UM2262
User manual

Getting started with the X-CUBE-SBSFU STM32Cube Expansion Package

Introduction

This user manual describes how to get started with the X-CUBE-SBSFU STM32Cube Expansion Package.

The Secure Boot (SB) and Secure Firmware Update (SFU) 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 such as secret code and firmware encryption key.

In addition, Secure Boot (Root of Trust services) checks and activates the STM32 security mechanisms, and checks the authenticity and integrity of user application code before every execution to ensure that invalid or malicious code cannot be run.

The Secure Firmware Update application receives the encrypted firmware image, checks its authenticity, decrypts it, and then checks the integrity of the code before installing it.

The Secure Firmware Update application supports:

- Two modes of operation:
  - The dual-image mode, which enables safe image programming, with firmware image backup and rollback capabilities
  - The single-image mode, which maximizes the user application size
- Three cryptographic schemes using symmetric and asymmetric cryptographic operations

X-CUBE-SBSFU supplements the STM32Cube software technology, making portability across different STM32 microcontrollers easy. It comes with an example implementation running on the NUCLEO-L476RG platform.

X-CUBE-SBSFU is provided as reference code to demonstrate state-of-the-art usage of the STM32 security protection mechanisms. It is a starting point for OEMs to develop their own SBSFU as a function of their product security requirement levels.

X-CUBE-SBSFU is classified ECCN 5D002.

Contents

List of tables

List of figures

1 General information

Table 1 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
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
GUI	Graphical user interface
HAL	Hardware abstraction layer
IDE	Integrated development environment
IV	Initialization vector
IWDG	Independent watch dog
FW	Firmware
FWALL	Firewall
MAC	Message authentication code
MCU	Microcontroller unit
MPU	Memory protection unit
NONCE	Number used only once
PCROP	Proprietary code read out protection
PEM	Privacy enhanced mail
RDP	Read protection
SB	Secure boot
SE	Secure engine
SFU	Secure firmware update
SM	State machine
UART	Universal asynchronous receiver/transmitter
UUID	Universally unique identifier
WRP	Write protection

Table 2 presents the definition of terms that are relevant for a better understanding of this document.

Table 2. List of terms

Acronym	Description
Firmware image	A binary image (executable) run by the device as user application.
Firmware header	Bundle of meta-data describing the firmware image to be installed. It contains firmware information and cryptographic information.
sfb file	Binary file packing the firmware header and the firmware image.

The X-CUBE-SBSFU Secure Boot and Secure Firmware Update Expansion Package runs on STM32 32-bit microcontrollers based on the Arm®(a) Cortex®-M processor.

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

2 STM32Cube overview

What is STM32Cube?

STMCube™ is an STMicroelectronics original initiative to make developers' lives easier by reducing development effort, time and cost.STM32Cube is the implementation of STMCube™ that covers the whole STM32 portfolio.

STM32Cube includes:

- STM32CubeMX, a graphical software configuration tool that allows the generation of C initialization code using graphical wizards.
- A comprehensive MCU Package, delivered per STM32 microcontroller Series (such as the STM32CubeF4 for the STM32F4 Series), with:
  - The STM32CubeHAL, STM32 abstraction layer embedded software ensuring maximized portability across the STM32 portfolio.
  - Low-layer APIs (LL) offering a fast light-weight expert-oriented layer which is closer to the hardware than the HAL. LL APIs are available only for a set of peripherals.
  - A consistent set of middleware components such as RTOS, USB and Graphics.
  - All embedded software utilities, delivered with a full set of examples.

How does this software complement STM32Cube?

The proposed software is based on the STM32CubeHAL, the hardware abstraction layer for the STM32 microcontroller. The package extends STM32Cube by providing a middleware component (Secure Engine) for managing all critical data and operations (such as cryptography operations accessing firmware encryption key and others).

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

- SE_CoreBin application: provides a binary including all the "trusted" code.
- Secure Boot and Secure Firmware Upgrade (SBSFU) application:
  - Secure Boot (Root of Trust)
  - Local download via UART Virtual COM
  - FW installation management
- User application:
  - Downloads a new use firmware in dual-image mode of operation
  - Provides examples testing protection mechanisms

The sample applications are delivered in dual-image and single-image modes of operation and can be configured in any of the supported cryptographic scheme.

This user manual describes the typical use of the package:

- Based on the NUCLEO-L476RG board
- With sample applications operating in dual-image mode and configured with asymmetric authentication and symmetric FW encryption

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

3 Secure Boot and Secure Firmware Update (SBSFU)

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. In fact, 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 in a secure way.

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

Typical countermeasures are based on cryptography (with associated secret key) and on 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.

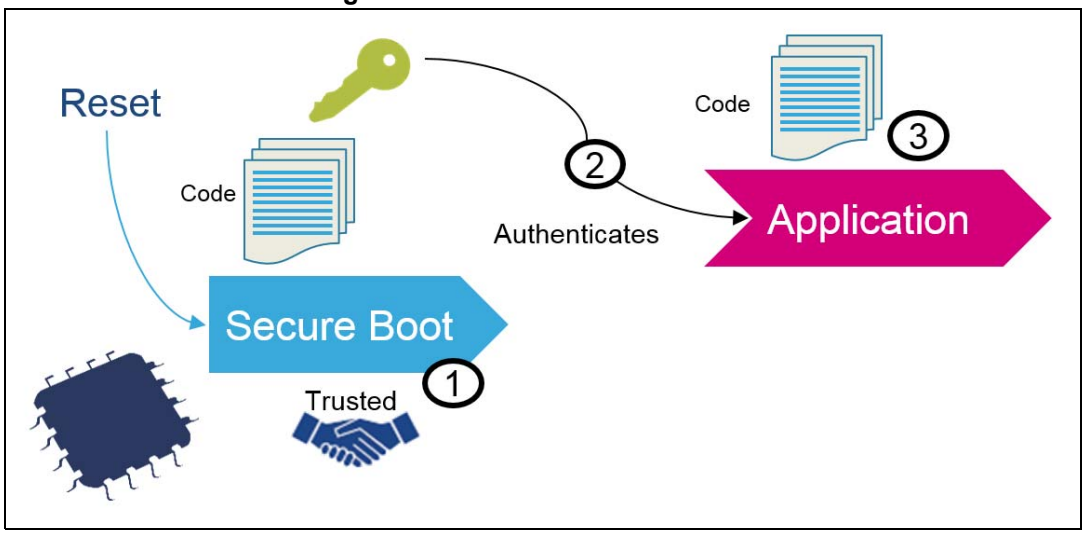
3.2 Secure Boot

Secure Boot (SB) asserts the integrity and authenticity of the user application image that is executed: cryptographic checks are used in order to prevent any unauthorized or maliciously modified software from running. The Secure Boot process implements a Root of Trust (refer to *Figure 1*): starting from this trusted component (1), every other component is authenticated (2) before its execution (3).

Integrity is verified so as to be sure that the image that is going to be executed has not been corrupted or maliciously modified.

Authenticity check aims to verify that the firmware image is coming from a trusted and known source in order to prevent unauthorized entities to install and execute code.

Figure 1. Secure Boot Root of Trust



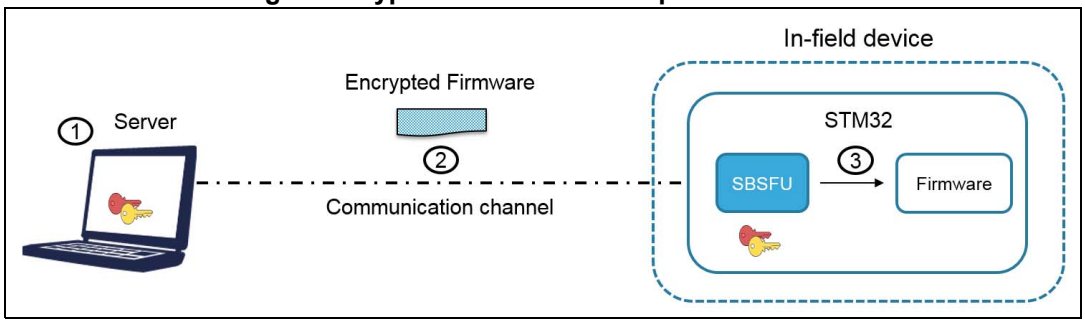
3.3 Secure Firmware Update

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 *Figure 2*, 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 image and 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.

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

Confidentiality is implemented so as 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.

Integrity is verified so as to be sure that the received image is not corrupted.

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

3.4 Cryptography operations

The X-CUBE-SBSFU STM32Cube Expansion Package 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 X-CUBE-SBSFU Expansion Package are:

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

Table 3 presents the various security features associated with each of the cryptographic schemes.

Table 3. Cryptographic scheme comparison

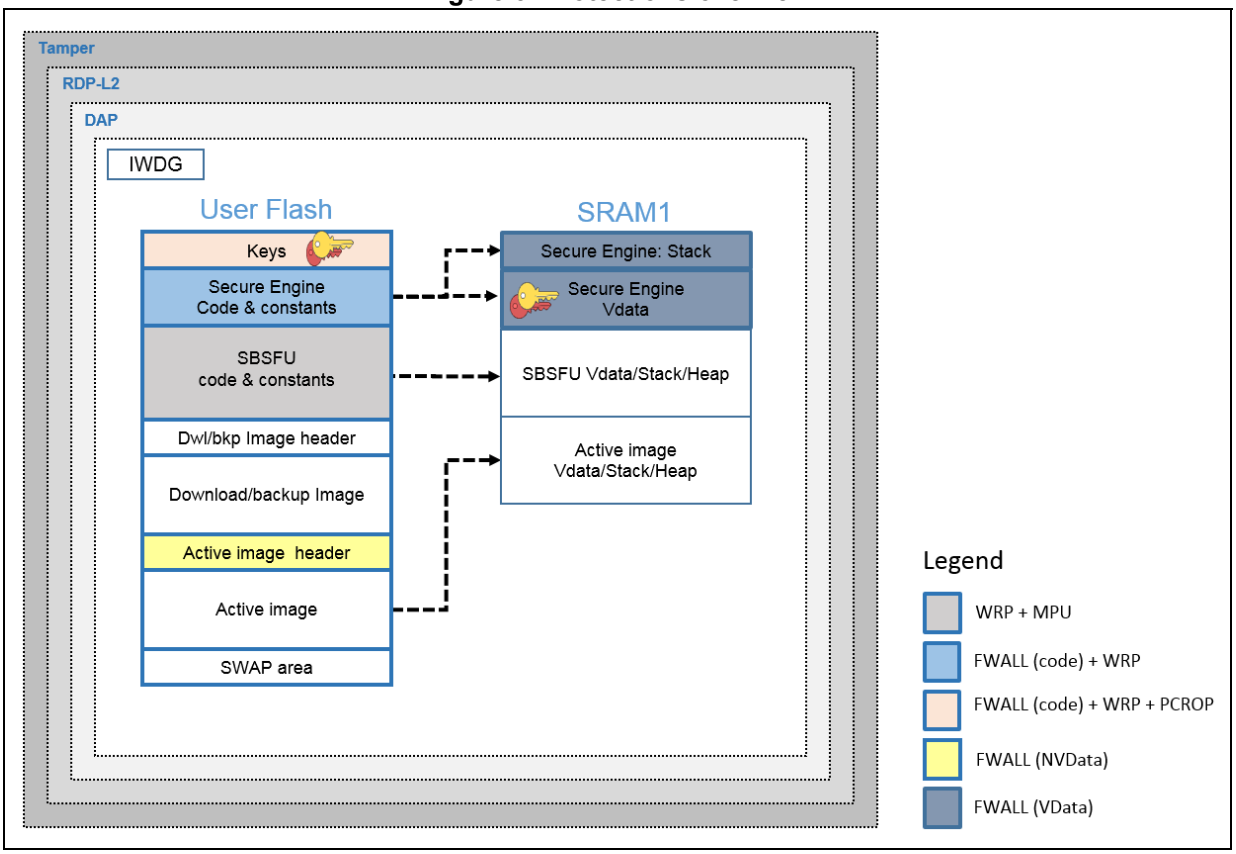
Features	Asymmetric with AES encryption	Asymmetric without encryption	Symmetric (AES GCM)
Confidentiality	AES CBC encryption (FW binary)	None: the user FW is in clear format.	AES GCM encryption (FW binary)
Integrity	SHA256 (FW header and FW binary)		AES GCM Tag (FW header and FW binary)
Authentication	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)

3.5 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 strategy are needed in order to protect critical operations and sensitive data (such as a secret key), and the execution flow, in order to be resistant to possible attacks.

Figure 3 illustrates how the system, the code, and the data are protected in the X-CUBE-SBSFU application example.

Figure 3. Protections overview



3.5.1 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 example, RDP, tamper, DAP and IWDG protections are used to protect 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 HW interface to SRAM1, SRAM2, and Flash is forbidden. This prevents attacks aiming to change SBSFU code and therefore mining the Root of Trust.
  - Option bytes cannot be changed. This means that other protections such as WRP and PCROP cannot be changed anymore.

