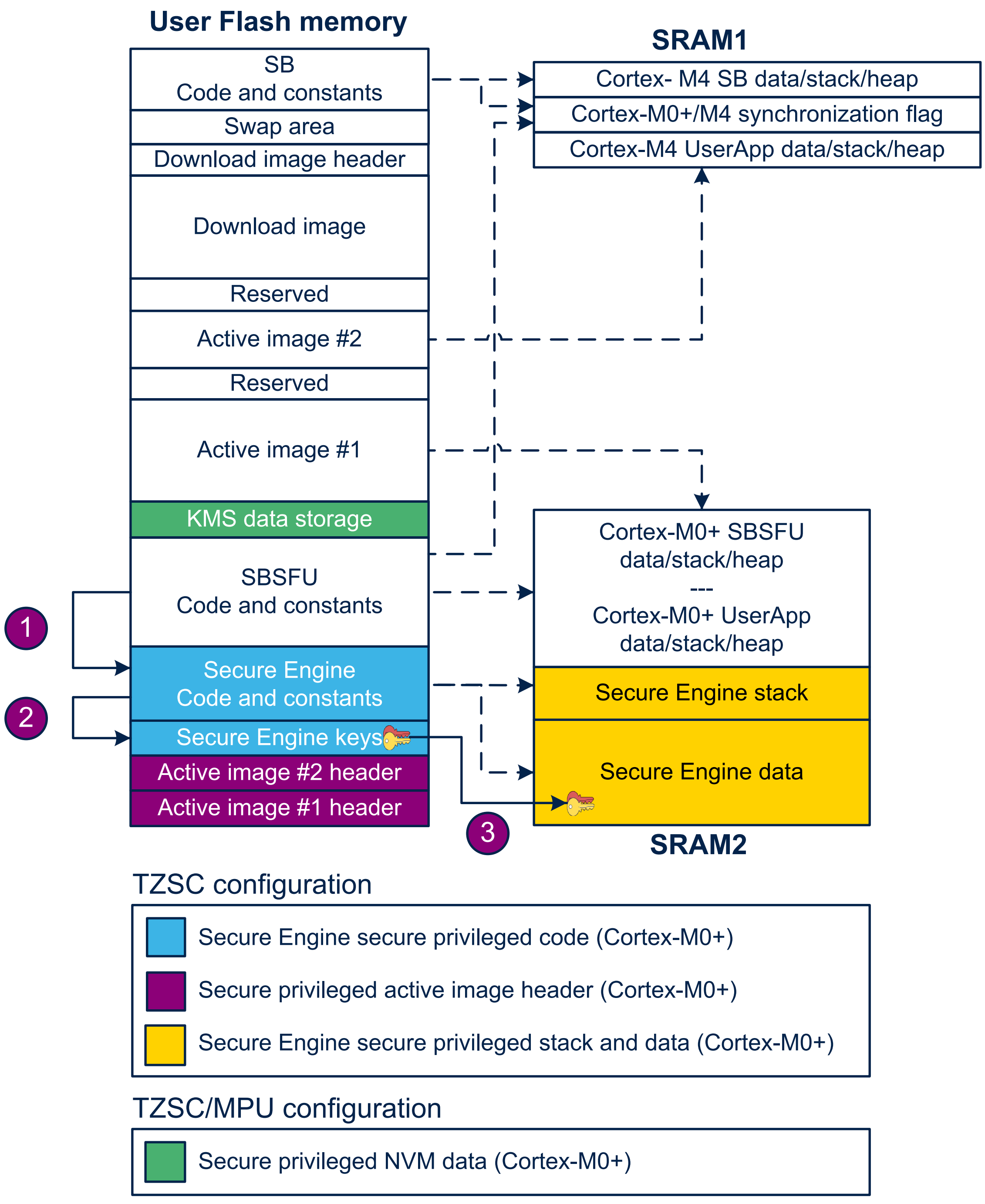
Note: *The code section must include all the code executed. For instance, if the call sequence is callgate->f1()->f1a()->f1b(), all the three functions f1(), f1a() and f1b() must be included in the secure privileged code section.*

The figure below shows the steps to perform cryptographic operations (that require access to the key) to respect the call gate mechanism.

For the cryptographic functions:

1. The SBSFU code calls the call-gate function to execute protected code
2. The call-gate function checks parameters and securities and then calls the requested crypto function
3. The SE crypto function can access the keys, copies them into the protected section of SRAM2 and then uses them in the cryptographic operations.

Figure 35. **Secure Engine call-gate mechanism**

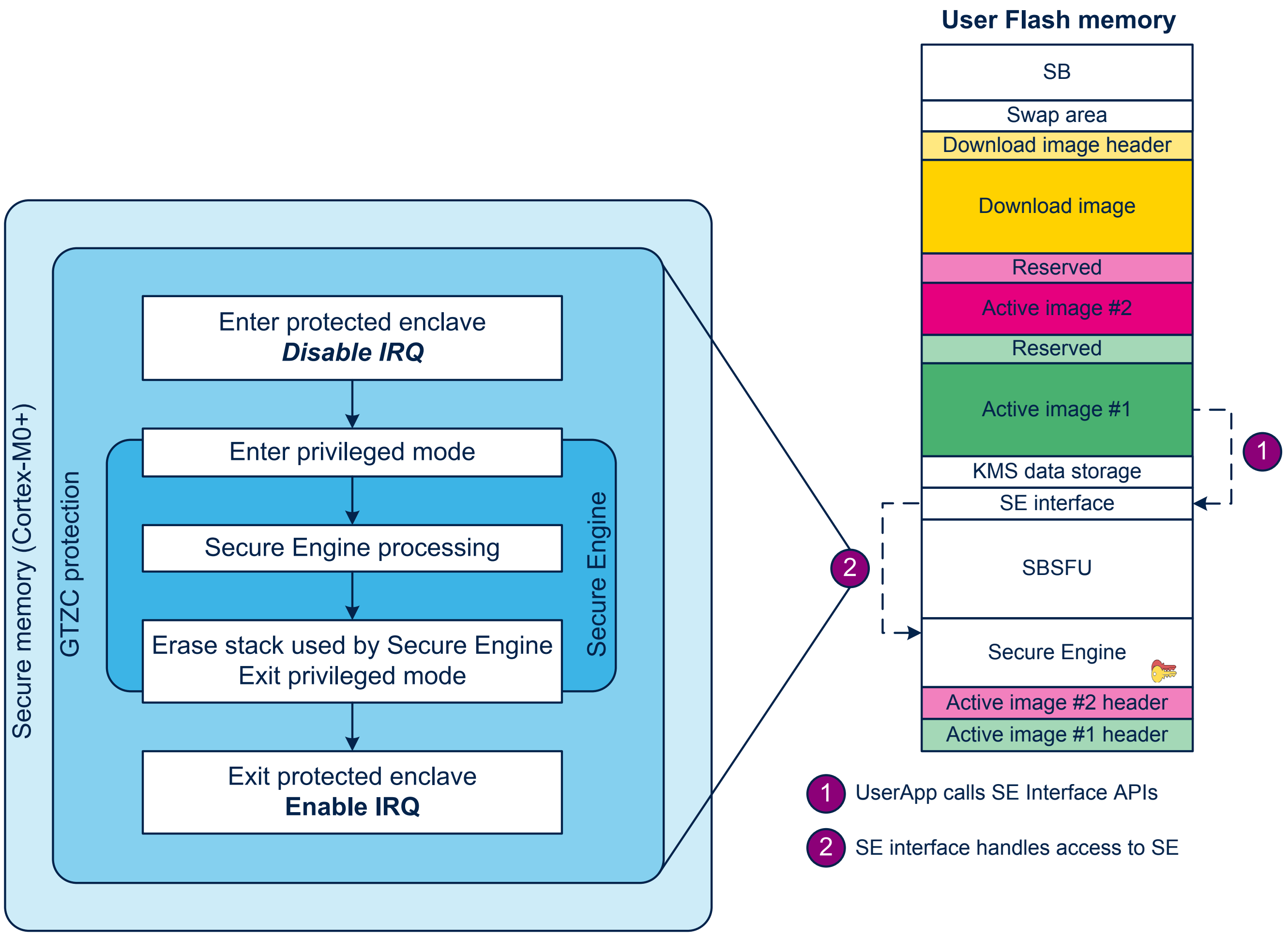


A.1.2 SE interface

Protected code must be non-interruptible and interrupts must be disabled before calling the call-gate function.

SE interface provides a user-friendly wrapper handling the entrance and exit to a protected enclave where the actual SE call-gate function is executed as illustrated in the following figure:

Figure 36. **Secure Engine interface**



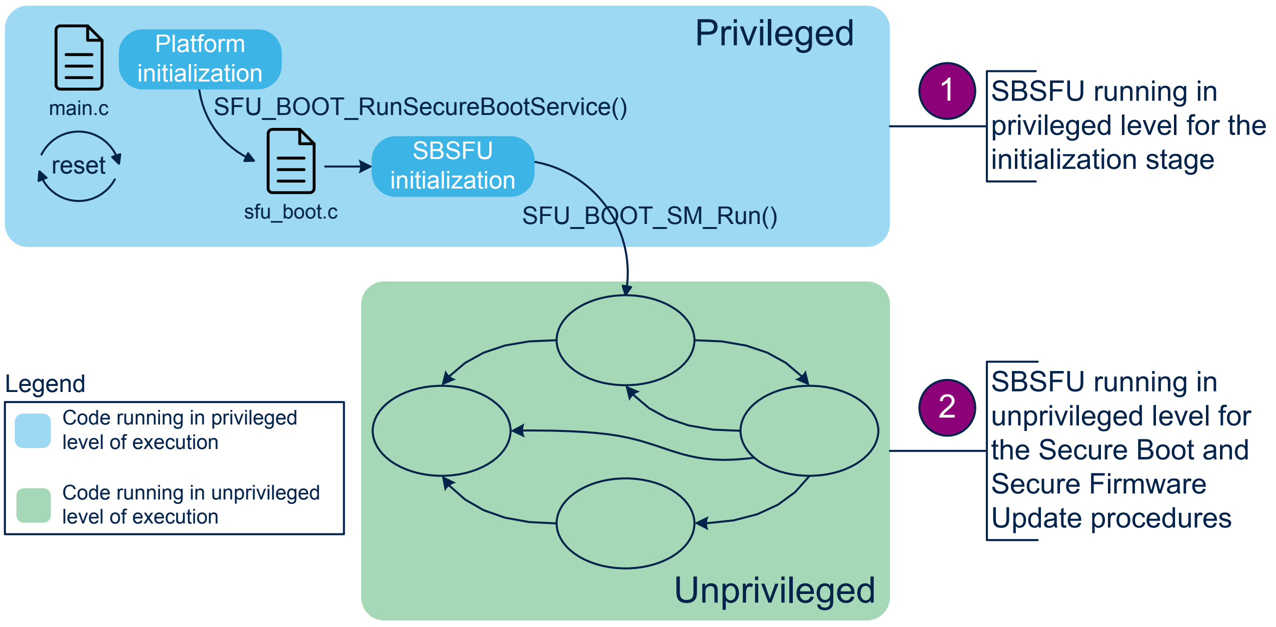
SE interface mechanism simplifies the control access to the call-gate independent from user implementation. SE interface APIs are shared with the user application, which therefore executes sensitive operations (if not locked via the Secure Engine lock service) in a secure way using the services provided by SE.