  Caution - RDP level 1 is not proposed for the following reasons:

  1. Secure Boot / Root of Trust (single entry point and immutable code) cannot be ensured, because Option bytes (WRP) can be modified in RDP L1.

  2. Device internal flash can be fully reprogrammed (after flash mass erase via RDP L0 regression) with a new FW without any security.

  3. Secrets in RAM memory protected by firewall can be accessed by attaching the debugger via the JTAG HW interface on a system reset.

  In case JTAG HW 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.
- **Tamper**: the anti-tamper protection is used to detect physical tampering actions on the device and to take related counter measures. In case of tampering detection, the SBSFU application example forces a reboot.
- **DAP** (Debug Access Port): the DAP protection consists in de-activating the DAP (Debug Access Port). Once de-activated, JTAG pins are no longer connected to the STM32 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 periodically refresh before it causes a reset. This mechanism allows the control of SBSFU execution duration.

3.5.2 Protections against inner attacks

Inner attacks refer to attacks triggered by code running in the STM32. Attacks may be due to either malicious firmware exploiting bugs or security breaches, or unwanted operations. In th SBSFU application example, WRP, firewall, PCROP, and MPU protections preserve the product from inner attacks:

- **FWALL** (firewall): the firewall is configured to protect the code, volatile data and non-volatile data. Protected code is accessible through a single entry point (the call gate mechanism is described in *Appendix A*). Any attempt to jump and try to execute any of the functions included in the code section without passing through the entry point generates a system reset.
- **PCROP** (proprietary code readout protection): a section of Flash is defined as execute-only applying PCROP protection on it: it is not possible to access this section in reading nor writing. Being an execute-only area, a key is protected with PCROP only if it is

"embedded" in a piece of code: executing this code moves the key to a specific pointer in RAM. Placed behind the firewall, its execution is not possible from outside.

- **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.
- **MPU** (memory protection unit): the MPU is used to make an embedded system more robust by splitting the memory map for Flash and SRAMs into regions having their own access rights. In the SBSFU application example, MPU is configured in order to ensure that no other code is executed from any memories during SBSFU code execution. When leaving SBSFU application to execute UserApp, MPU is de-configured.

4 Package description

This section details the X-CUBE-SBSFU package content and the way to use it.

4.1 General description

X-CUBE-SBSFU is a software package for STM32 microcontrollers.

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 X-CUBE-CRYPTOLIB.
- Two modes of operation:
  - The dual-image mode, which enables safe image programming, with firmware image backup and rollback capabilities
  - The single-image mode, which maximizes the user application size
- Integration of security peripherals and mechanisms in order to implement a SBSFU Root of Trust. RDP, WRP, PCROP, firewall, MPU, tamper, and IWDG are combined to achieve the highest security level.
- Use of a Secure Engine (SE) module as part of the middleware in order to provide a protected environment managing all critical data and operations such as secure key storage, cryptographic operations and others.
- Availability of the user application example source code.
- Availability of the firmware image preparation tool, provided both as executable and source code.

X-CUBE-SBSFU is ported on the STM32L4 Series.

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 source-code.

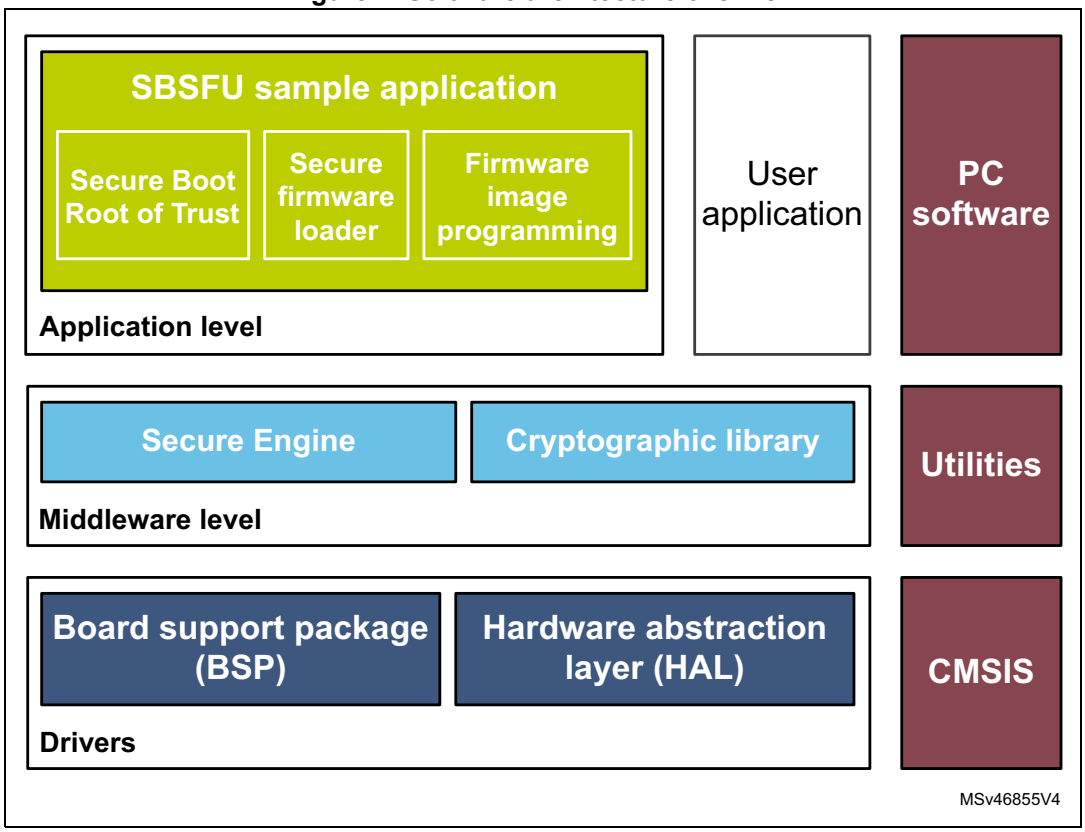
The following integrated development environments are supported:

- IAR Embedded Workbench® for Arm® (EWARM)
- Keil® Microcontroller Development Kit (MDK-ARM)
- System Workbench for STM32

4.2 Architecture

This section describes the software components of the X-CUBE-SBSFU package illustrated in *Figure 4*.

Figure 4. Software architecture overview



4.2.1 STM32CubeHAL

The HAL driver layer provides a generic multi instance simple set of APIs (application programming interfaces) 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 reusability and guarantees an easy portability onto other devices.

4.2.2 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.

4.2.3 Cryptographic Library

X-CUBE-CRYPTOLIB supports symmetric and asymmetric key approaches (AES-GCM, AES-CBC, ECDSA) as well as hash computation (SHA256) for decryption and verification. SW cryptographic functions are used to avoid storing secret key in HW Crypto IP registers that are not protected by firewall.

4.2.4 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 *Appendix A* 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.

4.2.5 Secure Boot and Secure Firmware Upgrade (SBSFU) application

Secure Boot (Root of Trust)

- Checks and applies the security mechanisms of STM32 platform to protect critical operations and secrets from attacks
- Checks the number of boots from critical failure and rollback to previous valid image (if any) in case of 3 consecutive *Boot On Error* boot sequences between two power-on
- Authenticates and verifies the user application before each execution

Local download via UART virtual COM

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

FW installation management

- Detects new FW version to install
  - From local download service via the UART interface
  - Downloaded via user application (dual-image variant only)
- Secures FW upgrade:
  - Authentication and integrity check
  - FW decryption
  - FW installation
- Supports single image for maximizing the user application size
- Supports dual image for safe image programming
  - Rollback management: in case of error during firmware installation, previous valid firmware is re-installed.
  - Multiple firmware images management: handles two firmware images (UserApp1 image and UserApp2 image) stored in internal STM32 Flash. A SWAP area is used in order to limit memory overhead needed during firmware installation or rollback procedures (refer to *Appendix B* to get details about multiple images management).

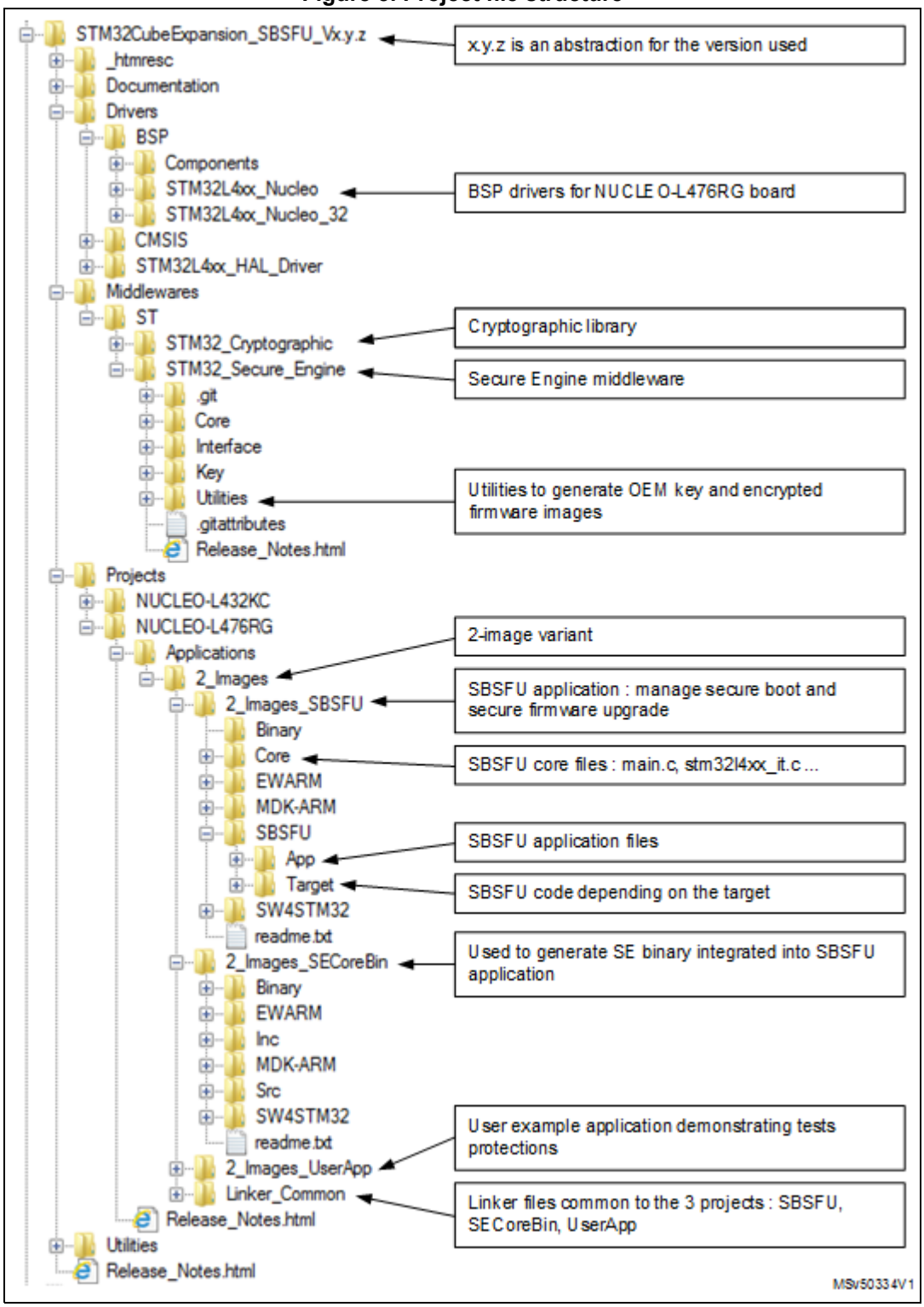
4.2.6 User application

- Provides an example for downloading the user application via Ymodem protocol over a UART (Over The Air download mechanisms, such as BLE, Wi-Fi® or others, can be implemented in the user application but are not provided as examples in the X-CUBE-SBSFU application example).
- Provides examples testing the protections mechanisms.
- Provides an example for using some of the functionalities exported by SE such as getting information about the current firmware image.
- The user application can be updated using the local download functionality of the SBSFU application or the download functionality managed by the user application itself (encrypted firmware is downloaded by the user application and the secured installation is managed by the SBSFU application).

4.3 Folder structure

A top-level view of the file structure is shown in *Figure 5*.

Figure 5. Project file structure



4.4 APIs

Detailed technical information about the APIs available to the user is provided in a compiled HTML file located in the *Documentation* folder of the software package where all the functions and parameters are described.

4.5 Application compilation process with IAR™ toolchain

Figure 6 outlines the steps needed in order to build the application and to demonstrate Secure Boot and Secure Firmware Update:

- Step 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 firewall code section. The SE Callgate function is specified as the entry point for the binary. The binary is linked with the SBSFU code in step 2.

- Step 2: SBSFU

  This step compiles the SBSFU source code implementing the state machine and configuring the protections. In addition, it links the code with the SECore binary generated at step 1 in order 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.

- Step 3: user application example

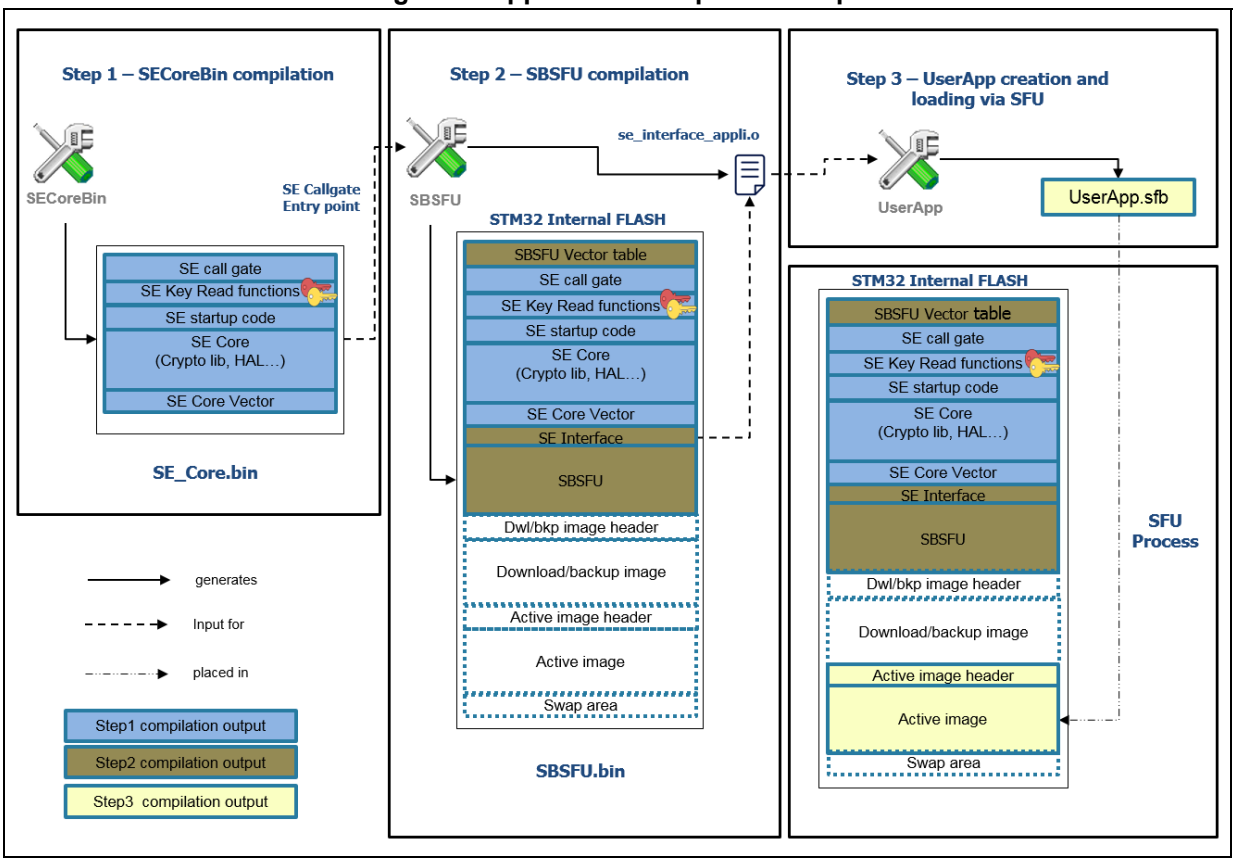
  It generates:

  – 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 user application binary in clear format, and the corresponding FW header.

  These three elements are placed properly for both the SBSFU and user application to run when the binary file is flashed into the device with a flasher tool. Hence, no FW installation procedure is required for SBSFU to start and boot the user application. This is a convenient way to test the user application with a single flashing stage.

Figure 6. Application compilation steps



5 Hardware and software environment setup

This section describes the hardware and software setup procedures.

5.1 Hardware Setup

To set up the hardware environment, one of the supported boards introduced in *Section 4.1: General description on page 16* must be connected to a personal computer via a USB cable. This connection with the PC allows the user:

- Flashing the board
- Interacting with the board via a UART console
- Debugging when the protections are disabled

5.2 Software setup

This section lists the minimum requirements for the developer to setup the SDK, run the sample scenario, and customize applications.

5.2.1 Development toolchains and compilers

Select one of the Integrated Development Environments supported by the STM32Cube Expansion Package.

Take into account the system requirements and setup information provided by the selected IDE provider.

5.2.2 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*.

Caution: Make sure to use an up-to-date version of ST-LINK (V2.J27 or later).

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, RAM, and OTP) as well as external memories. 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 STM32CubeProg STM32CubeProgrammer software on *www.st.com*.

5.2.3 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 that can be downloaded from the https://osdn.net/projects/ttssh2/ webpage. Any other similar tool can be used instead (Ymodem protocol support is required).

5.2.4 X-CUBE-SBSFU firmware image preparation tool

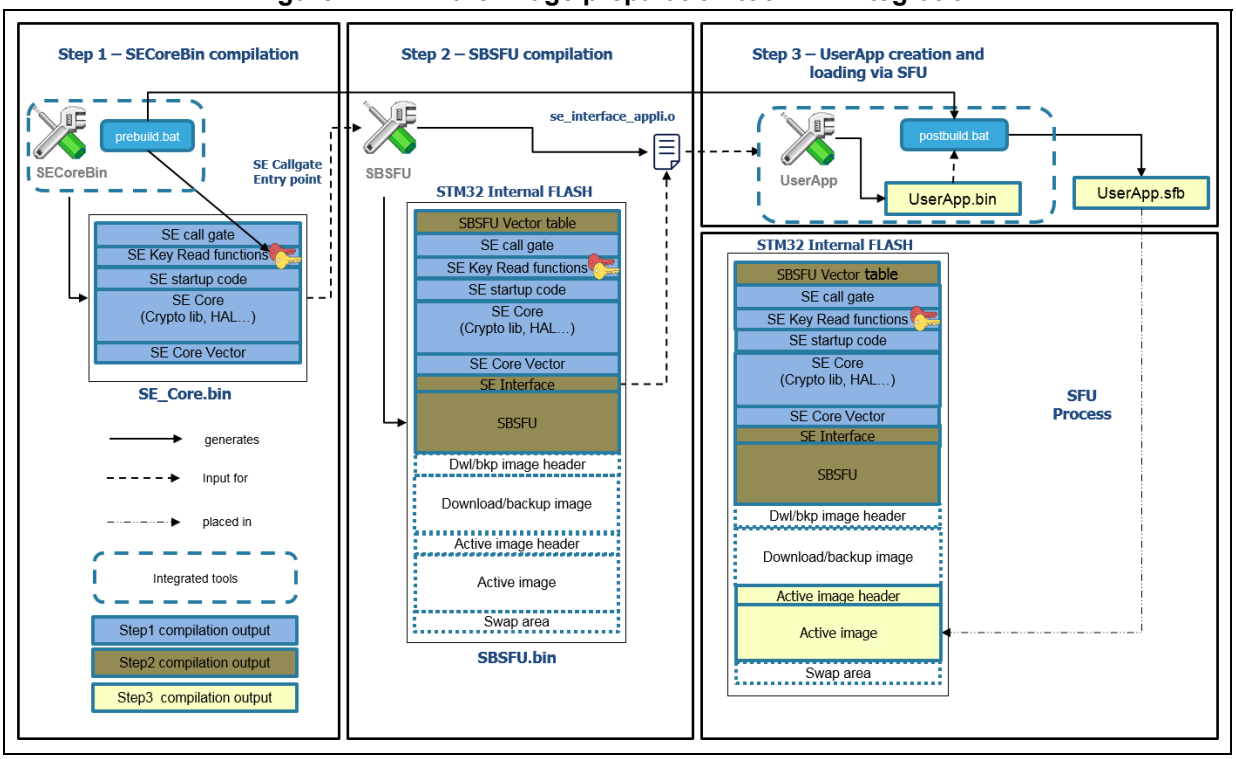
The X-CUBE-SBSFU Expansion Package for STM32Cube is delivered with the *prepareimage* tool handling the cryptographic keys and firmware image preparation.

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 in the supported IDEs and compilation process as shown in *Figure 7*.

Figure 7. Firmware image preparation tool IDE integration



More information about the preparation tool are provided in *Appendix E: Firmware image preparation tool on page 49*.

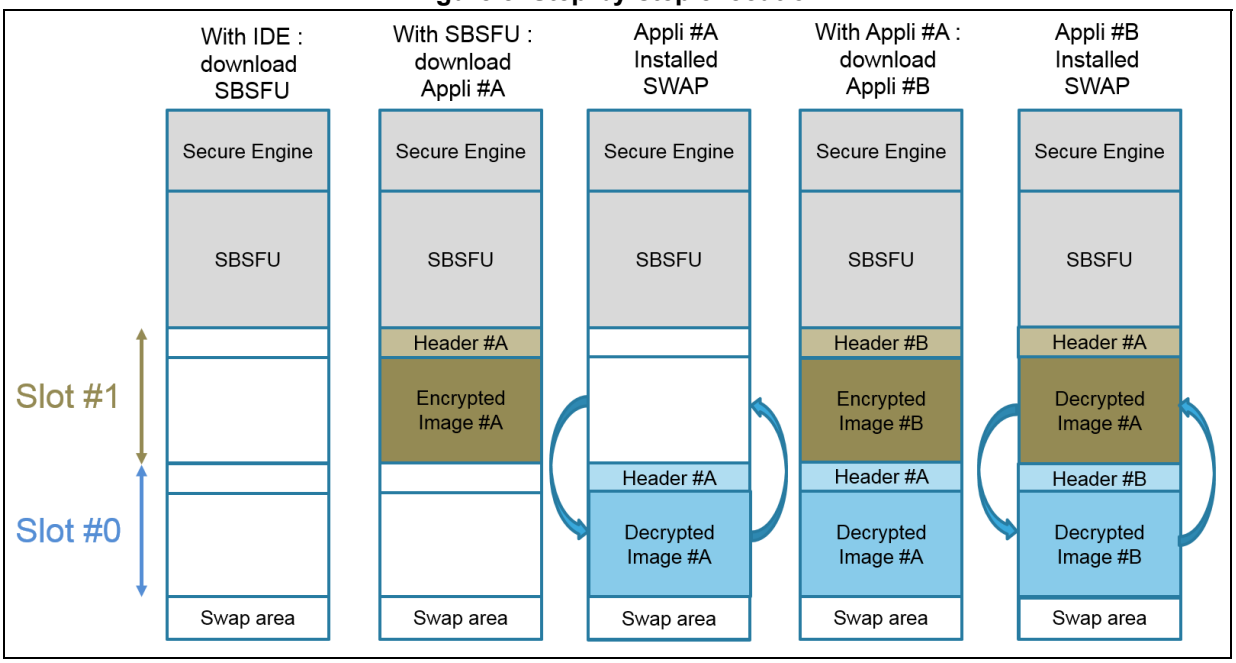
6 Step-by-step execution

The followings steps describe a dual-image SBSFU scenario with the default cryptographic scheme, further illustrated in *Figure 8*:

1. Download SBSFU application
2. SBSFU is running : download UserApp #A
3. UserApp #A is installed
4. UserApp#A is running, download UserApp #B
5. UserApp #B is installed then running

The UserApp#A and UserApp#B binaries are generated on the basis of the 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 application.

Figure 8. Step-by-step execution



6.1 STM32 board preparation

The target option bytes setting is the following:

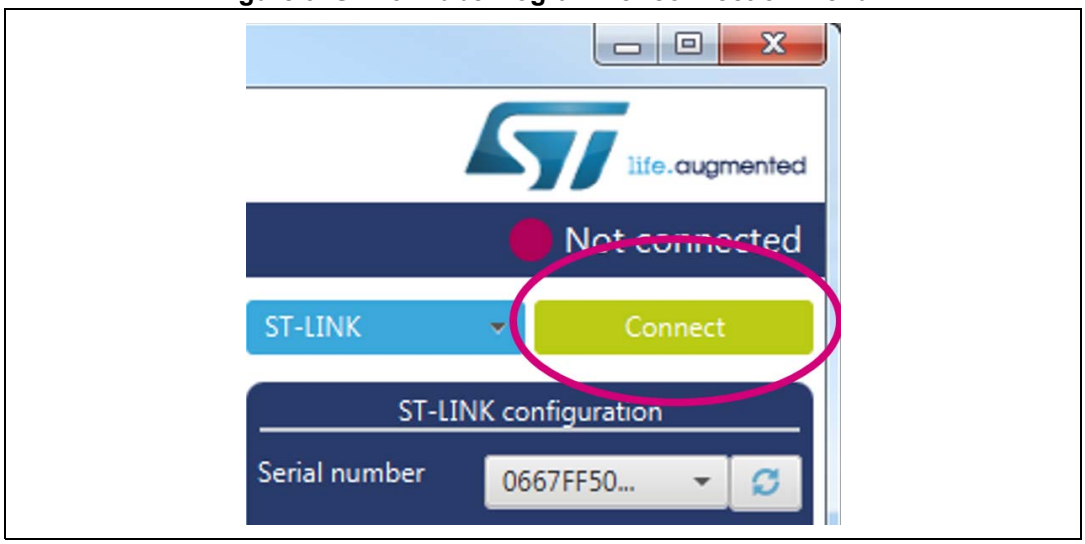
- RDP Level 0 is set
- Write protection is disabled on all Flash pages
- PCROP protection is disabled[a]
- Chip is erased[a]

a. Automatically done when switching from RDP level 1 to RDP level 0.