A.2 GTZC / MPU-based Secure Engine isolation

A.2.1 Principle

The GTZC / MPU-based Secure Engine isolation relies on the concept of privileged and unprivileged levels of software execution. The software must run in an unprivileged level of execution by default (when SBSFU or the User application is running), except for very specific actions like platform initialization or interrupt handling. This is described in the following figure.

Figure 37. SBSFU running in the unprivileged level of software execution for standard operations


Privileged
main.c
Platform initialization
reset
SFU_BOOT_RunSecureBootService()
sfu_boot.c
SBSFU initialization
SFU_BOOT_SM_Run()
1
SBSFU running in privileged level for the initialization stage
Legend
Code running in privileged level of execution
Code running in unprivileged level of execution
2
SBSFU running in unprivileged level for the Secure Boot and Secure Firmware Update procedures
Unprivileged

When the software runs in an unprivileged mode, any attempt to access the Secure Engine code or data results in an GTZC fault (or MPU fault for KMS data storage): this ensures the isolation of the critical assets.

This isolation of the Secure Engine is implemented thanks to specific GTZC and MPU regions as shown in the next table.

Table 6. GTZC regions for Secure Engine isolation

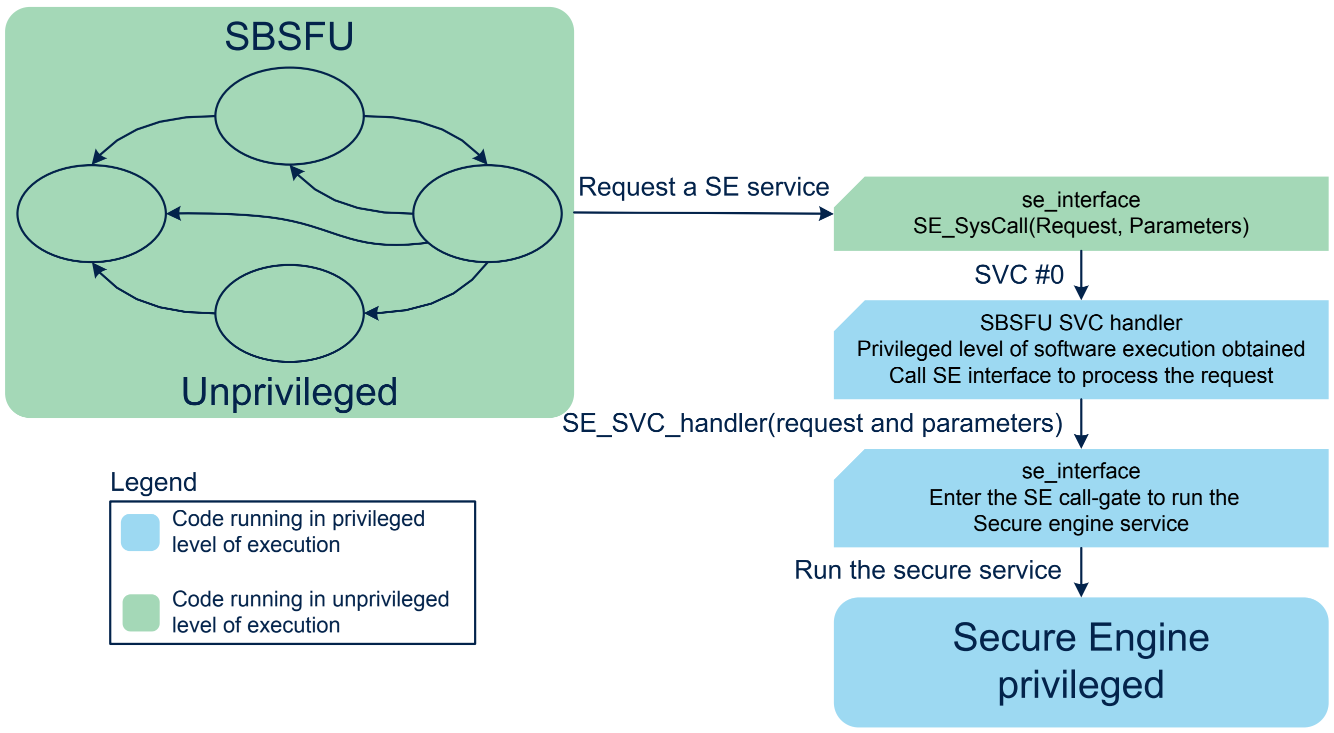
Region content	Secure privileged permission	Non-secure and/or unprivileged permission
Secure Engine code and constants	Read only (execution allowed)	No access
Secure Engine stack and data	Read write (not executable)	No access

To run a Secure Engine service, the caller must first enter the privileged level of software execution through a controlled access point. This is done using the concept of SE interface (refer to Section A.1.2 SE interface).
It abstracts the request to get the privileged level of software execution: this request consists of triggering a supervisor (SVC) call.

In the SBSFU examples delivered in the STM32CubeWL SBSFU, the SBSFU application implements an SVC handler to catch this SVC call and process it with another SE interface service to enter the Secure Engine via its call-gate as shown in the next figure.

Note: The SVC handler must be trusted because it is a key element of the Secure Engine access control.

Figure 38. **SBSFU requesting a Secure Engine service**


SBSFU
Unprivileged
Request a SE service
se_interface
SE_SysCall(Request, Parameters)
SVC #0
SBSFU SVC handler
Privileged level of software execution obtained
Call SE interface to process the request
SE_SVC_handler(request and parameters)
se_interface
Enter the SE call-gate to run the
Secure engine service
Run the secure service
Secure Engine
privileged
Legend
Code running in privileged level of execution
Code running in unprivileged level of execution

The SE call gate mechanism uses the concepts described in Section A.1.1 Secure Engine Core call-gate mechanism to provide a unique entry point to the Secure Engine services. The constraints for the placement of the call-gate code are only the GTZC and MPU regions constraints (the call-gate must be located in the privileged code region).

When the Secure Engine processing ends, the call-gate concept provides a single exit point and the SVC call return sequence applies. This return sequence brings the software back to the unprivileged level of execution. Then, any further direct access to the Secure Engine code and data generates an GTZC fault.

The Secure Engine service exit is described in the figure presented below.

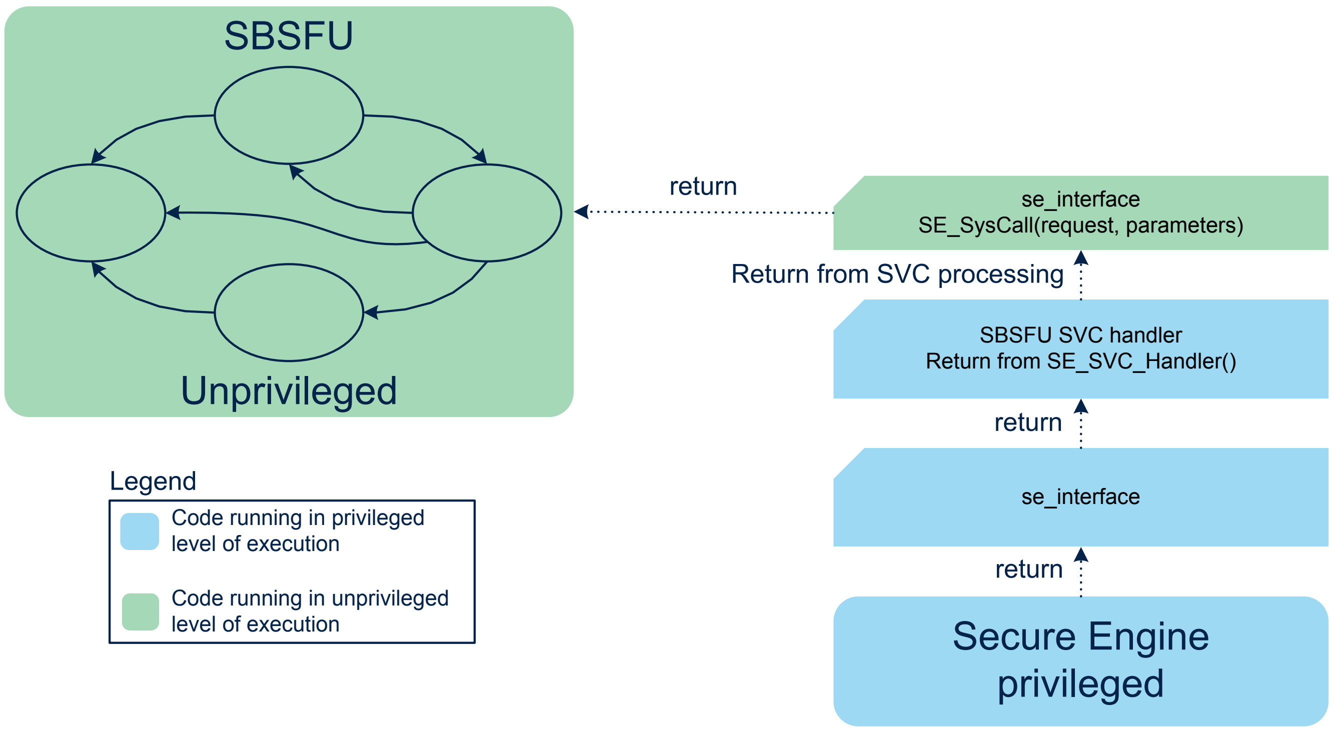

SBSFU
Unprivileged
return
se_interface
SE_SysCall(request, parameters)
Return from SVC processing
SBSFU SVC handler
Return from SE_SVC_Handler()
return
se_interface
return
Secure Engine privileged
Legend
Code running in privileged level of execution
Code running in unprivileged level of execution

Figure 39. **Exiting a Secure Engine service**

A.2.2 Additional protections

Thanks to the secure mode:

- Only the privileged code can configure the DMAs to access privileged code or data
- Only the secure code can configure the DMAs to access secure code or data
- The SE keys and the active image headers are located in the HDP area: once the user applications are running, the SE keys and the active image headers are no more accessible.
- Once SBSFU has configured TZSC, its registers are locked and it is no longer possible to update the configuration until next reset

A.2.3 Constraints

The GTZC-based Secure Engine isolation relies fully on the fact that a privileged level of software execution is required to access the Secure Engine services. The SVC handler is the controlled access point to a privileged level of execution (this must be trusted code). Additionally, any piece of code running in privileged mode must be trusted also (interrupt routines, initialization code, and others) so that the controlled access point is not bypassed. It is key to partition the software very carefully and avoid granting a privileged level of execution when not required (the software must run in an unprivileged mode as much as possible).

Appendix B Dual-slot configuration

SBSFU_2_Slots_DualCore example handles three slots located in internal Flash memory.

BFU_2_Slots example handles two slots located in internal Flash memory.