Option bytes setting is verified by means of the STM32CubeProgrammer through the following four steps:

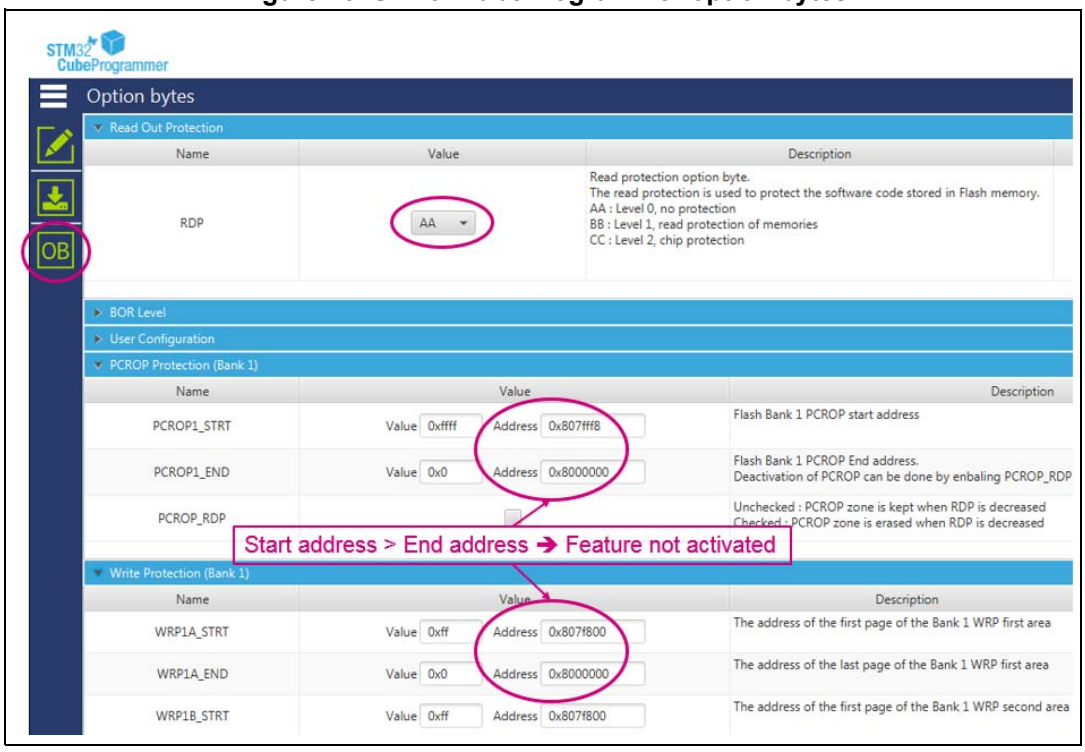
1. Connection: Menu Target / Connect (refer to *Figure 9*)

Figure 9. STM32CubeProgrammer connection menu



2. Option Bytes settings : Menu Target / Option Bytes (refer to *Figure 10*)

Figure 10. STM32CubeProgrammer option bytes



3. Erase chip: Menu Target / Erase Chip

Figure 11. STM32CubeProgrammer erasing

4. Disconnect: Menu Target / Disconnect

6.2 Application compilation

With the selected toolchain (IAR, Keil, or System Workbench) rebuild all the projects as explained in *Section 4.5: Application compilation process with IAR™ toolchain on page 21*.

Download the SB SFU project software to the target without starting a debug session (Security protections managed by SBSFU forbid JTAG connection as it is interpreted as an external attack).

6.3 Tera Term connection

Tera Term connection is achieved by applying in sequence the steps described from *Section 6.3.1* to *Section 6.3.4*.

6.3.1 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 VJ26M15 onwards[(a)]:

a) Power cycle the board after flashing SBSFU (unplug/plug the USB cable).

b) 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.

c) 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.

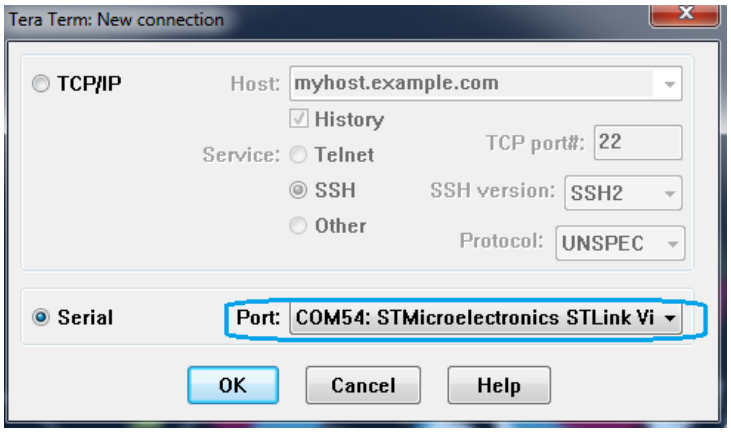
a. Make sure the ST-LINK debugger/programmer embedded on the board runs the proper firmware version (VJ26M15 or higher). If this is not the case, please upgrade this firmware first.

6.3.2 Tera Term launch

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

Figure 12 illustrates an example based on the selection of port COM54.

Figure 12. Tera Term connection screen

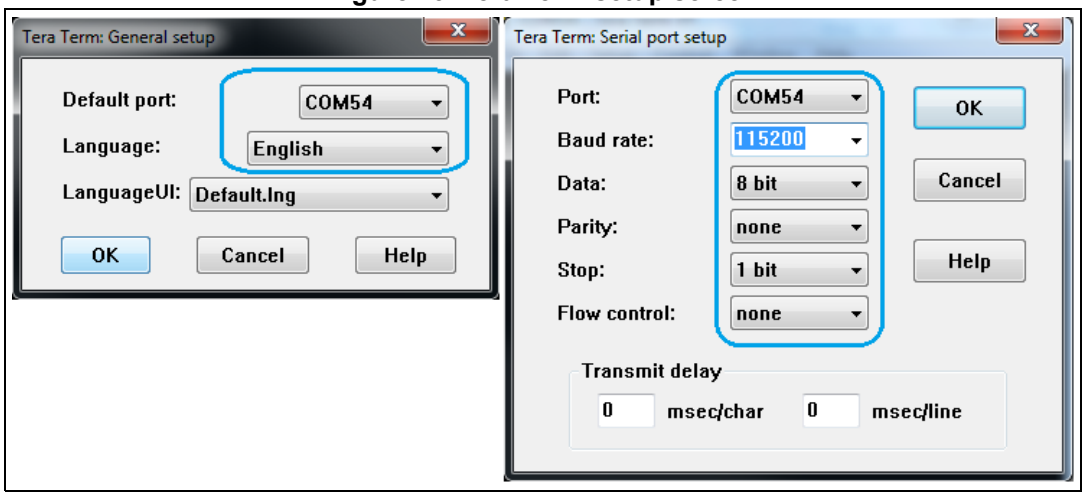


6.3.3 Tera Term configuration

The Tera Term configuration is performed through the *General* and *Serial port* setup menus.

Figure 12 illustrates the *General setup* and *Serial port setup* menus.

Figure 13. Tera Term setup screen



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

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

6.3.4 Welcome screen display

The welcome screen is displayed on Tera Term as illustrated in *Figure 14*

Figure 14. SBSFU welcome screen display

6.4 SBSFU application execution

The SBSFU state machine is detailed in *Appendix F*.

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.

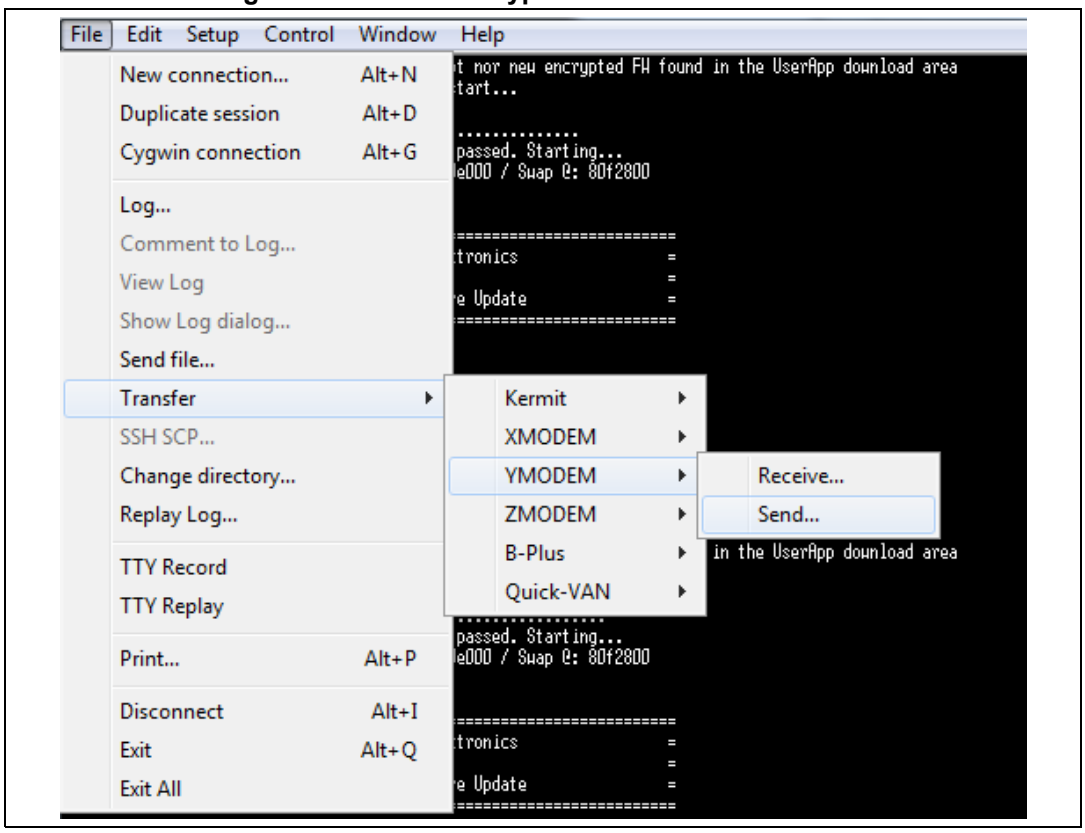
6.4.1 Download request

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

6.4.2 Send firmware

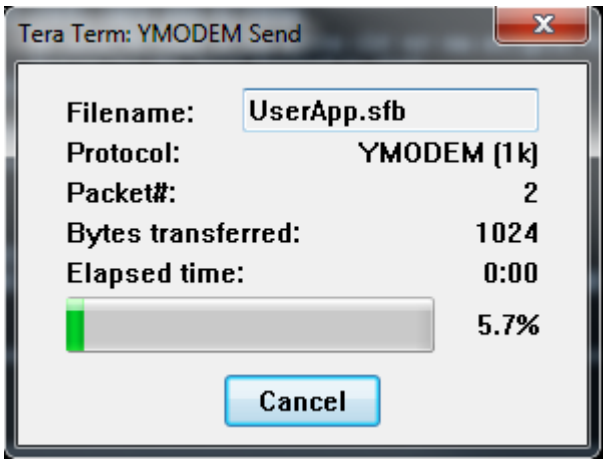
For sending the firmware (**.sfb*), use the *File* > *Transfer* > *YMODEM* > *Send* menu in Tera Term as shown in *Figure 15*.

Figure 15. SBSFU encrypted firmware transfer start



Once the *UserApp.sfb* file is selected, the Ymodem transfer starts. Transfer progress is reported as shown in *Figure 16*.

Figure 16. SBSFU encrypted firmware transfer in progress



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 slot where the firmware image is downloaded.

6.4.3 File transfer completion

After the file transfer is completed, the system forces a reboot as shown in *Figure 17*.

Figure 17. SBSFU reboot after encrypted firmware transfer

The system status that is printed as shown in *Figure 17* 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 user firmware is executed.

6.4.4 System restart

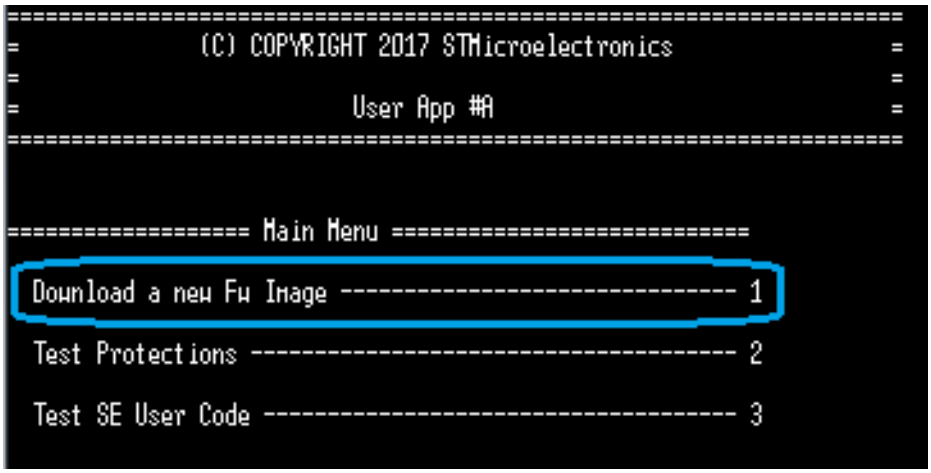
Pressing the Reset button forces the system to restart: the user application is started by SBSFU.