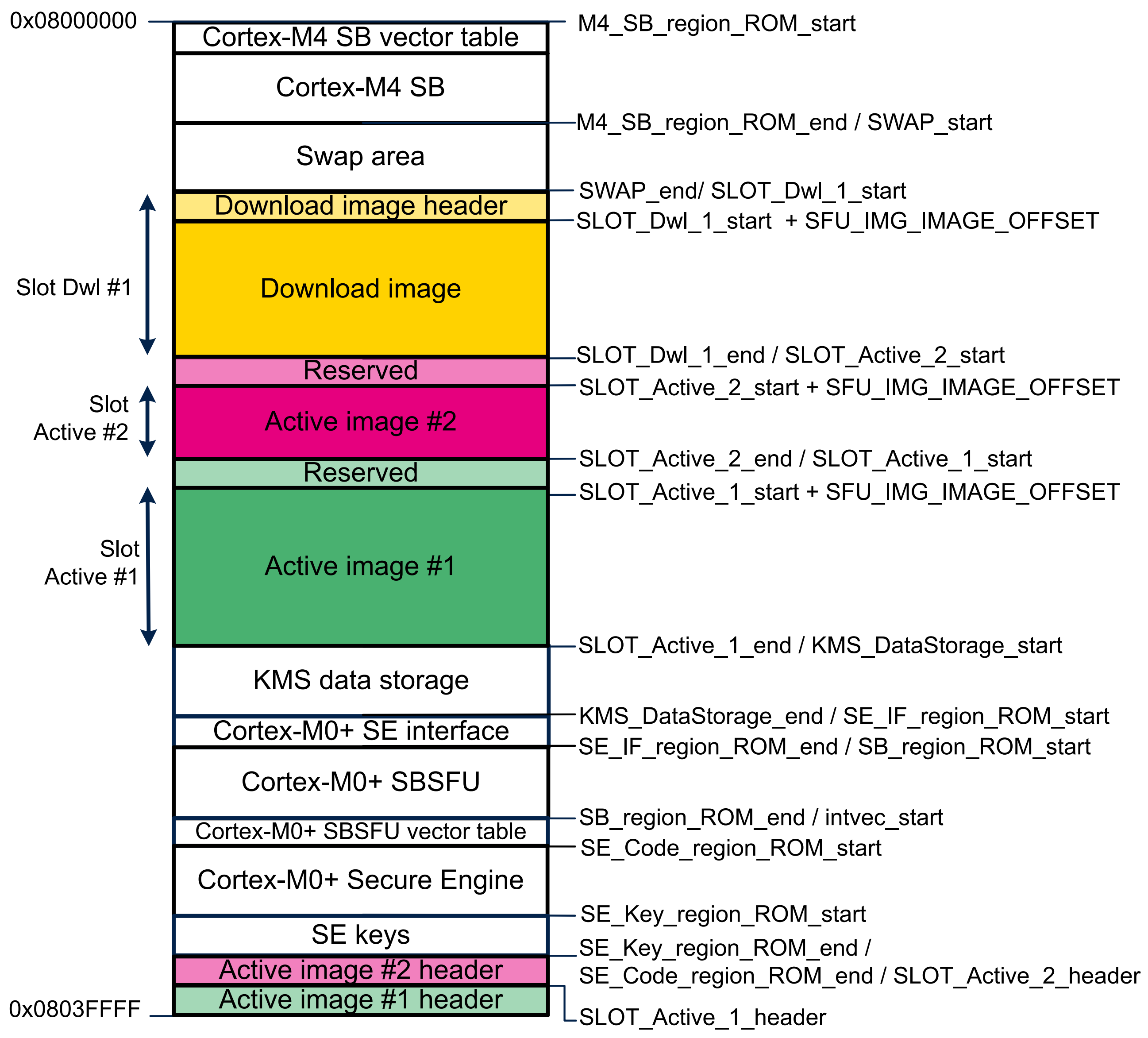
B.1 Elements and roles on dual-core dual-slot configuration

- Cortex®-M0+ active slot:
  - This slot contains the Cortex®-M0+ active firmware (firmware associated with a non-contiguous firmware header). This is the Cortex®-M0+ user application that is launched at boot time by SBSFU (after verifying its validity)
- Cortex®-M4 active slot:
  - This slot contains the Cortex®-M4 active firmware (firmware associated with a non-contiguous firmware header). This is the Cortex®-M4 user application that is launched at boot time by Cortex®-M4 SB (after SBSFU verified its validity)
- Download slot:
  - This slot is used to store the downloaded firmware (firmware header and encrypted firmware) to be installed at the next reboot.
  - In the case of a partial image, the size of this slot can be lower than the size of the biggest active slot (which contains the full image). The download slot can be sized according to the maximum possible partial image
- Swap region:
  - This is a Flash memory area used to swap the content of active and download slots.
  - Nevertheless, this area is not a buffer used for every swap of the Flash memory sector. It is used to move the first sector, hence creating a shift in Flash memory allowing swapping the two slots sector by sector.

The next figure represents the mapping on NUCLEO-WL55JC. The mapping order for slots and swap elements depends on the STM32 Series:

- The SE keys and the active slot headers must be mapped at the end of the Flash memory to be protected by the HDP area
- The Secure Engine must be mapped just before the SE keys and the active slot headers to be protected by the secure privileged memory
- The SBSFU and the Cortex®-M0+ active slot must be mapped just before the Secure Engine to be mapped in the secure unprivileged memory
- The SE keys and the Secure Engine must be adjacent as they are both part of SE_Core.bin
- The Secure Engine and SBSFU must be adjacent to be protected by the same WRP area
- The download slot and swap must be mapped in the non-secure unprivileged memory to be accessible by the Cortex®-M4 loader

Figure 40. Internal user Flash memory mapping with 512-byte headers (dual-core configuration)

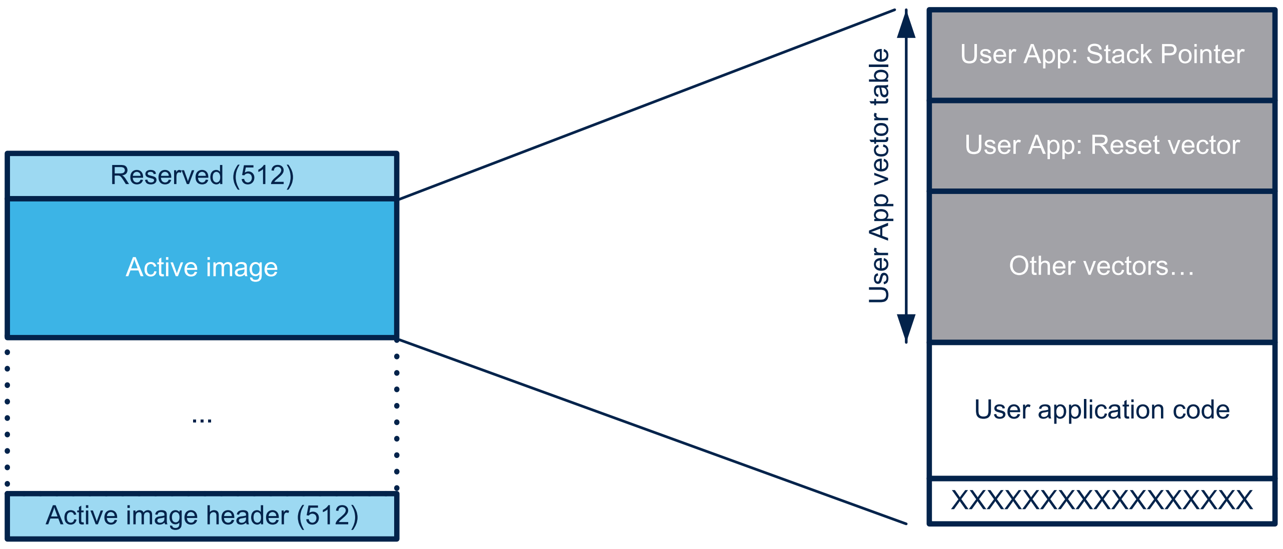

0x08000000
Cortex-M4 SB vector table
Cortex-M4 SB
Swap area
Download image header
Download image
Reserved
Active image #2
Reserved
Active image #1
KMS data storage
Cortex-M0+ SE interface
Cortex-M0+ SBSFU
Cortex-M0+ SBSFU vector table
Cortex-M0+ Secure Engine
SE keys
Active image #2 header
Active image #1 header
0x0803FFFF
Slot Dwl #1
Slot Active #2
Slot Active #1
M4_SB_region_ROM_start
M4_SB_region_ROM_end / SWAP_start
SWAP_end/ SLOT_Dwl_1_start
SLOT_Dwl_1_start + SFU_IMG_IMAGE_OFFSET
SLOT_Dwl_1_end / SLOT_Active_2_start
SLOT_Active_2_start + SFU_IMG_IMAGE_OFFSET
SLOT_Active_2_end / SLOT_Active_1_start
SLOT_Active_1_start + SFU_IMG_IMAGE_OFFSET
SLOT_Active_1_end / KMS_DataStorage_start
KMS_DataStorage_end / SE_IF_region_ROM_start
SE_IF_region_ROM_end / SB_region_ROM_start
SB_region_ROM_end / intvec_start
SE_Code_region_ROM_start
SE_Key_region_ROM_start
SE_Key_region_ROM_end /
SE_Code_region_ROM_end / SLOT_Active_2_header
SLOT_Active_1_header

B.2 Mapping definition on dual-core dual-slot configuration

The mapping definition is located in the *Linker_Common* folder for each example. The following figure shows how to find information such as slot size and SBSFU code size in the NUCLEO-WL55JC example.

To start the application, SBSFU initializes the SP register with the user application stack pointer value, then jumps to the user application reset vector as shown on the next figure.

Figure 41. **User application vector table (dual-core configuration)**



Cortex-M0+ source code: SBSFU sfu_fwimg_common.c, sfu_mpu_isolation.c, sfu_se_mpu.s and Cortex-M0+ UserApp system_stm32wlxx.c

1. End of Cortex-M0+ SBSFU execution : SFU_IMG_LaunchActiveImg(uint32_t SlotNumber)
```
/* this function does not return */
/* first image identified as master image */
SFU_MPU_SysCall((uint32_t)SB_SYSCALL_LAUNCH_APP, SlotStartAdd[SlotNumber] +
SFU_IMG_IMAGE_OFFSET);
```
2. Enter the privileged mode to be able to perform the jump: SFU_MPU_SVC_Handler(uint32_t *args)
```
launch_application(args[1], (uint32_t)jump_to_function);
```
3. Jump to user application: jump_to_function
```
LDR R1, [R0]
LDR R2, [R0,#4]
MOV SP, R1
BX  R2
```
4. Start of Cortex-M0+ USER application: SystemInit(void)
```
SCB->VTOR = INTVECT_START;
```

***Note:** The configuration of privileged/ unprivileged mode can be done either by MPU or GTZC. In the delivered example, this handling is done by GTZC however, the MPU naming was kept as it can be changed back and forth.*

Cortex-M4 source code: SB main.c and Cortex-M4 UserApp system_stm32wlxx.c

1. End of Cortex-M4 SB execution : SFU_IMG_LaunchActiveImg(uint32_t SlotNumber)
```
jump_address = *(__IO uint32_t *)((SLOT_ACTIVE_2_START + SFU_IMG_IMAGE_OFFSET
+ 4));
/* Jump to user application */
p_jump_to_function = (Function_Pointer) jump_address;
/* Initialize user application's Stack Pointer */
__set_MSP(*(__IO uint32_t *)(SLOT_ACTIVE_2_START + SFU_IMG_IMAGE_OFFSET));
```
2. Start of Cortex-M4 USER application: SystemInit(void)
```
SCB->VTOR = INTVECT_START;
```

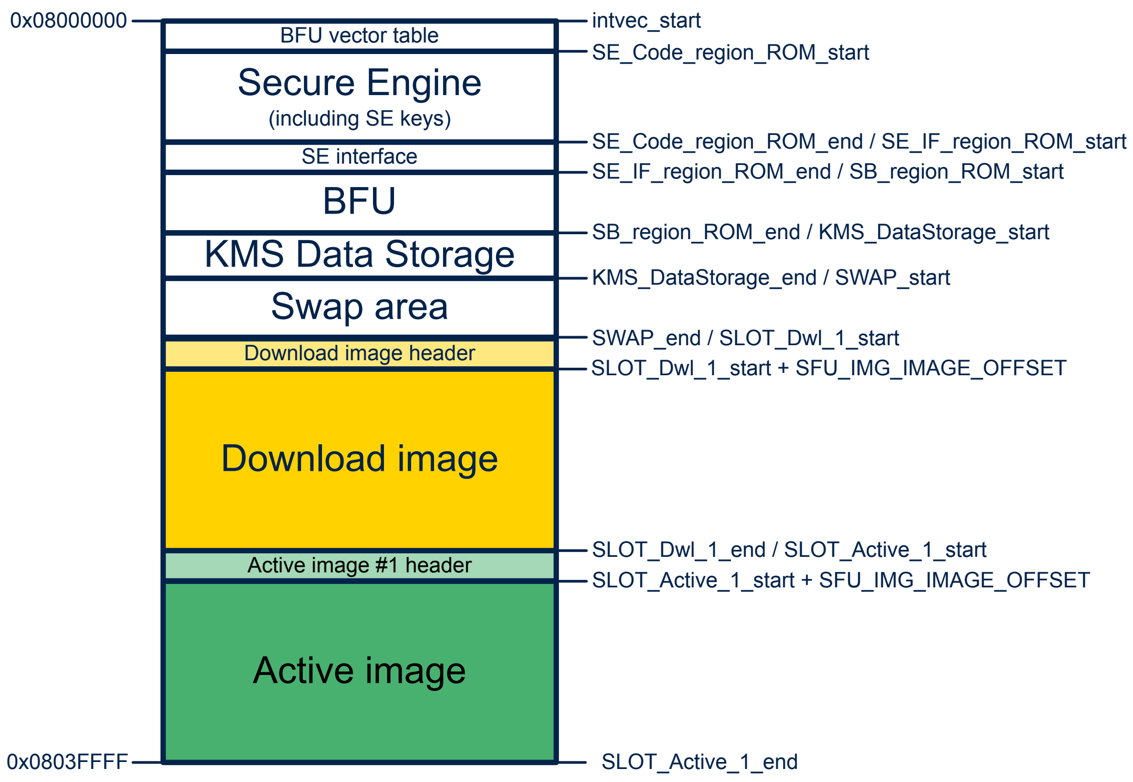
B.3 Elements and roles on single-core dual-slot configuration

- Cortex®-M4 active slot:
  - This slot contains the active firmware (firmware header and firmware). This is the user application that is launched at boot time by BFU (after verifying its validity).
- Download slot:
  - This slot is used to store the downloaded firmware (firmware header and encrypted firmware) to be installed at the next reboot.
  - In the case of a partial image, the size of this slot can be lower than the size of the active slot (which contains the full image). The download slot can be sized according to the maximum possible partial image
- Swap region:
  - This is a Flash memory area used to swap the content of active and download slots.
  - Nevertheless, this area is not a buffer used for every swap of the Flash memory sector. It is used to move the first sector, hence creating a shift in Flash memory allowing swapping the two slots sector by sector.

The next figure represents the mapping on NUCLEO-WL55JC. The mapping order for slots and swap elements depends on the STM32 Series device:

- The SE keys and the Secure Engine must be adjacent as they are both part of SE_Core.bin
- The Secure Engine and SBSFU must be adjacent to be protected by the same WRP area.

Figure 42. Internal user Flash memory mapping with 512-byte headers (single-core configuration)

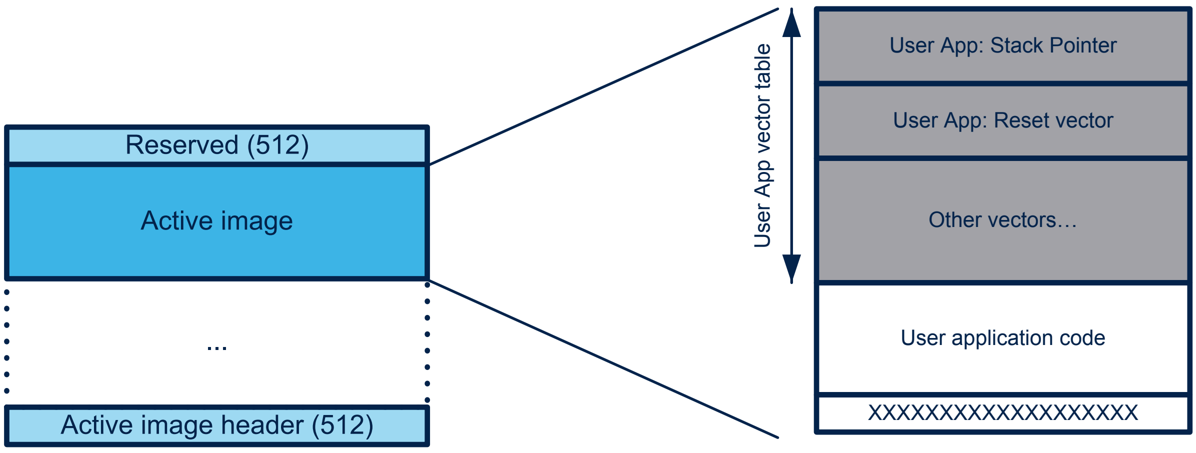

0x08000000
BFU vector table
Secure Engine
(including SE keys)
SE interface
BFU
KMS Data Storage
Swap area
Download image header
Download image
Active image #1 header
Active image
0x0803FFFF
intvec_start
SE_Code_region_ROM_start
SE_Code_region_ROM_end / SE_IF_region_ROM_start
SE_IF_region_ROM_end / SB_region_ROM_start
SB_region_ROM_end / KMS_DataStorage_start
KMS_DataStorage_end / SWAP_start
SWAP_end / SLOT_Dwl_1_start
SLOT_Dwl_1_start + SFU_IMG_IMAGE_OFFSET
SLOT_Dwl_1_end / SLOT_Active_1_start
SLOT_Active_1_start + SFU_IMG_IMAGE_OFFSET
SLOT_Active_1_end

B.4 Mapping definition on single-core dual-slot configuration

The mapping definition is located in the *Linker_Common* folder for each example. The following figure shows how to find information such as slot size and BFU code size in the NUCLEO-WL55JC example.

To start the application, BFU initializes the SP register with the user application stack pointer value, then jumps to the user application reset vector as shown on the next figure.

Figure 43. **User application vector table (single-core configuration)**



Source code: SBSFU sfu_fwimg_common.c & UserApp system_stm32wlxx.c

1. End of BFU execution : SFU_IMG_LaunchActiveImg(uint32_t SlotNumber)
```
/* Initialize address of user application to jump into */
jump_address = *(__IO uint32_t *)((SlotStartAdd[SlotNumber] + SFU_IMG_IMAGE_OFFSET + 4));
p_jump_to_function = (Function_Pointer) jump_address;

/* Initialize user application's Stack Pointer */
__set_MSP(*(__IO uint32_t *)(SlotStartAdd[SlotNumber] + SFU_IMG_IMAGE_OFFSET));

if (SFU_SUCCESS == e_ret_status)
{
  /* JUMP into User App */
  p_jump_to_function();
}
```
2. Start of USER application: SystemInit(void)
```
SCB->VTOR = INTVECT_START;
```

Appendix C Single-slot configuration

BFU_1_Slot example handles one slot located in internal Flash memory (Cortex®-M4 active firmware).

SBSFU_1_Slot_DualCore example handles two slots located in internal Flash memory (Cortex®-M0+ active firmware and Cortex®-M4 active firmware).

This mode of operation permits the maximization of the user firmware size by:

- Reducing the BFU/SBSFU footprint in Flash memory
- Allocating more Flash memory space for the user application

These benefits come at the cost of some features:

- Safe firmware image programming cannot be ensured: as soon as the installation is interrupted (power off), the firmware update process must be restarted from the beginning (including the download phase)
- The user application cannot download a new firmware image: the local download procedure is the only way to update the active user code

C.1 Elements and roles on dual-core single-slot configuration

Cortex®-M0+ active slot:

- This slot contains the Cortex®-M0+ active firmware (firmware associated with a non-contiguous firmware header). This is the Cortex®-M0+ user application that is launched at boot time by SBSFU (after verifying its validity)
- This slot is directly updated when a new firmware image is downloaded and installed (after firmware header verification)

Cortex®-M4 active slot:

- This slot contains the Cortex®-M4 active firmware (firmware associated with a non-contiguous firmware header). This is the Cortex®-M4 user application that is launched at boot time by Cortex®-M4 SB (after SBSFU verified its validity)
- This slot is directly updated when a new firmware image is downloaded and installed (after firmware header verification).

C.2 Mapping definition on dual-core single-slot configuration

The mapping definition is located in the *Linker_Common* folder for each example. Figure 41. User application vector table (dual-core configuration) shows how to find information such as slot size and SBSFU code size in the NUCLEO-WL55JC example.

To start the application, SBSFU initializes the SP register with the user application stack pointer value, then jumps to the user application reset vector (refer to Figure 41).

C.3 Elements and roles on single-core single-slot configuration

Active slot (Cortex®-M4):

- This slot contains the active firmware (firmware header and firmware). This is the user application that is launched at boot time by BFU (after verifying its validity)
- This slot is directly updated when a new firmware image is downloaded and installed (after firmware header verification)

C.4 Mapping definition on single-core single-slot configuration

The mapping definition is located in the *Linker_Common* folder for each example. Figure 43. User application vector table (single-core configuration) shows how to find information such as slot size and BFU code size in the NUCLEO-WL55JC example.

To start the application, BFU initializes the SP register with the user application stack pointer value, then jumps to the user application reset vector (refer to Figure 43).

Appendix D Cryptographic schemes handling

Three cryptographic schemes are provided as examples to illustrate the cryptographic operations. The default cryptographic scheme uses both symmetric (AES-CBC) and asymmetric (ECDSA) cryptography. So, it handles a private key (AES128 private key) as well as a public key (ECC key).

Two alternate schemes are provided. They are selected using a SECoreBin compiler switch (named ‘SECBOOT_CRYPTO_SCHEME’).

D.1 Cryptographic schemes contained in this package

The table below shows the cryptographic scheme selected with the `SECBOOT_CRYPTO_SCHEME` compiler switch.