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

6.5 User application execution

The user application is executed according to the selection illustrated in *Figure 18* and further described from *Section 6.5.1* to *Section 6.5.3*.

Figure 18. User application execution

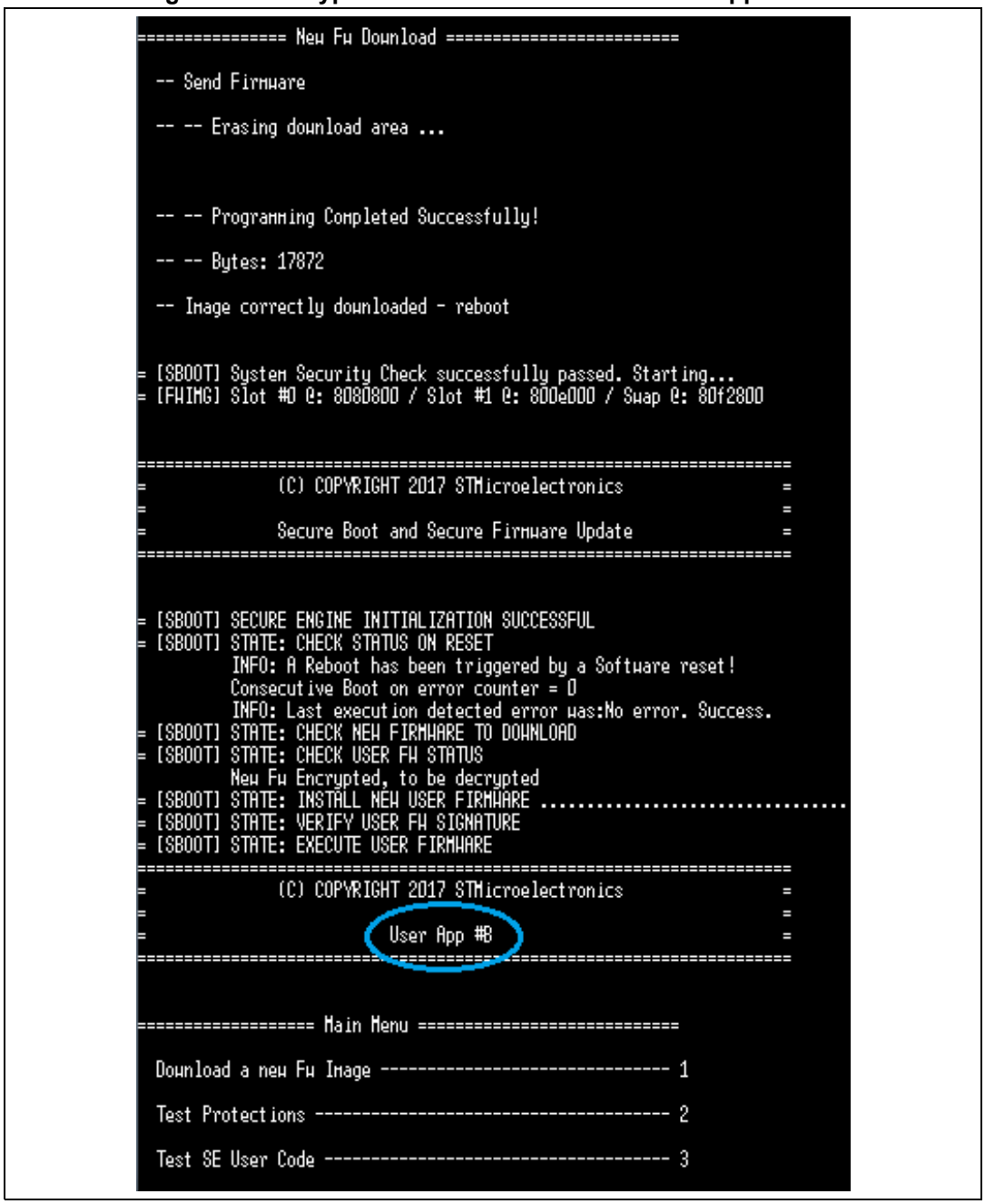


6.5.1 Download a new firmware image

The download of a new firmware image is performed through the same steps as those presented for SBSFU in *Section 6.4 on page 29*:

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

Figure 19. Encrypted firmware download via user application



6.5.2 Test protections

The test protection menu is shown in *Figure 20*.

Figure 20. User application test protection menu

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

- Firewall tests (#1, #2)
  - Causes a reset trying to access protected code or data (either in RAM or Flash)
- PCROP test (#3)
  - Causes an error trying to access the PCROP region protecting the keys
- WRP test (#4)
  - Causes an error trying to erase write protected code
- IWDG test (#5)
  - Causes a reset simulating a deadlock by not refreshing the watchdog
- TAMPER test (#6)
  - Causes a reset if a tamper event is detected
  - In order to generate a tamper event, the user must connect PA0 (CN7.28) to GND (It may be enough to put a finger close to CN7.28).
- CORRUPT IMAGE test (#7)
  - Causes a signature verification failure at next boot.

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

6.5.3 Test Secure Engine user code

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

7 Understanding the last execution status message at boot-up

Table 4 lists the main error messages together with their explanation.

Table 4. Error messages at boot-up

Error message	Meaning
No error. Success.	No problem encountered.
Firewall error.	A firewall exception occurred: some code or data protected by the firewall has been addressed out of the Secure Engine context.
Watchdog error.	Watchdog expiry: a processing is too long and the watchdog has not been reloaded in due time.
Memory fault.	Memory fault 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 up (generic error).
Check new user FW to download error.	Error encountered while checking if there is a local download request (generic error).
Download new user FW error.	Error encountered while performing a local download (generic error).
Verify user FW status error.	Error encountered while verifying the status of the user firmware. This error is reached when the Flash state does not allow determining the firmware status (generic error).
Decrypt user FW 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 FW error.	Error encountered during the installation of a 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.
Rollback prev user FW error.	Error encountered during a recovery procedure (generic error).
Execute user FW error.	Error encountered while trying to launch the active firmware (generic error).
Max. consecutive errors reached.	The system has reached the maximum number of consecutive errors (either in the firmware or in the SBSFU context). In the example, this means that 3 consecutive IWDG or firewall exceptions have occurred.
SE lock cannot be set.	Error encountered while trying to configure Secure Engine in “Firmware execution” mode (unprivileged mode) before starting the active firmware.
Inconsistent FW size.	This error means that during a local download procedure the size indicated in the header does not match the size of the downloaded file.
FW too big.	This error means that during a local download procedure the header indicated a firmware size bigger than the capacity of the slot #1.

Table 4. Error messages at boot-up (continued)

Error message	Meaning
Ymodem com failure.	During a local download procedure, the download operation did not complete successfully (Ymodem protocol issue).
File not correctly received.	During a local download procedure, the binary file has not been received properly. At the moment this detection is minimalist.
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 slot #1. This error reports a decryption or an 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 should not be reached as a signature issue would be captured at decrypt stage (reporting "`Decrypt failure.`").
Incorrect binary format (not encrypted).	Error encountered during an installation procedure: the binary present in slot #1 is not encrypted. Maybe the clear version of the firmware instead of the encrypted version was downloaded?
Flash error.	Flash error encountered during an installation procedure.
FWIMG pattern issue.	Error encountered during an installation procedure: internal issue while writing some SBSFU patterns.
Error while swapping the images in slot #0 and slot #1.	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 area that is shared with user application.
  - 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 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

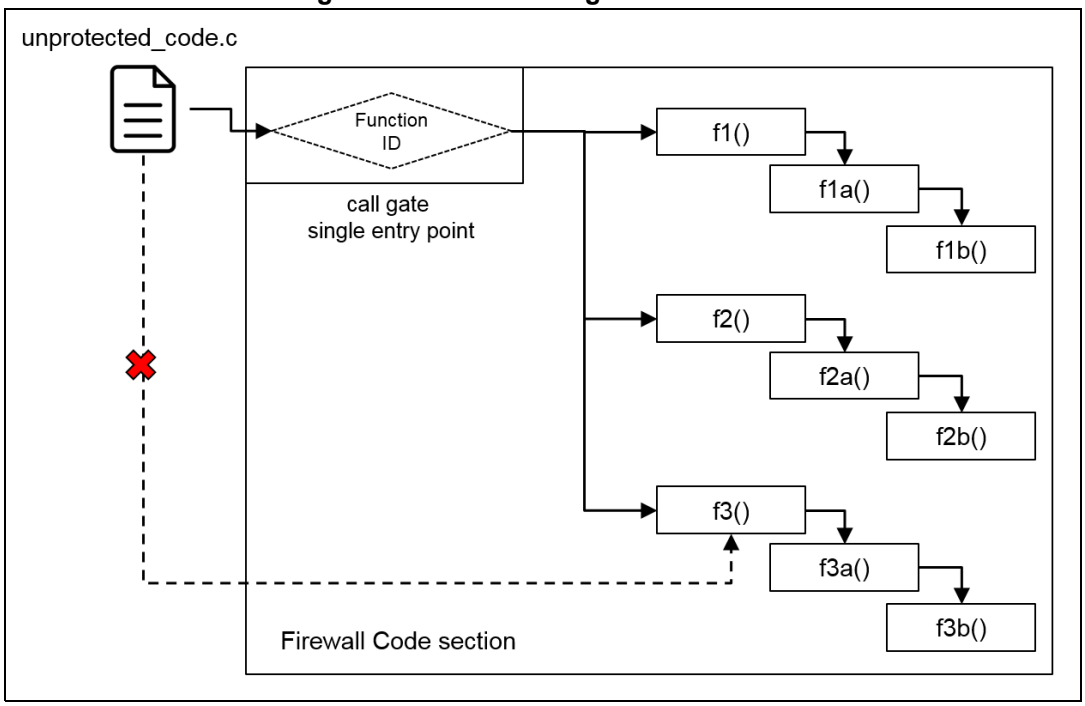
In order to deal with the firewall callgate mechanism and to provide the user with a set of secure APIs, SE is designed with a two-level architecture, composed of SE Core and SE Interface.

A.1 SE core call gate mechanism

The firewall is opened or closed using a specific "call gate" mechanism: a single entry point (placed at the 2nd word of the Code segment base address) must be used to open the gate and to execute the code protected by the firewall. If the protected 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 protected by the firewall and called from unprotected code outside it (e.g. 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 21. Firewall call gate mechanism



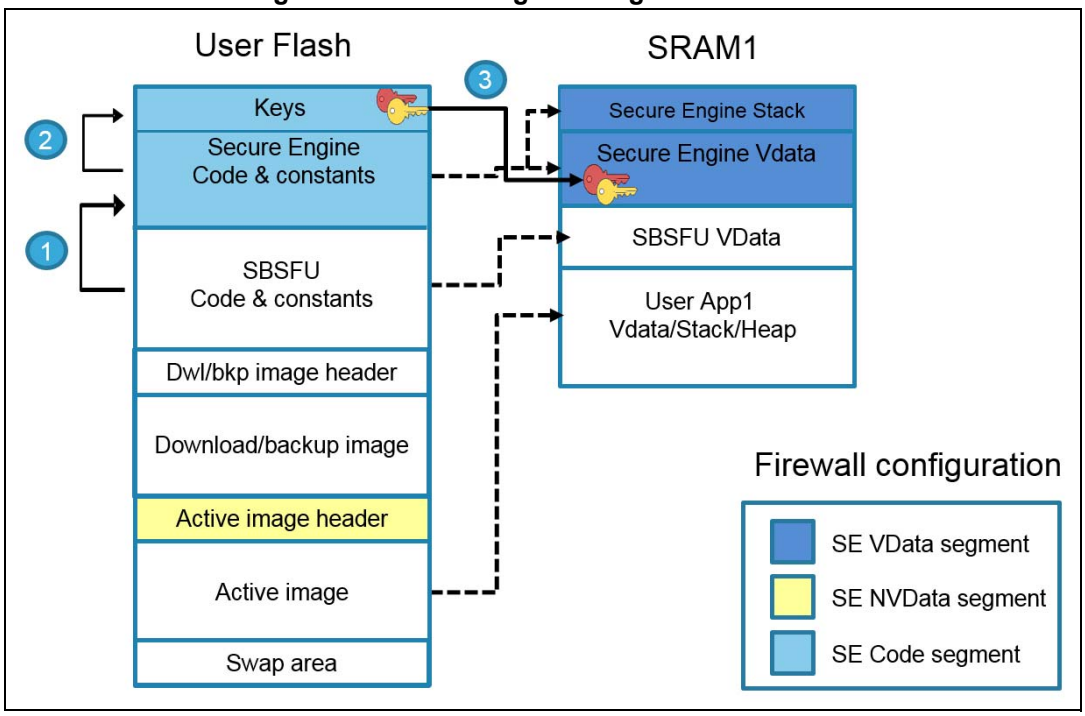
Caution: The code section must include all the code executed when the firewall is open. For instance, if the call sequence is callgate->f1()->f1a()->f1b(), all the three functions f1(), f1a() and f1b() must be included in the code section.