Table 7. **Cryptographic scheme list**

SECBOOT_CRYPTO_SCHEME value	Authentication	Confidentiality	Integrity
SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256 (default)	ECDSA	AES128-CBC	SHA256
SECBOOT_ECCDSA_WITHOUT_ENCRYPT_SHA256	ECDSA	None(1)	SHA256
SECBOOT_AES128_GCM_AES128_GCM_AES128_GCM(2)	AES-GCM		

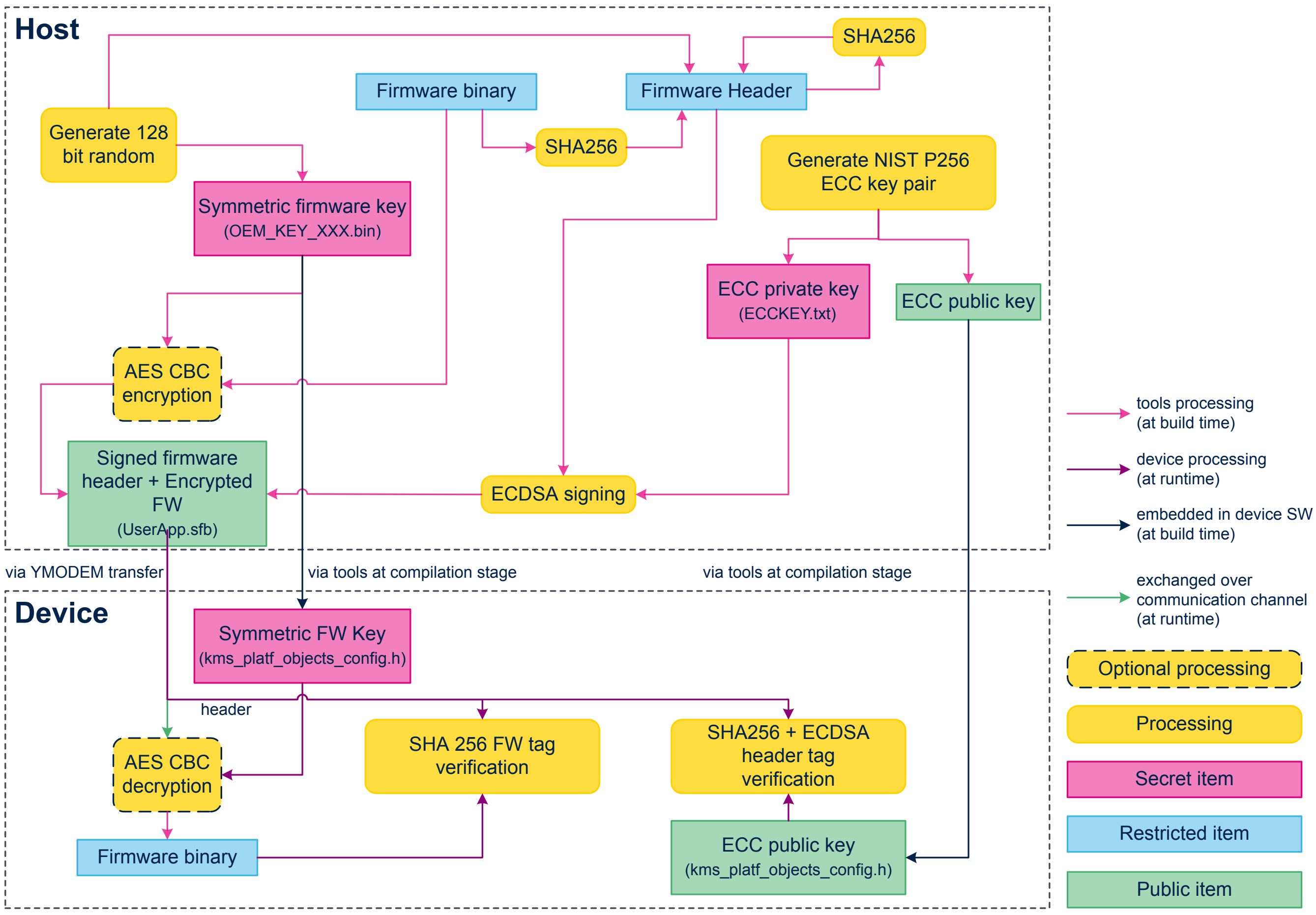
1. *The SBSFU project must also be configured to deal with a clear firmware image by setting the compiler switch SFU_IMAGE_PROGRAMMING_TYPE to the value SFU_CLEAR_IMAGE.*
2. *For the symmetric cryptographic scheme, it is highly recommended to configure a unique symmetric key for each product.*

D.2 Asymmetric verification and symmetric-encryption schemes

D.2.1 Cryptographic schemes with full software-implementation

These schemes (SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256 SECBOOT_ECCDSA_WITHOUT_ENCRYPT_SHA256) are implemented for firmware decryption and verification as illustrated below.

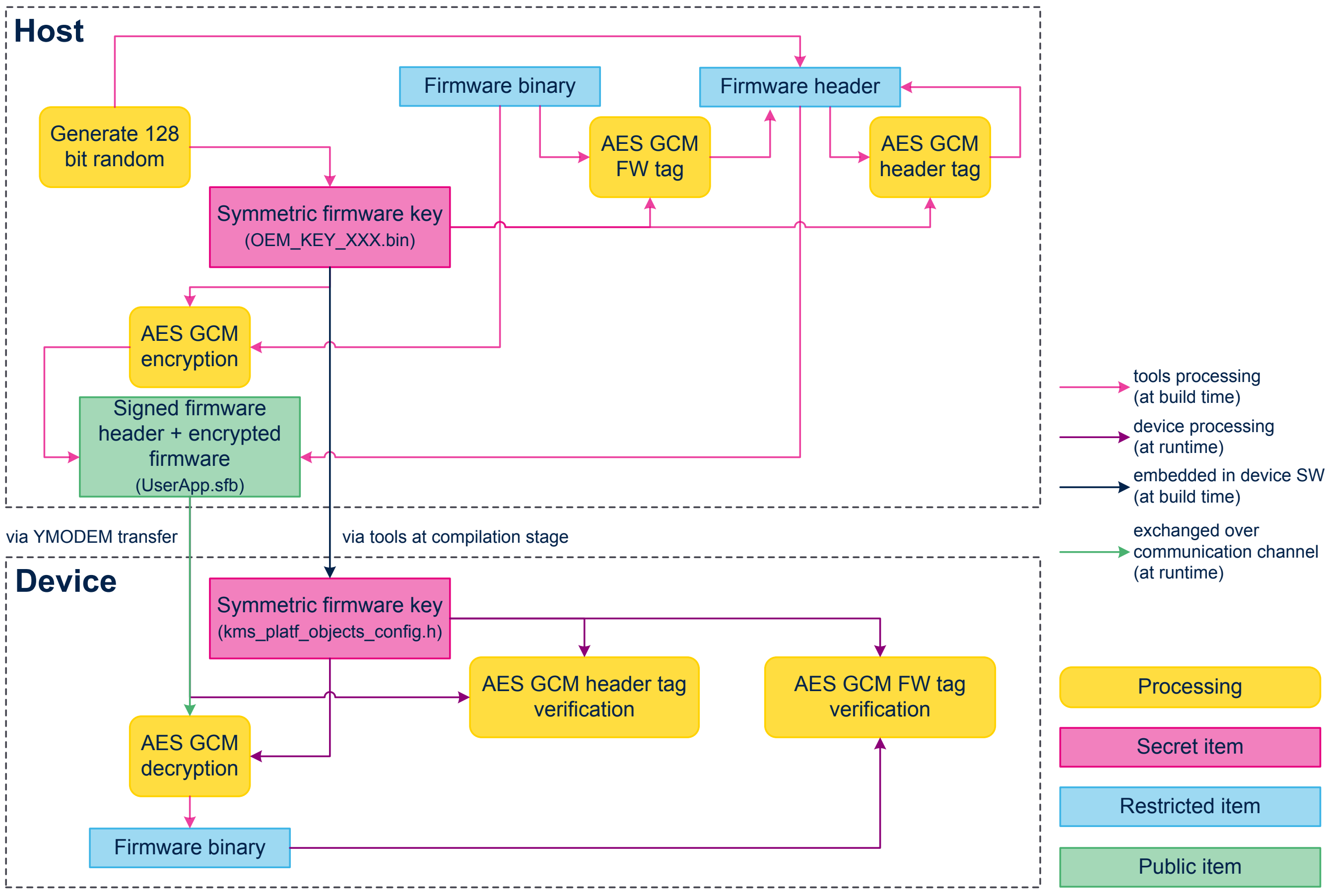
Figure 44. Asymmetric verification and symmetric encryption


Host
SHA256
Firmware binary
Firmware Header
Generate 128 bit random
SHA256
Generate NIST P256 ECC key pair
Symmetric firmware key (OEM_KEY_XXX.bin)
ECC private key (ECCKEY.txt)
ECC public key
AES CBC encryption
Signed firmware header + Encrypted FW (UserApp.sfb)
ECDSA signing
tools processing (at build time)
device processing (at runtime)
embedded in device SW (at build time)
exchanged over communication channel (at runtime)
via YMODEM transfer
via tools at compilation stage
via tools at compilation stage
Device
Symmetric FW Key (kms_platf_objects_config.h)
header
AES CBC decryption
SHA 256 FW tag verification
SHA256 + ECDSA header tag verification
Firmware binary
ECC public key (kms_platf_objects_config.h)
Optional processing
Processing
Secret item
Restricted item
Public item

D.3 Symmetric verification and encryption scheme

This scheme (SECBOOT_AES128_GCM_AES128_GCM_AES128_GCM) is implemented for firmware decryption and verification as illustrated below.

Figure 45. Symmetric verification and encryption


Host
Firmware binary
Firmware header
Generate 128 bit random
AES GCM FW tag
AES GCM header tag
Symmetric firmware key (OEM_KEY_XXX.bin)
AES GCM encryption
Signed firmware header + encrypted firmware (UserApp.sfb)
via YMODEM transfer
via tools at compilation stage
Device
Symmetric firmware key (kms_platf_objects_config.h)
AES GCM header tag verification
AES GCM FW tag verification
AES GCM decryption
Firmware binary
tools processing (at build time)
device processing (at runtime)
embedded in device SW (at build time)
exchanged over communication channel (at runtime)
Processing
Secret item
Restricted item
Public item

D.4 Secure Boot and Secure Firmware Update flow

The two following figures indicate how the cryptographic operations (asymmetric cryptographic scheme with firmware encryption) are integrated into the SBSFU execution boot flows.

Figure 46. SBSFU dual-slot boot flows

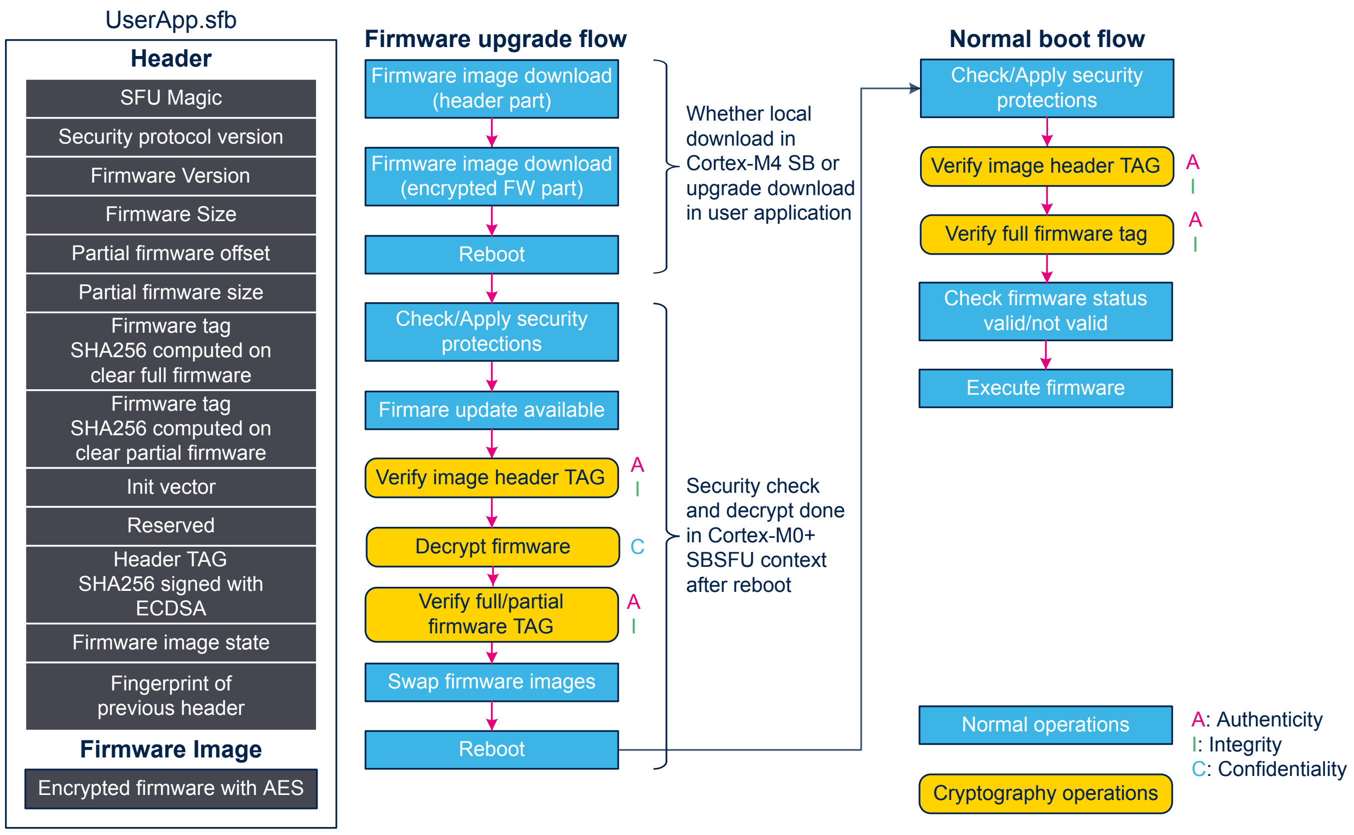
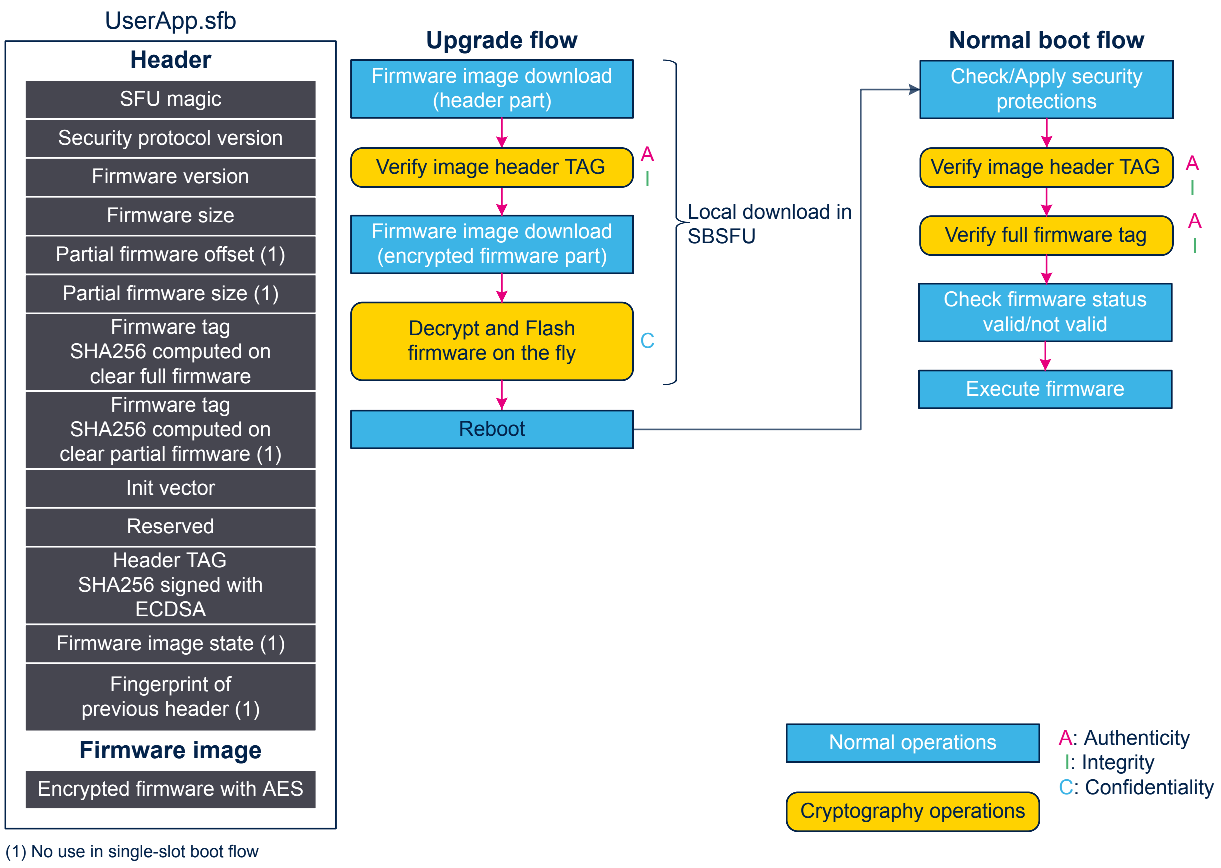

UserApp.sfb
Header
SFU Magic
Security protocol version
Firmware Version
Firmware Size
Partial firmware offset
Partial firmware size
Firmware tag SHA256 computed on clear full firmware
Firmware tag SHA256 computed on clear partial firmware
Init vector
Reserved
Header TAG SHA256 signed with ECDSA
Firmware image state
Fingerprint of previous header
Firmware Image
Encrypted firmware with AES
Firmware upgrade flow
Firmware image download (header part)
Firmware image download (encrypted FW part)
Reboot
Whether local download in Cortex-M4 SB or upgrade download in user application
Check/Apply security protections
Firmare update available
Verify image header TAG
A
I
Decrypt firmware
C
Verify full/partial firmware TAG
A
I
Swap firmware images
Reboot
Security check and decrypt done in Cortex-M0+ SBSFU context after reboot
Normal boot flow
Check/Apply security protections
Verify image header TAG
A
I
Verify full firmware tag
A
I
Check firmware status valid/not valid
Execute firmware
Normal operations
Cryptography operations
A: Authenticity
I: Integrity
C: Confidentiality

Figure 47. **SBSFU single-slot boot flows**


UserApp.sfb
Header
SFU magic
Security protocol version
Firmware version
Firmware size
Partial firmware offset (1)
Partial firmware size (1)
Firmware tag SHA256 computed on clear full firmware
Firmware tag SHA256 computed on clear partial firmware (1)
Init vector
Reserved
Header TAG SHA256 signed with ECDSA
Firmware image state (1)
Fingerprint of previous header (1)
Firmware image
Encrypted firmware with AES
(1) No use in single-slot boot flow
Upgrade flow
Firmware image download (header part)
Verify image header TAG
A
I
Firmware image download (encrypted firmware part)
Decrypt and Flash firmware on the fly
C
Reboot
Local download in SBSFU
Normal boot flow
Check/Apply security protections
Verify image header TAG
A
I
Verify full firmware tag
A
I
Check firmware status valid/not valid
Execute firmware
Normal operations
Cryptography operations
A: Authenticity
I: Integrity
C: Confidentiality

Appendix E Firmware image preparation tool

The STM32CubeWL SBSFU is delivered with the *prepareimage* firmware image preparation tool which allows to:

- Take into account the selected cryptographic scheme and keys
- Encrypt the firmware image when required
- Generate partial firmware image, by extracting binary differences between two full images
- Generate the firmware header with all the data required for the authentication and integrity checks

The *prepareimage* tool is delivered in two formats:

- Windows® executable: the standard Windows® command interpreter is required
- Python™ scripts: a Python™ interpreter as well as the elements listed in *Middlewares\ST\STM32_Secure_Engine\Utilities\KeysAndImages\readme.txt* are required

The Windows® executable enables quick and easy use of the package with all three predefined cryptographic schemes. The Python™ scripts, delivered as source code, offer the possibility to define additional cryptographic schemes flexibly.

Note: Refer to Section Appendix F for KMS specificities.

E.1 Tool location

The Python™ scripts as well as the Windows® executable are located in the Secure Engine component, in the following folder: *Middlewares\ST\STM32_Secure_Engine\Utilities\KeysAndImages*.

E.2 Inputs

The package is delivered with some default keys and cryptography settings in folder *Applications\SBSFU_2_Slots_DualCore\2_Images_SECoreBin\Binary*.

Each of the following files can be used as such, or modified to take the user settings into account:

- *ECCKEY1.txt*: private ECC key in PEM format. It is used to sign the firmware header. This key is **not** embedded in the *SECoreBin*, only the corresponding public key is generated by the tools in the *kms_platf_objects_config.h* file (see Section Appendix F for more details)
- *nonce.bin*: this is either a nonce (when AES-GCM is used) or an IV (when AES-CBC is used). This value is added automatically by the tools to the firmware header
- *OEM_KEY_COMPANY1_key_AES_CBC.bin*: symmetric AES-CBC key. This key is used for the AES-CBC encryption and decryption operations and is embedded in the *kms_platf_objects_config.h* file. This file is exclusive with *OEM_KEY_COMPANY1_key_AES_GCM.bin*
- *OEM_KEY_COMPANY1_key_AES_GCM.bin*: symmetric AES-GCM key. This key is used for all AES-GCM operations and is embedded in the *kms_platf_objects_config.h* file. This file is exclusive with *OEM_KEY_COMPANY1_key_AES_CBC.bin*

The tool uses the appropriate set of files based on the cryptographic scheme selected using `SECBOOT_CRYPTO_SCHEME` in *Applications\SBSFU_2_Slots_DualCore\2_Images_SECoreBin\Inc\se_crypto_config.h*

E.3 Outputs

The tool generates:

- The *kms_platf_objects_config.h* file compiled in the *SECoreBin* project: this file contains the keys (private symmetric key and public ECC key when applicable) embedded in the device and the code to access them. When running the tool from the IDE, this file is located in *Applications\SBSFU_2_Slots_DualCore\2_Images_SECoreBin\EWARM\*. In the case of multiple images configuration, one set of keys is generated per image
- A *sfb* file packing the Cortex®-M0+ user firmware header and the encrypted user firmware image (when the selected cryptographic scheme enables user firmware encryption). When running the tool from the IDE, this file is generated in *Applications\SBSFU_2_Slots_DualCore\2_Images_UserApp_M0Plus\Binary\*.
- A *sfb* file packing the Cortex®-M4 user firmware header and the encrypted user firmware image (when the selected cryptographic scheme enables user firmware encryption). When running the tool from the IDE, this file is generated in *Applications\SBSFU_2_Slots_DualCore\2_Images_UserApp_M4\Binary\*.
- A *bin* file concatenating the SBSFU binary, both UserApp binaries, and active firmware image headers. Flashing this file into the device with a flasher tool makes the two UserApp installation process simple since the firmware headers and firmware images are already correctly installed. It is not needed to use the SBSFU application for installing the two UserApp.