Figure 22 shows the steps to perform cryptographic operations (that require access to the key) in order to respect the call gate mechanism.

For the cryptographic functions:

1. The SBSFU code calls the call gate function in order to open the firewall and to execute protected code
2. The call gate function check parameters and securities and then calls the requested Crypto function
3. The SE Crypto functions calls an internal ReadKey function that moves the keys into the protected section of SRAM1 and then use them in the cryptographic operations.

Figure 22. Secure Engine call-gate mechanism

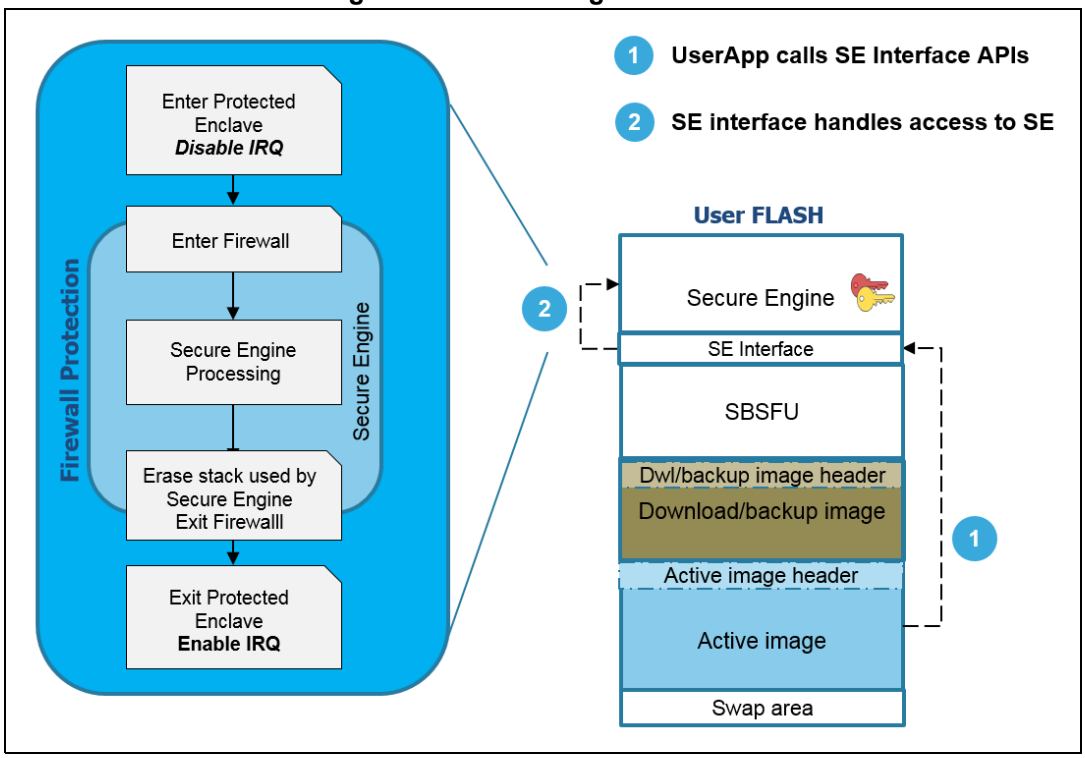


A.2 SE interface

Code protected by the firewall must be non-interruptible and it is up to the user code to disable interrupts before opening the firewall.

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 *Figure 23*.

Figure 23. 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.

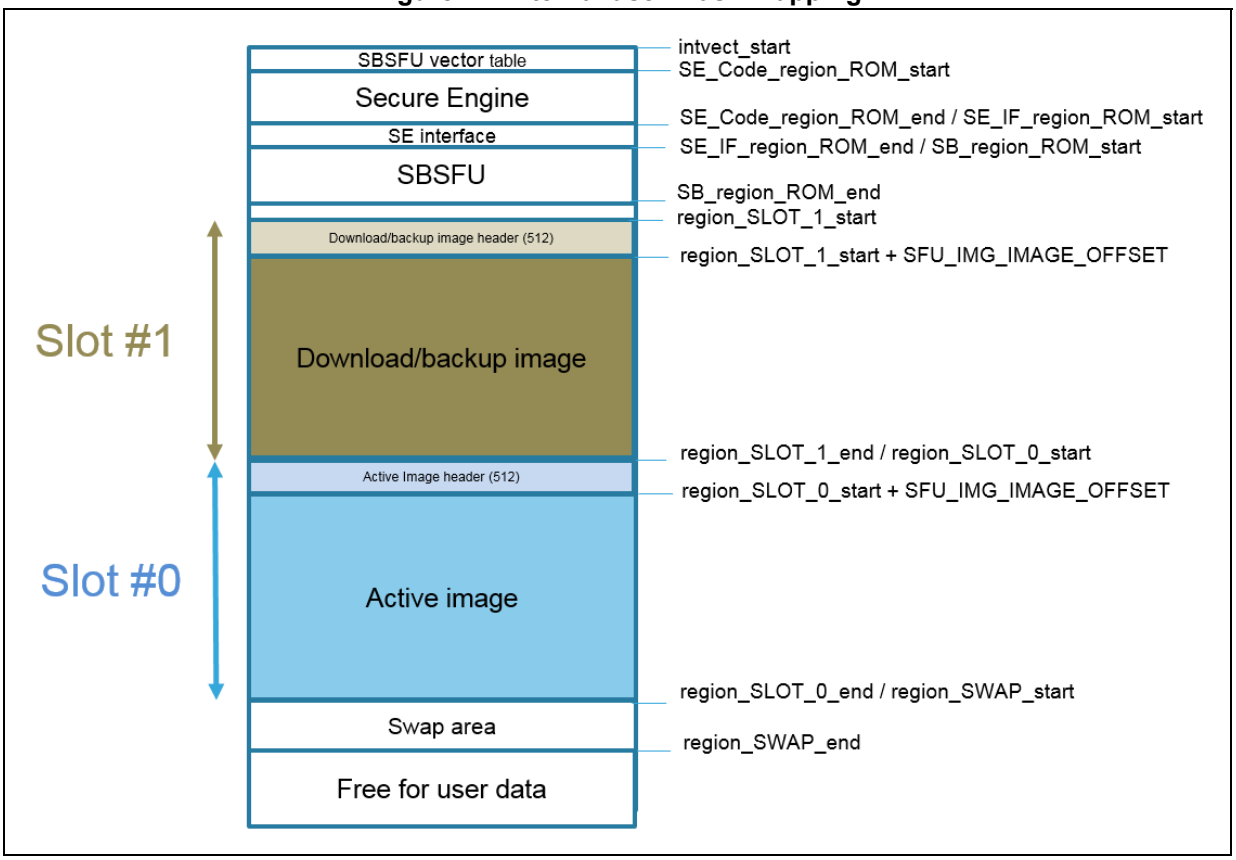
Appendix B Dual-image handling

Some SBSFU application examples handle two firmware images stored in internal Flash.

B.1 Elements and Roles

- Slot #0:
  - This slot contains the active firmware (firmware header + firmware). This is the user application that is launched at boot time by SBSFU (after verifying its validity).
- Slot #1:
  - After a download procedure, this slot is used to store the downloaded firmware (firmware header + encrypted firmware) to be installed at next reboot.
  - After an installation procedure, this slot contains the backed-up firmware (clear form) until a rollback procedure occurs or a new download procedure is triggered.
- Swap Region:
  - This is a Flash area used to swap the content of Slot #0 and Slot #1.
  - Nevertheless, this area is not a buffer used for each and every swap of Flash sector. It is used to move a first sector, hence creating a shift in Flash allowing swapping the two slots sector by sector.

Figure 24. Internal user Flash mapping



B.2 Mapping definition

The Flash memory is organized as indicated in *Table 5*, which shows figures coming from the IAR example projects for some STM32L4 Series products.

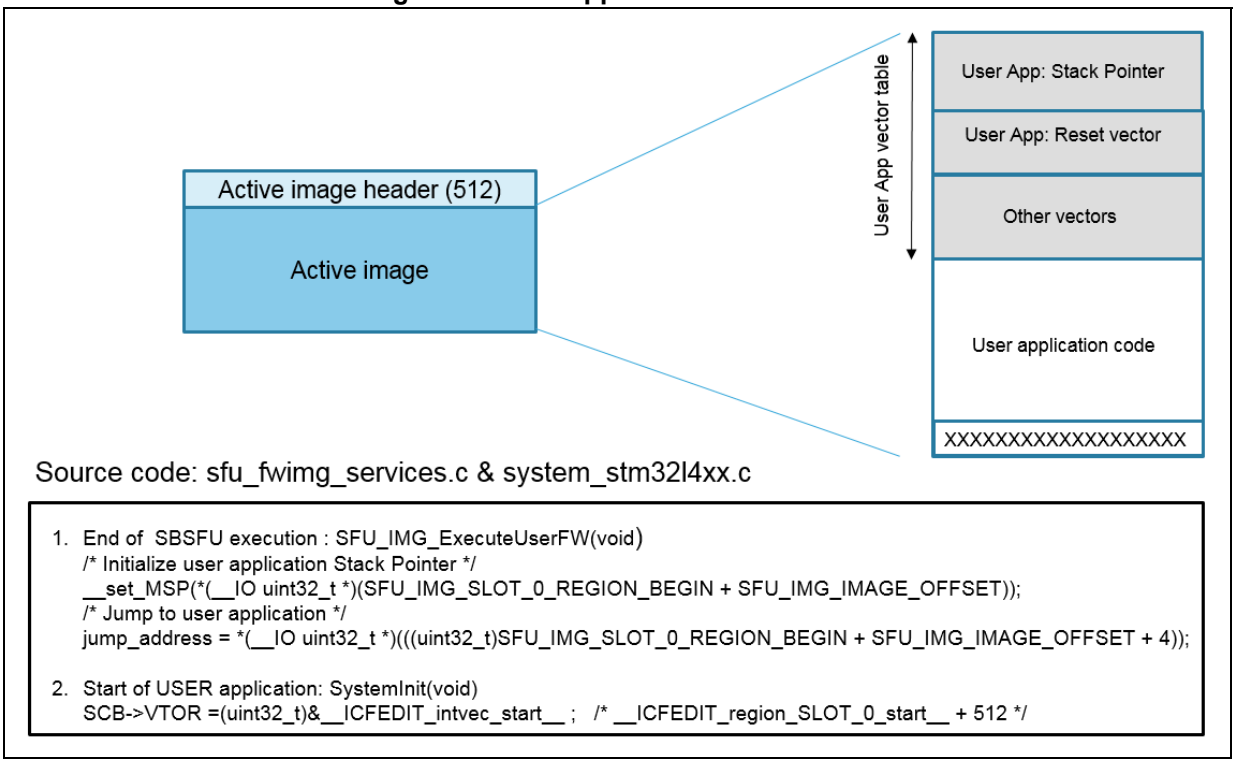
Table 5. Dual-image Flash organization

	NUCLEO-L476RG	NUCLEO-L432KC	B-L475E-IOT01A	32L496GDISCOVERY
Code start address	0x0800 0000	0x0800 0000	0x0800 0000	0x0800 0000
Code size	54 Kbytes	54 Kbytes	54 Kbytes	58 + 4 Kbytes(1)
Slot #0 size	456 Kbytes	96 Kbytes	456 Kbytes	448 Kbytes
Slot #1 size	456 Kbytes	96 Kbytes	456 Kbytes	448 Kbytes
Swap region size	8 Kbytes	4 Kbytes	8 Kbytes	16 Kbytes

1. Extra 4 Kbytes are due to the BSP (I/O expander) of the 32L496GDISCOVERY board.

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 25*).

Figure 25. User application vector table



Source code: sfu_fwimg_services.c & system_stm32l4xx.c

```
1. End of  SBSFU execution : SFU_IMG_ExecuteUserFW(void)
   /* Initialize user application Stack Pointer */
   __set_MSP(*(__IO uint32_t *)(SFU_IMG_SLOT_0_REGION_BEGIN + SFU_IMG_IMAGE_OFFSET));
   /* Jump to user application */
   jump_address = *(__IO uint32_t *)(((uint32_t)SFU_IMG_SLOT_0_REGION_BEGIN + SFU_IMG_IMAGE_OFFSET + 4));

2. Start of USER application: SystemInit(void)
   SCB->VTOR =(uint32_t)&__ICFEDIT_intvec_start__ ;   /* __ICFEDIT_region_SLOT_0_start__ + 512 */
```

Appendix C Single-image handling

Some SBSFU application examples handle one single firmware image stored in internal Flash.