Note: Before programming the bin file into the device, a mass deletion must be performed. To detect any malicious software, SBSFU verifies at startup that there is no additional code after UserApp in the active slots.

- Three log files, *output.txt*, located in

Applications\SBSFU_2_Slots_DualCore\2_Images_SECoreBin\EWARM\, in *Applications\SBSFU_2_Slots_DualCore\2_Images_UserApp_M0Plus\EWARM* and in *Applications\SBSFU_2_Slots_DualCore\2_Images_UserApp_M4\EWARM* to trace the executions of *prebuild.bat* and *postbuild.bat*.

E.4 IDE integration

The *prepareimage* tool is integrated with the IDEs as Windows® batch files for:

- Pre-build actions for the *SECoreBin* application: at this stage, the cryptographic keys are managed
- Post-build actions for the *UserApp* applications: at this stage, the matching firmware image (Cortex®-M4 or Cortex®-M0+) is built

When compiling with the IDE, the keys and the firmware image are handled. No extra action is required from the user. At the end of the compilation steps:

- The required keys are embedded in the *SECoreBin* binary
- The firmware images to be installed are generated in the proper format, with the appropriate firmware header, as *sfb* files. These *sfb* files can be transferred over the Ymodem protocol for installation by SBSFU
- The *bin* file that can be flashed for the test of UserApp (*SBFU_UserApp.bin*).

The batch files integrating the tool in the IDE are located in the folder *Applications\SBSFU_2_Slots_DualCore\2_Images_SECoreBin\EWARM*:

- *prebuild.bat*: invoking the tool to perform the pre-build actions when compiling the *SECoreBin* project
- *postbuild.bat*: invoking the tool to perform the post-build actions when compiling the *UserApp* projects

These batch files allow seamless switching from the Windows® executable variant to the Python™ script variant of the *prepareimage* tool. The procedure is described in the files themselves.

E.5 Partial image

A partial image contains only the binary portion of the new firmware image to install, versus the active firmware image.

Partial-image usage presents various benefits:

- Smaller firmware image to download (reduced download duration)
- Faster firmware image installation
- Memory mapping optimization, with possibly smaller download slot (maximum size of partial image) and bigger active slots (maximum size of the full image)

This feature is available on the dual-slot variant only (not available on the single-slot variant).

For the partial-image feature, the header structure includes two instances of the firmware tag:

- Partial firmware tag: tag of the partial image only. This tag is checked by SBSFU during the image installation phase
- Firmware tag: tag of the full image (after the installation of the partial image). This tag is checked by SBSFU during the image boot phase

The *prepareimage* tool can be used to generate partial firmware image, starting from the active image and the new firmware image to install:

1. Perform first a binary comparison between the two full images in clear
2. Then apply the usual image preparation procedure

Refer to the detailed procedure *Example for partial update* described in the *readme.txt file* of the *prepareimage* tool (*Middlewares\ST\STM32_Secure_Engine\Utilities\KeysAndImages\readme.txt*).

Appendix F Key management services (KMS)

F.1 Key update process description

PKCS#11 APIs manage keys through objects containing a different type of information:

- Object header: giving information about the object itself, such as attribute size, number of attributes and object ID
- Object attributes: such as type, size, and value

Static embedded keys are embedded in the code and cannot be modified. On the contrary, updatable keys can be modified in an NVM storage located inside the protected/isolated environment:

- An updatable key with dynamic ID can be created via a secure object creation procedure running inside the protected/isolated environment ensuring that the key remains inside the protected/isolated environment (key value is stored in the NVM storage and only the object ID is returned to the application)
- An updatable key with static ID can be updated in the NVM storage via a secure update procedure using static embedded root keys (authenticity check, data integrity check and data decryption). It means that the key must be provided to KMS in a specific format to ensure key authenticity, integrity, and confidentiality. KMS example is provided with a tool allowing to automatically generate the encrypted object based on ECDSA asymmetric cryptography for data authenticity/integrity verification and based on AES-CBC symmetric cryptography for data confidentiality. Once an encrypted object is downloaded into the device, the SBSFU application detects it at the next system reset and SBSFU application processes it via the KMS secure update procedure (`C_STM_ImportBlob()` function).

The next two figures illustrate the key creation and update procedure.

Figure 48. Encrypted object creation

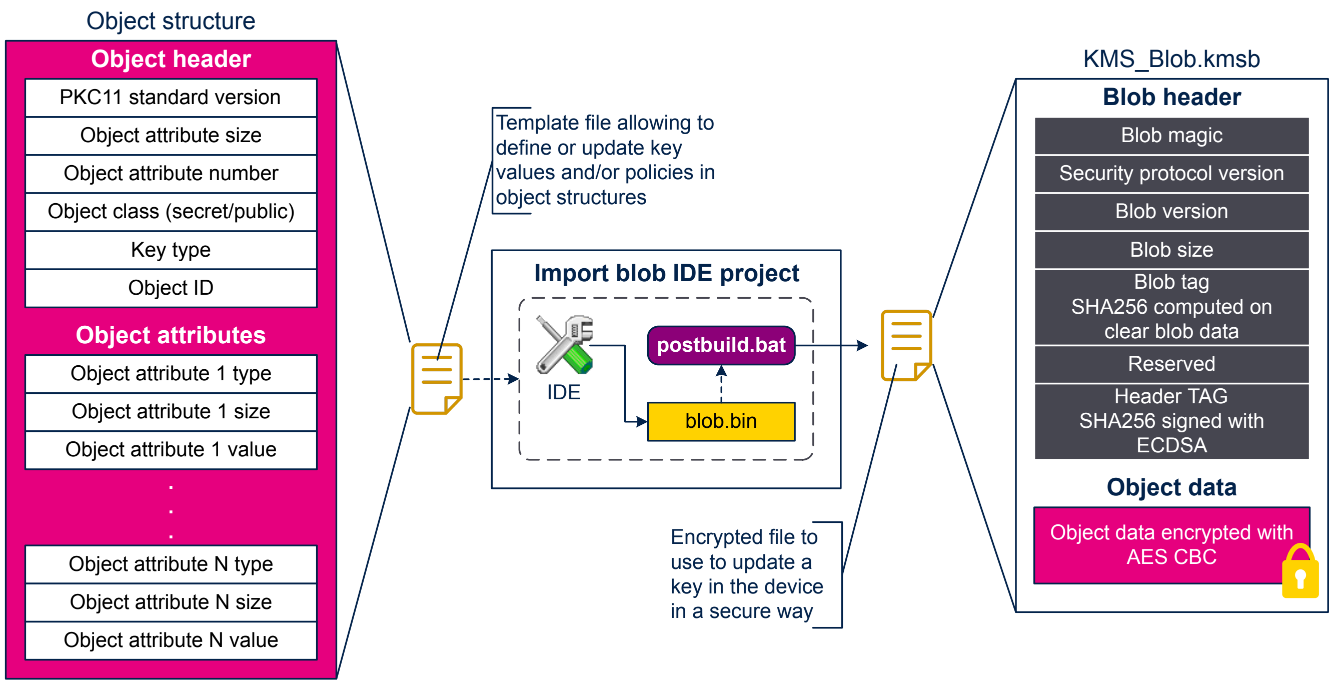
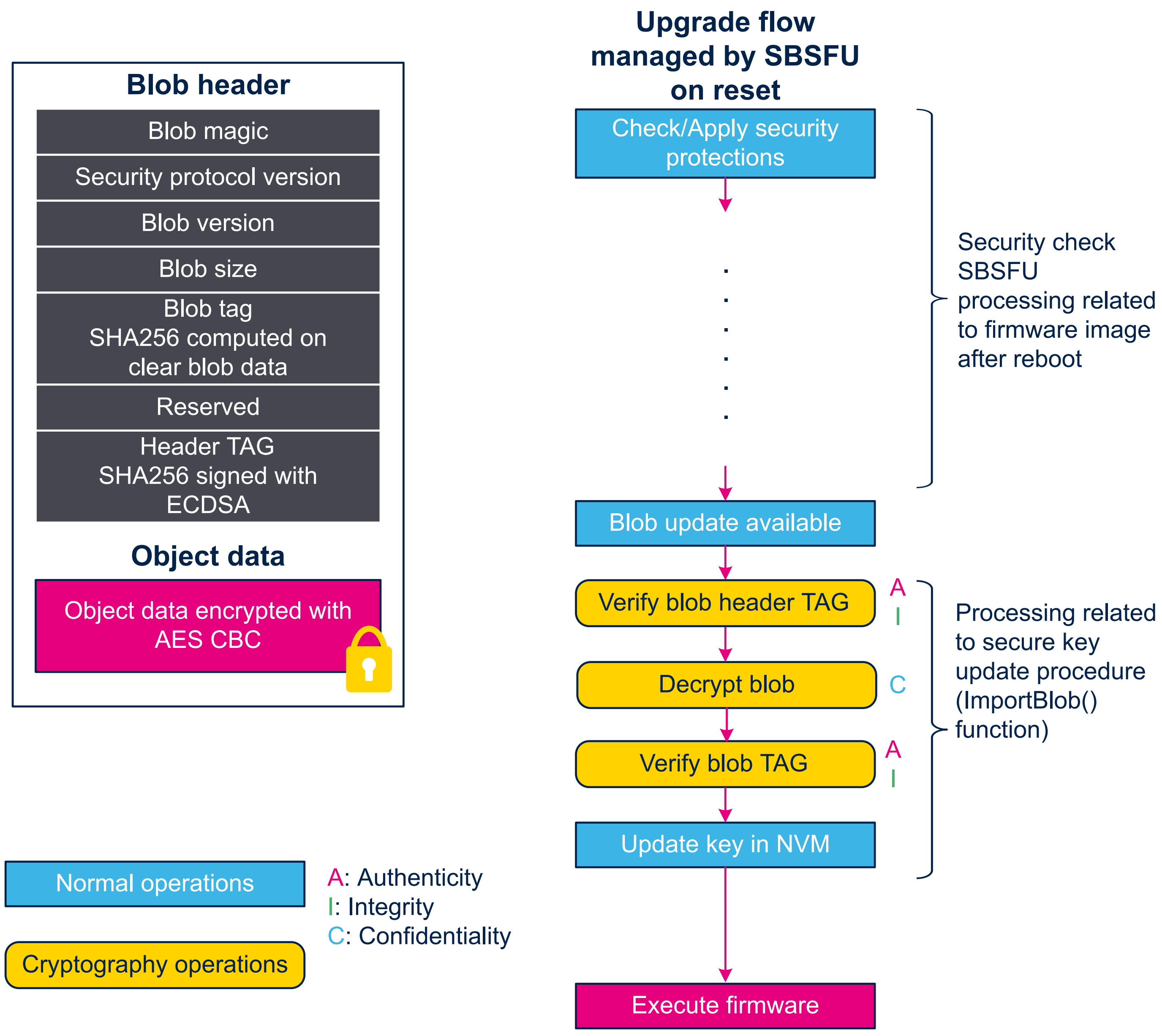


Figure 49. Secure update procedure

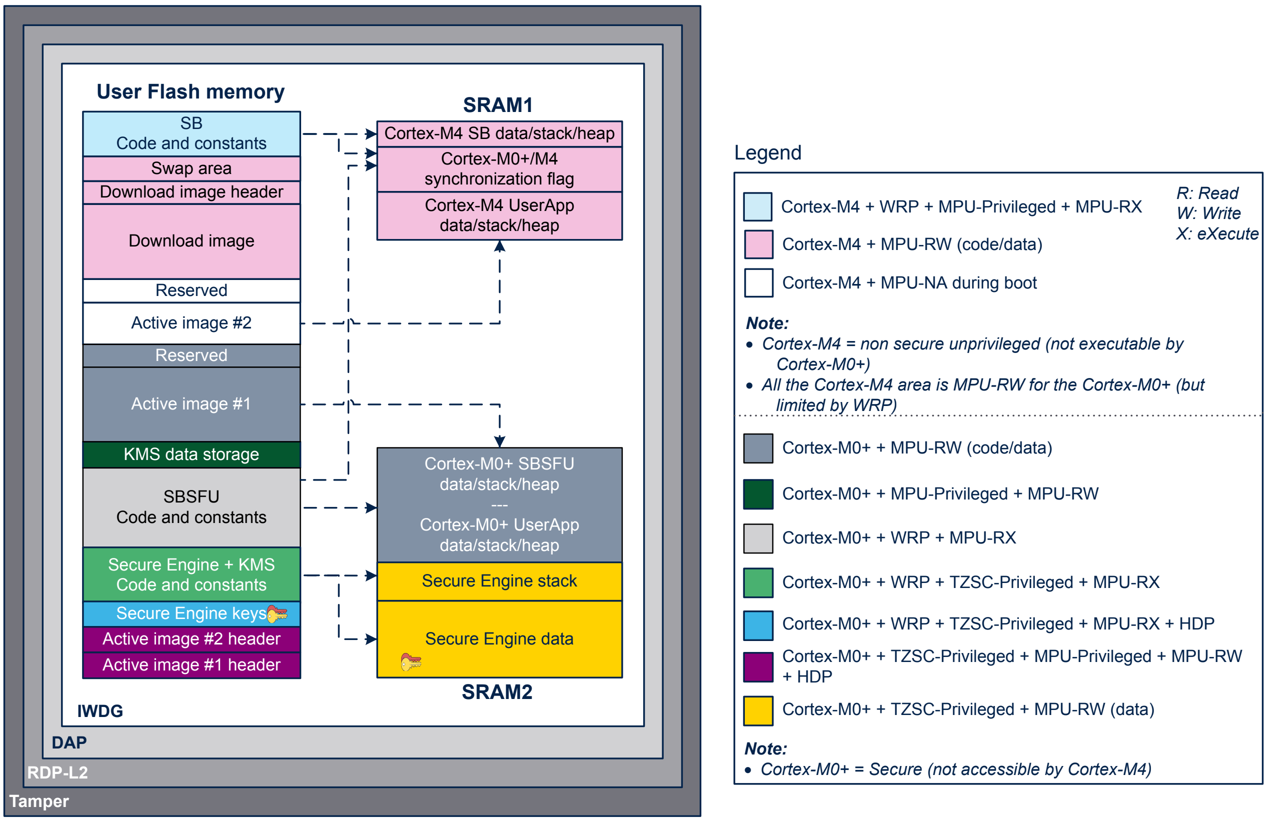

Blob header
Blob magic
Security protocol version
Blob version
Blob size
Blob tag
SHA256 computed on clear blob data
Reserved
Header TAG
SHA256 signed with ECDSA
Object data
Object data encrypted with AES CBC
Upgrade flow managed by SBSFU on reset
Check/Apply security protections
Security check SBSFU processing related to firmware image after reboot
Blob update available
Verify blob header TAG
A
I
Decrypt blob
C
Verify blob TAG
A
I
Update key in NVM
Processing related to secure key update procedure (ImportBlob() function)
Execute firmware
Normal operations
Cryptography operations
A: Authenticity
I: Integrity
C: Confidentiality

F.2 SBSFU static keys generation

With KMS middleware integration, SBSFU keys are stored inside the KMS code running in the secure enclave as shown in the figure below. During the SECoreBin compilation stage, *prebuild.bat* updates SBSFU static embedded keys inside *kms_platf_objects_config.h* located in *Applications\SBSFU_2_Slots_DualCore\2_Images_SECoreBin\EWARM\*.

Note: As shown in the figure below, the Secure Engine keys are located on their own sector to be protected by HDP. However, they are still part of the Secure Engine binary (Secure Engine + KMS).

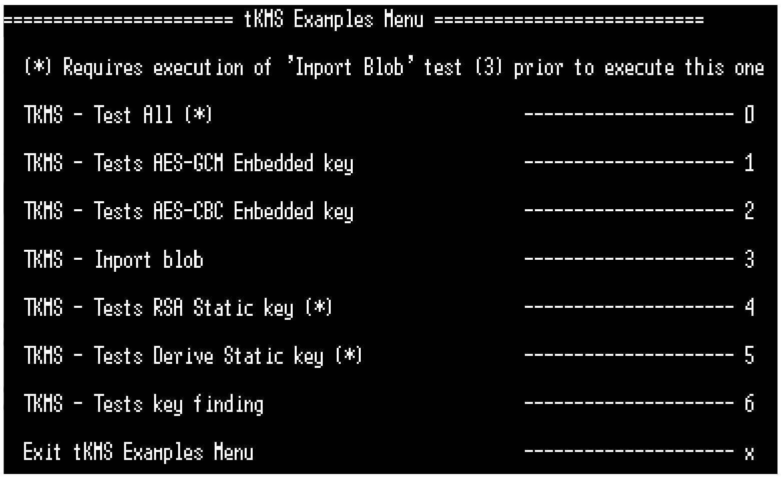
Figure 50. KMS key storage


User Flash memory
SB Code and constants
Swap area
Download image header
Download image
Reserved
Active image #2
Reserved
Active image #1
KMS data storage
SBSFU Code and constants
Secure Engine + KMS Code and constants
Secure Engine keys
Active image #2 header
Active image #1 header
SRAM1
Cortex-M4 SB data/stack/heap
Cortex-M0+/M4 synchronization flag
Cortex-M4 UserApp data/stack/heap
Cortex-M0+ SBSFU data/stack/heap

Cortex-M0+ UserApp data/stack/heap
Secure Engine stack
Secure Engine data
SRAM2
IWDG
DAP
RDP-L2
Tamper
Legend
Cortex-M4 + WRP + MPU-Privileged + MPU-RX
Cortex-M4 + MPU-RW (code/data)
Cortex-M4 + MPU-NA during boot
R: Read
W: Write
X: eXecute
Note:
Cortex-M4 = non secure unprivileged (not executable by Cortex-M0+)
All the Cortex-M4 area is MPU-RW for the Cortex-M0+ (but limited by WRP)
Cortex-M0+ + MPU-RW (code/data)
Cortex-M0+ + MPU-Privileged + MPU-RW
Cortex-M0+ + WRP + MPU-RX
Cortex-M0+ + WRP + TZSC-Privileged + MPU-RX
Cortex-M0+ + WRP + TZSC-Privileged + MPU-RX + HDP
Cortex-M0+ + TZSC-Privileged + MPU-Privileged + MPU-RW + HDP
Cortex-M0+ + TZSC-Privileged + MPU-RW (data)
Note:
Cortex-M0+ = Secure (not accessible by Cortex-M4)

F.3 UserApp menu

A specific menu is added providing examples using KMS services exported services through a standard PKCS#11 interface: AES-GCM/CBC encryption/decryption, RSA signature/verification, key provisioning, and AES ECB key derivation.



```
====================== tKMS Examples Menu ==========================

 (*) Requires execution of 'Import Blob' test (3) prior to execute this one

 TKMS - Test All (*)                                --------------------- 0

 TKMS - Tests AES-GCM Embedded key                  --------------------- 1

 TKMS - Tests AES-CBC Embedded key                  --------------------- 2

 TKMS - Import blob                                 --------------------- 3

 TKMS - Tests RSA Static key (*)                    --------------------- 4

 TKMS - Tests Derive Static key (*)                 --------------------- 5

 TKMS - Tests key finding                           --------------------- 6

 Exit tKMS Examples Menu                            --------------------- x
```

Figure 51. **KMS menu**

Revision history

Table 8. **Document revision history**

Date	Version	Changes
27-Nov-2020	1	Initial release.
2	08-Jul-2021	Updated: • How does this software integrate in STM32Cube? • Section 5 Protection measures and security strategy • Section 6.1 General description • Section 6.2 Architecture • Section 6.3 Folder structure including Figure 8 and Figure 9 • Section 6.5 Application compilation process with IAR Embedded Workbench toolchain • Section 7.2.1 Development toolchains and compilers • Section 7.2.4 STM32CubeWL SBSFU firmware image preparation tool • Section 8.2 Application compilation • Section 8.4.3 First file transfer completion • Section 8.4.6 Second file transfer completion • Title of Section 8.5.1 Download a new firmware image (dual-slot configuration only) • Section 8.6 Programming a new software when the securities are activated • Section Appendix B Dual-slot configuration • Title of Section B.1 Elements and roles on dual-core dual-slot configuration and Figure 40. Internal user Flash memory mapping with 512-byte headers (dual-core configuration) • Title of Section B.2 Mapping definition on dual-core dual-slot configuration and Figure 41. User application vector table (dual-core configuration) • Section Appendix C Single-slot configuration • Title of Section C.3 Elements and roles on single-core single-slot configuration • Section C.4 Mapping definition on single-core single-slot configuration • Updated *"SBSFU_2_Images_DualCore"* references to *"SBSFU_2_Slots_DualCore"* on Section E.2 Inputs, Section E.3 Outputs, Section E.4 IDE integration and Section F.2 SBSFU static keys generation • Figure 51. KMS menu Added: • Section B.3 Elements and roles on single-core dual-slot configuration • Section B.4 Mapping definition on single-core dual-slot configuration • Section C.1 Elements and roles on dual-core single-slot configuration • Section C.2 Mapping definition on dual-core single-slot configuration

Contents

List of tables

List of figures