This mode of operation allows maximizing the user firmware size by:

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

These benefits come at the cost of some features:

- Safe firmware image programming cannot be ensured:
  - No backup of the active firmware image when an update is triggered
  - No rollback possibility if the active firmware becomes invalid
- 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

Slot #0:

- This slot contains the active firmware (firmware header + firmware). This is the 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)

C.2 Mapping definition

The Flash memory is organized as indicated in *Table 6*, which shows figures coming from the IAR example projects for some STM32L4 Series products.

Table 6. Single-image Flash organization

	NUCLEO-L476RG	NUCLEO-L432KC	B-L475E-IOT01A	32L496GDISCOVERY
Code start address	Not supported(1)	0x0800 0000	Not supported(1)	Not supported(1)
Code size		47 Kbytes		
Slot #0 size		208 Kbytes		

1. Single-image variant cannot be supported because on dual-bank Flash memory devices:
   - The firewall code segment must be located in bank 1, and the firewall data segment in bank 2.
   - The firewall code and data segments must be located at the same offset from base address in each bank (ensuring that secrets are always protected even if the banks are swapped)

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 25: User application vector table*).

Appendix D Cryptographic schemes handling

Three cryptographic schemes are provided as example 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 and can be selected thanks to a SECoreBin compiler switch (named "SECBOOT_CRYPTO_SCHEME").

D.1 Cryptographic schemes contained in this package

Table 7 shows the cryptographic scheme selected with the `SECBOOT_CRYPTO_SCHEME` compiler switch.

Table 7. Cryptographic scheme list

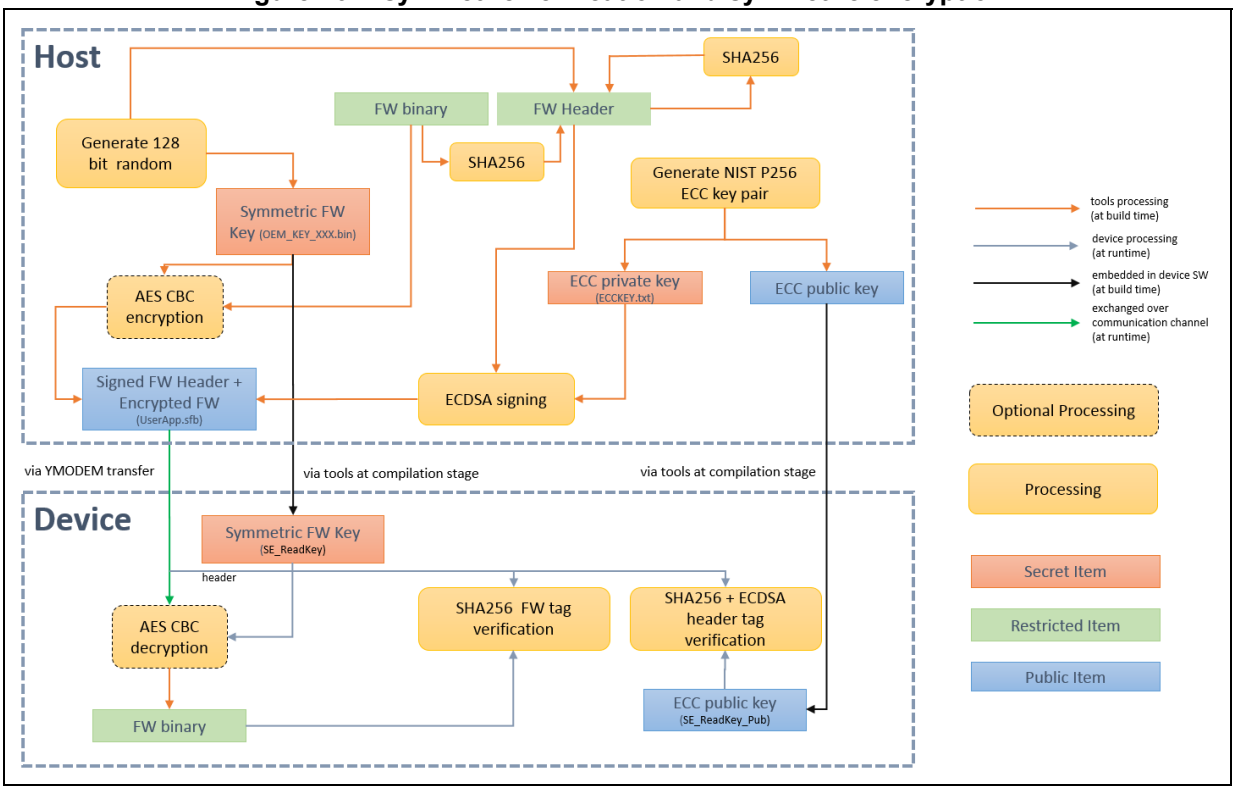
SECBOOT_CRYPTO_SCHEME value	Authentication	Confidentiality	Integrity
SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256 (default)	ECDSA	AES128-CBC	SHA256
SECBOOT_ECCDSA_WITHOUT_ENCRYPT_SHA256	ECDSA	None(1)	SHA256
SECBOOT_AES128_GCM_AES128_GCM_AES128_GCM	AES GCM		

1. The SBSFU project must also be configured to deal with a clear firmware image by setting the SFU_IMAGE_PROGRAMMING_TYPE compiler switch to the value SFU_CLEAR_IMAGE.

D.2 Asymmetric verification and symmetric encryption schemes

These schemes (SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256, SECBOOT_ECCDSA_WITHOUT_ENCRYPT_SHA256) are implemented for firmware decryption and verification as illustrated in *Figure 26*

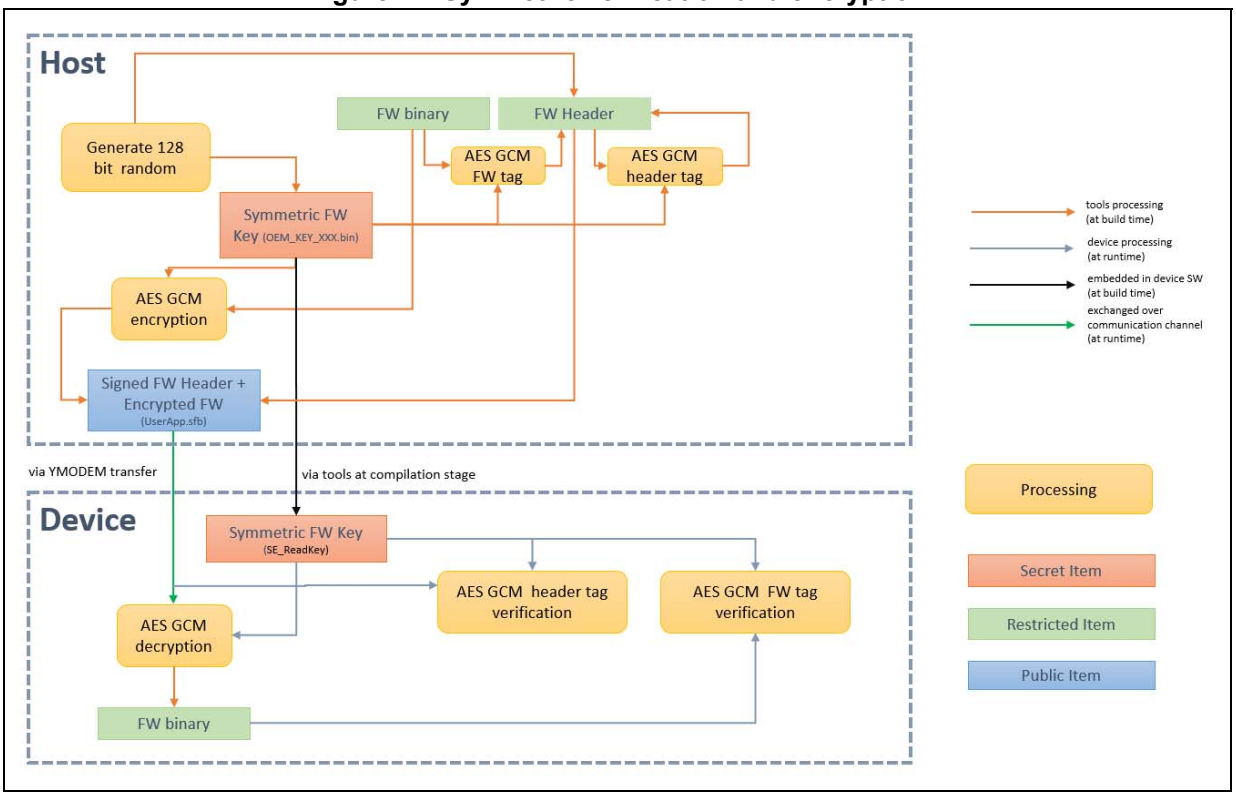
Figure 26. Asymmetric verification and symmetric encryption



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 in *Figure 27*

Figure 27. Symmetric verification and encryption



D.4 Secure Boot and Secure Firmware Update flow

Figure 28 and *Figure 29* indicate how the cryptographic operations (asymmetric cryptographic scheme with FW encryption) are integrated in the SBSFU execution boot flows.

Figure 28. SBSFU dual-image boot flows

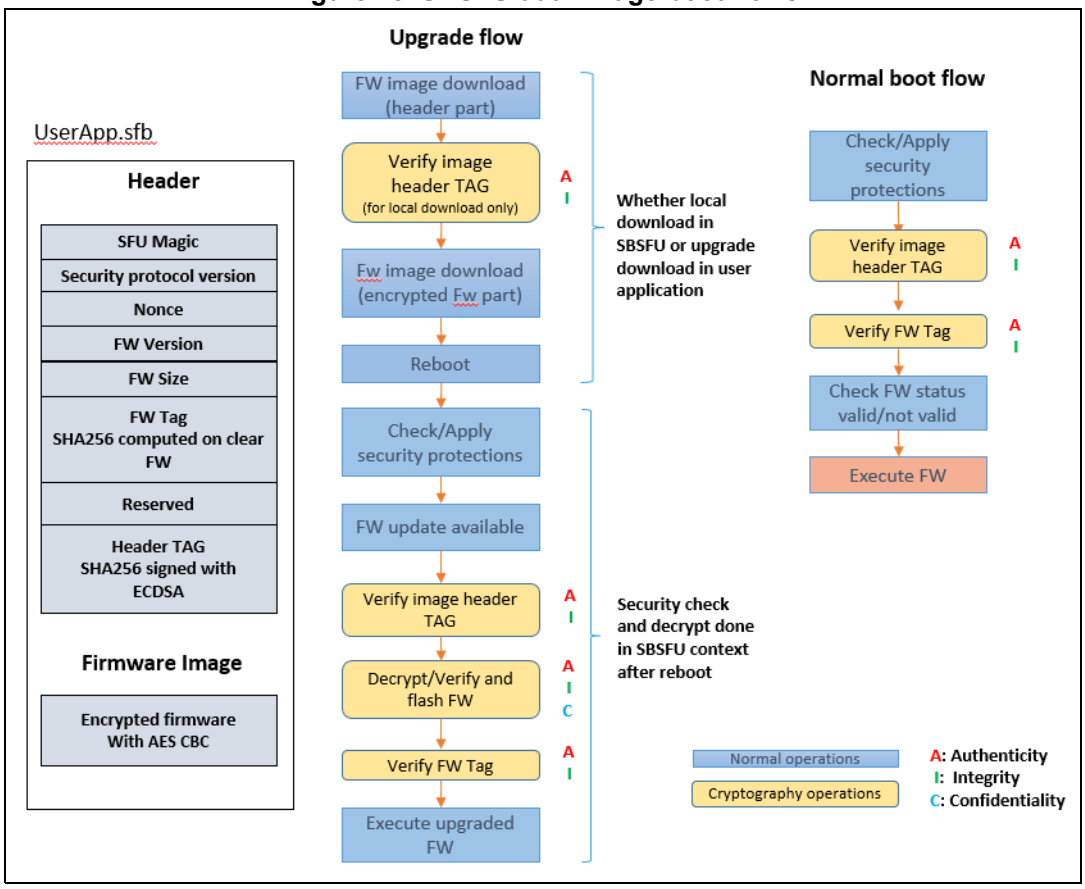
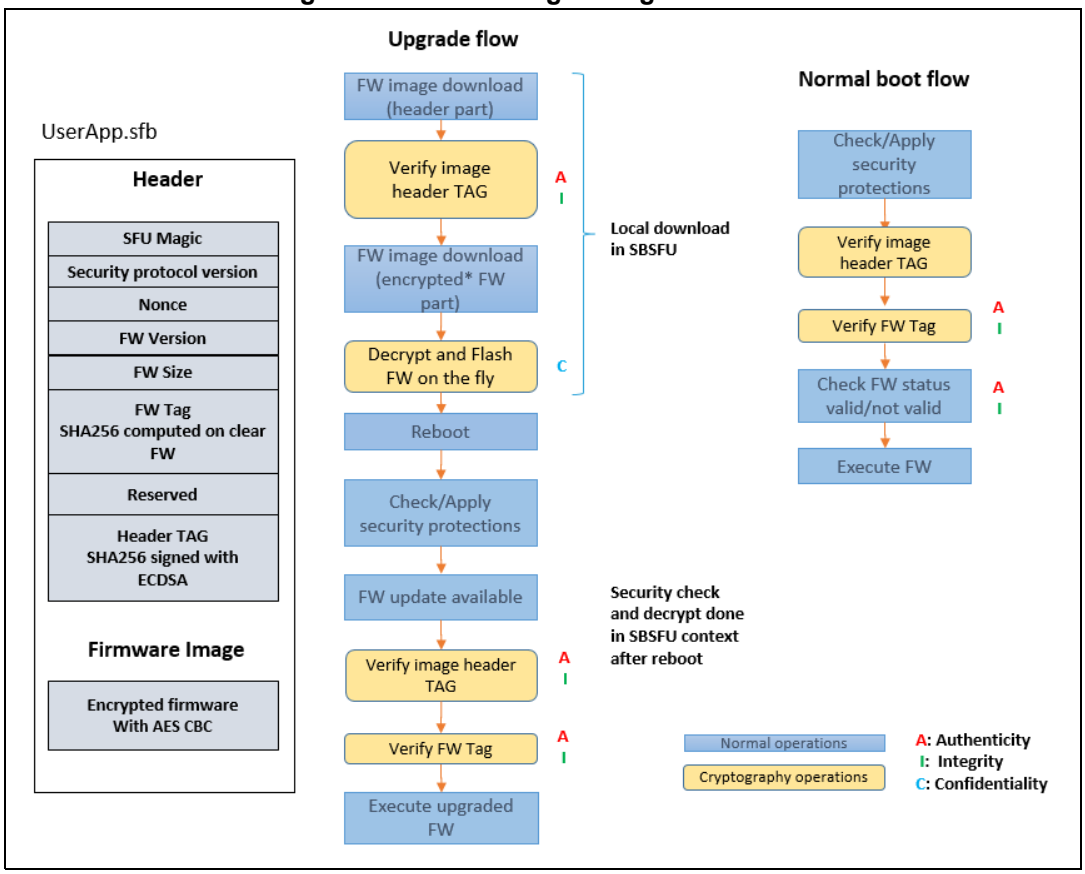


Figure 29. SBSFU single-image boot flows



Appendix E Firmware image preparation tool

The X-CUBE-SBSFU STM32Cube Expansion Package is delivered with the *prepareimage* firmware image preparation tool allowing:

- Taking into account the selected cryptographic scheme and keys
- Encrypting the firmware image when required
- Generating 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 a 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 in a flexible manner.

E.1 Tool location

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

E.2 Inputs

The package is delivered with some default keys and cryptography settings in folder *\Projects\ NUCLEO-L476RG \Applications\2_Images\2_Images SECoreBin\Binary*.

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

- *ECCKEY.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 file *se_key.s*
- *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 file *se_key.s*. 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 file *se_key.s* . 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 by means of `SECBOOT_CRYPTO_SCHEME` in file *Projects\ NUCLEO-L476RG \Applications\2_Images\2_Images SECoreBin\Inc\se_crypto_config.h*.

E.3 Outputs

The tool generates:

- The *se_key.s* 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 *Projects\ NUCLEO-L476RG \Applications\2_Images \2_Images SECoreBin\Src*.
- A *.sfb* file packing the 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 *Projects\ NUCLEO-L476RG \Applications\2_Images\2_Images UserApp\Binary*.
- A *.bin* file concatenating the SBSFU binary, UserApp binary, and active FW image header. Flashing this file into the device with a flasher tool makes the UserApp installation process simple, since the FW header and FW image are already correctly installed. It is not needed to use the SBSFU application for installing the UserApp.

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* application: at this stage, the firmware image is built

When compiling with the IDE, the keys and 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 image to be installed is generated in the proper format, with the appropriate firmware header, as a *.sfb* file. This *.sfb* file 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 folder *\Projects\ NUCLEO-L476RG \Applications\2_Images\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* project

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.

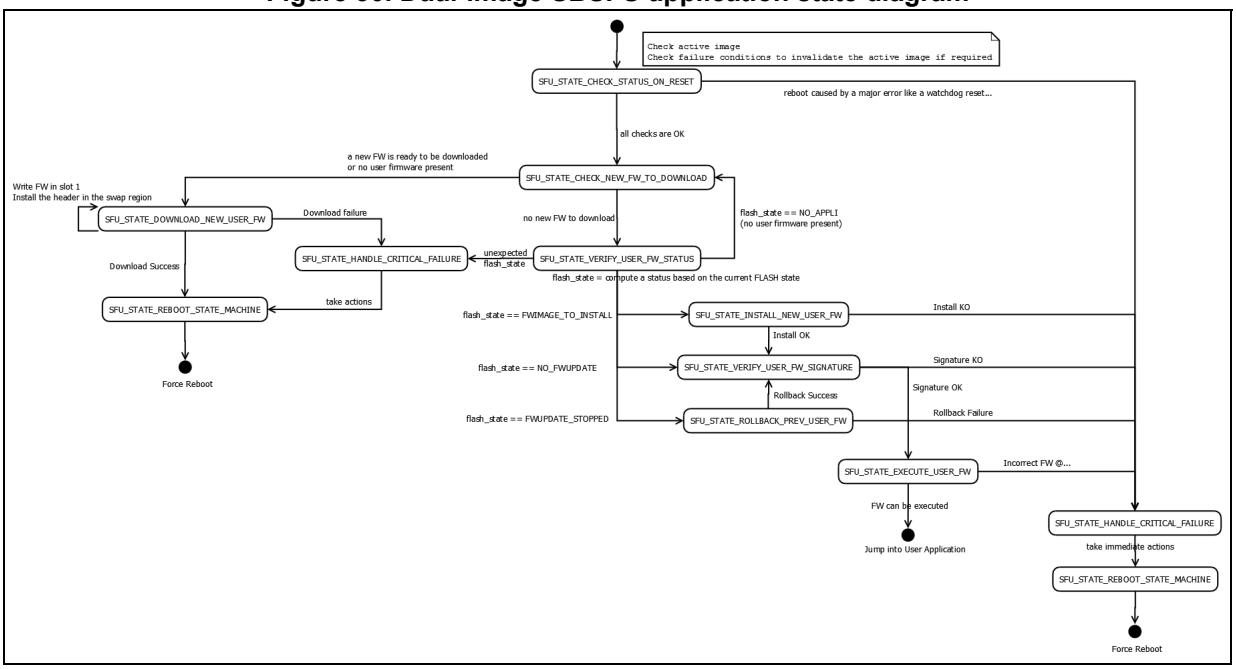
Appendix F SBSFU application state machine

This section describes the states of the SBSFU application state machine provided in the X-CUBE-SBSFU package.

F.1 Dual-image SBSFU

The corresponding state diagram is illustrated in *Figure 30*.

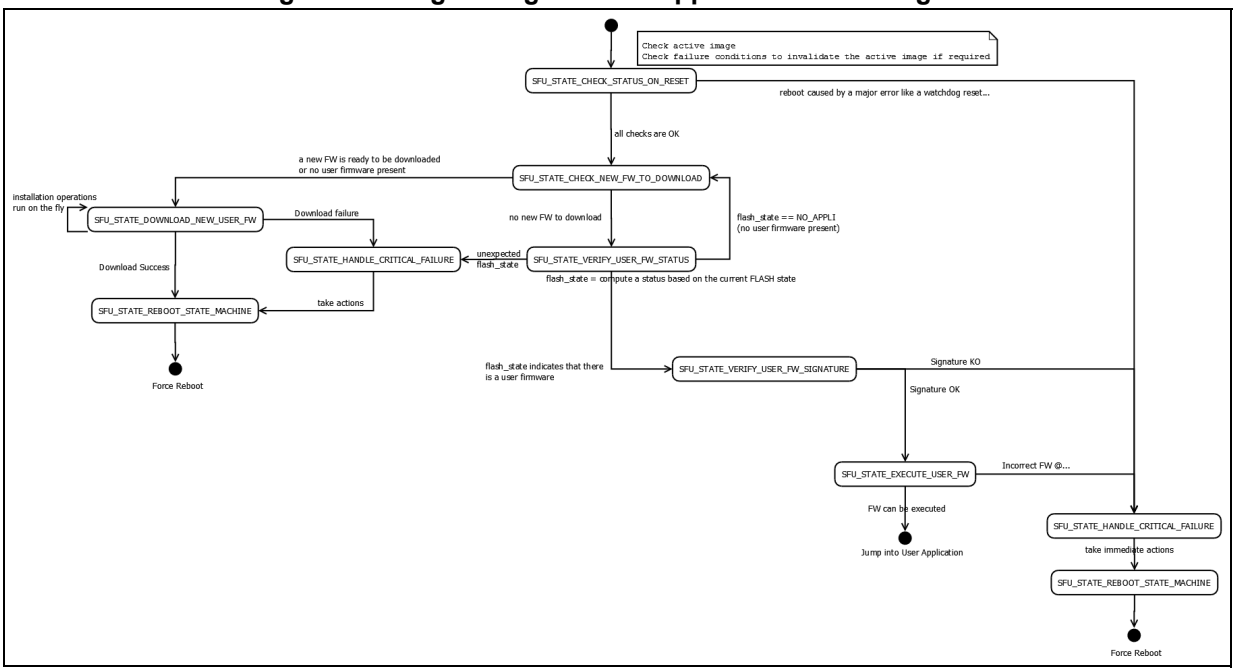
Figure 30. Dual-image SBSFU application state diagram



F.2 Single-image SBSFU

The corresponding state diagram is illustrated in *Figure 31*.

Figure 31. Single-image SBSFU application state diagram



F.3 SBSFU FSM states

The FSM states are:

- SFU_STATE_CHECK_STATUS_ON_RESET: at this stage SBSFU checks the reset cause and decide how to proceed
  - Either checking the local firmware download availability
  - Or handling the reset cause as a critical failure case
- SFU_STATE_CHECK_NEW_FW_TO_DOWNLOAD: at this stage SBSFU checks if a local download must be handled
  - SBSFU checks if the User button is pressed or not

    If so then the firmware header is downloaded and verified

    If this header is fine then the download is triggered
  - If the button is not pressed or if an issue is encountered with the header then SBSFU switches to the SFU_STATE_VERIFY_USER_FW_STATUS state
- SFU_STATE_DOWNLOAD_NEW_USER_FW: at this stage SBSFU downloads the encrypted firmware to be installed over UART and stores it in internal Flash (slot #1)
  - If this download process goes fine then a reboot is triggered
  - If an issue is encountered then a critical failure is handled
- SFU_STATE_VERIFY_USER_FW_STATUS: at this stage SBSFU checks the internal Flash state to derive a status
  - If a firmware installation has been interrupted: a recovery procedure is triggered
  - If a firmware is installed and no new firmware is ready for installation then the currently installed firmware (installed in slot #0) is verified before execution
  - If a new firmware is ready for installation then the installation procedure is triggered
- SFU_STATE_INSTALL_NEW_USER_FW: at this stage SBSFU decrypts and installs the firmware stored in slot #1
  - If everything goes fine the content of slot #0 and slot #1 is swapped and the firmware verification stage is entered
  - If a problem occurs then a critical failure is handled
- SFU_STATE_VERIFY_USER_FW_SIGNATURE: this is the last stage before running the user application. SBSFU checks the validity of the active firmware (installed in slot #0)
  - If the firmware is valid then it is launched
  - If the firmware is not valid a critical failure is handled
- SFU_STATE_EXECUTE_USER_FW: at this stage SBSFU prepares the context switch to launch the user application
  - If everything goes fine then the user application starts
  - If an issue is encountered then a critical failure is handled
- SFU_STATE_ROLLBACK_PREV_USER_FW: at this stage SBSFU re-installs the firmware backed up if any
  - If the recovery procedure succeeds then the firmware is verified before being launched
  - If a problem occurs a critical failure is handled
- SFU_STATE_HANDLE_CRITICAL_FAILURE: this is a placeholder to deal with all the

critical failures that are encountered

- SFU_STATE_REBOOT_STATE_MACHINE: a reboot is forced

Revision history

Table 8. Document revision history

Date	Revision	Changes
7-Dec-2017	1	Initial release.
20-Dec-2017	2	Removed references to the integration guide in *Chapter 7: Understanding the last execution status message at boot-up* and *B.2 Mapping definition*. Updated *Table 4: Error messages at boot-up* and *Section 5.2.2: Software tools for programming STM32 microcontrollers*.
20-Apr-2018	3	Document scope extended to asymmetric and symmetric cryptography schemes. Added single-image mode. Extended support of the STM32L4 Series: – Updated all chapters. – Updated *Appendix A Secure Engine protected environment* and *Appendix B Dual-image handling* – Added *Appendix C Single-image handling*, *Appendix D Cryptographic schemes handling*, and *Appendix E Firmware image preparation tool*. – Removed the MSC appendix.