Getting started with STM32CubeU5 TFM application

Introduction

This document describes how to get started with the STM32CubeU5 TFM (trusted firmware for Arm® Cortex®-M) application, delivered as part of the STM32CubeU5 MCU Package.

The STM32CubeU5 TFM application provides a root of trust solution, including Secure Boot and Secure Firmware Update functionalities. This solution is used before executing the application. It provides a set of secure services that are isolated from the nonsecure application, but can be used by the nonsecure application at runtime. The STM32CubeU5 TFM application is based on the open-source TF-M reference implementation, ported onto STM32U5 series microcontrollers (referred to as STM32U5 in this document). This brings the benefit of STM32U5 hardware security features such as:

- Arm® Cortex®-M33 TrustZone® and memory protection unit (MPU)
- TrustZone®-aware peripherals
- Memory protections (HDP, WRP)
- Enhanced life-cycle scheme (RDP)

Additionally, security can be augmented with the addition of a secure element, the STSAFE-A110 microcontroller (referred to as STSAFE in this document).

The secure services are implemented as upgradeable code that provides a set of services available at runtime for the nonsecure application. It also manages critical assets isolated from the nonsecure application. The nonsecure application cannot directly access any of the critical assets, but can call secure services that use the critical assets:

- Secure Boot (root of trust services) is a piece of immutable code that is always executed after a system reset. It checks the STM32U5 static protections, activates STM32U5 runtime protections, and then verifies the authenticity and integrity of the installed firmware before every execution. This ensures that invalid or malicious code cannot be run.
- The Secure Firmware Update application is a piece of immutable code. It detects that a new firmware image is available, then checks its authenticity, and the integrity of the code before installing it. The firmware update can be done on the single firmware image, including both secure and nonsecure parts of the firmware image. Alternatively, it can be done on the secure part of the firmware image, on the nonsecure part of the firmware image, or on both independently. The firmware update can also be done either in overwrite mode or in swap mode. Firmware can be received clear or encrypted.

The secure services are upgradeable code implementing a set of services managing critical assets that are isolated from the nonsecure application. This means that the nonsecure application cannot directly access any of the critical assets, but can only use secure services that use the critical assets:

- Crypto: secure cryptographic services, based on opaque key APIs
- Protected storage: protects data confidentiality/authenticity/integrity
- Internal trusted storage: protects data confidentiality/authenticity/integrity in internal flash memory (the most secure storage space for microcontrollers)
- Attestation: proves product identity via an entity attestation token

The TFM application presented in this document is a complete implementation of [TF-M]. A second application implementing only the Secure Boot and Secure Firmware Update functionalities of [TF-M], named STM32CubeU5 SBSFU, is also available in the STM32CubeU5 MCU Package. For further information on the SBSFU application, refer to [AN5447].

The first sections of this document (sections 4 to 6) present the open-source TF-M part (v1.3.0). The last sections of this document (sections 7 to 12) present TF-M ported onto the STM32U5 microcontroller and integrated in the STM32CubeU5 MCU Package. STM32CubeU5 TFM application and SBSFU application examples are provided for the B-U585I-IOT02A board.

Refer to [TF-M] for more information about the open-source TF-M reference implementation.

1 General information

The STM32CubeU5 TFM application runs on STM32U5 series 32-bit microcontrollers based on the Arm® Cortex®-M33 processor with Arm® TrustZone®.

Note: Arm and TrustZone are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

arm

1.1 Applicable products and default examples

This document applies to the STM32U5 series microcontrollers. The demonstration hardware setups use the following development boards:

- NUCLEO-U545RE-Q Nucleo-64 board for the devices with 512 Kbytes of flash memory
- B-U585I-IOT02A Discovery kit for the devices with 2 Mbytes of flash memory
- STM32U5A9J-DK and STM32U5G9J-DK2 Discovery kits for the devices with 4 Mbytes of flash memory

By default, the project examples provided are illustrated with the B-U585I-IOT02A Discovery kit.

The project examples are compatible with the following integrated development environments:

- IAR Systems® IAR Embedded Workbench® for Arm® (EWARM)
- Keil® Microcontroller Development Kit for Arm®-based microcontrollers (MDK-ARM)
- STMicroelectronics STM32CubeIDE

Note: IAR Systems is a registered trademark owned by IAR Systems AB.

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

1.2 Acronyms

Table 1 presents the definition of acronyms that are relevant for a better understanding of this document.

Table 1. List of acronyms

Acronym	Description
AEAD	Authenticated encryption with associated data.
AES	Advanced encryption standard.
BL2	Bootloader 2. Name of boot stage in TF-M terminology, based on the MCUboot open-source software. Included in the TFM_SBSFU_Boot project.
CLI	Command-line interface.
CTR	Counter mode, a cryptographic mode of operation for block ciphers.
DHUK	Derived hardware unique key.
DPA	Differential power analysis.
EAT	Entity attestation token.
ECDSA	Elliptic-curve digital signature algorithm. Asymmetric cryptography.
ECIES	Elliptic curve integrated encryption scheme.
FIH	Fault injection hardening
FWU	Firmware update service. Firmware update service provided by TF-M.
GUI	Graphical user interface.
HDP	Secure hide protection.
HUK	Hardware unique key.
IAT	Initial attestation.
IPC	Inter process communication.
ITS	Internal trusted storage service. Internal trusted storage service provided by TF-M.

Acronym	Description
NSPE	Nonsecure processing environment (PSA term). In TF-M, this means nonsecure domain typically running an operating system using services provided by TF-M.
MPU	Memory protection unit.
OAEP	Optimal asymmetric encryption padding is a padding scheme often used together with RSA encryption.
PS	Protected storage service. Protected storage service provided by TF-M.
PSA	Platform Security Architecture. Framework for securing devices.
RDP	Readout protection.
RNG	Random number generator.
RoT	Root of trust.
RSA	Asymmetric cryptographic system from Rivest–Shamir–Adleman
SBSFU	Secure Boot and Secure Firmware Update. In the STM32CubeU5, this is the name of the TF-M based application, with Secure Boot and Secure Firmware Update functionalities only.
SE	Secure element (STSAFE in the context of this document).
SFN	Secure function. An entry function to a secure service. Multiple SFN per SS are permitted.
SP	Secure partition. A logical container for a single secure service.
SPE	Secure processing environment (PSA term). In TF-M this means the secure domain protected by TF-M.
SPM	Secure partition manager. The TF-M component responsible for enumeration, management, and isolation of multiple secure partitions within the TEE.
SS	Secure service. A component within the TEE that is atomic from a security/trust point of view, which is viewed as a single entity from a TF-M point of view.
TBSA-M	Trusted base system architecture for Arm® Cortex®-M.
TEE	Trusted execution environment.
TFM	In the STM32CubeU5, this is the name of the TF-M-based application with complete functionalities.
TF-M	Trusted firmware for M-class Arm. TF-M provides a reference implementation of secure world software for Armv8-M.
TRNG	True random number generator.
WRP	Write protection.

2 Documents and open-source software resources

The resources below are public and available either on the STMicroelectronics website at *www.st.com* or on third-party websites.

Table 2. Document references

Reference	Document
[RM0456]	*STM32U5 Series Arm®-based 32-bit MCUs* - Reference manual(1)
[UM2237]	*STM32CubeProgrammer software description* - User manual(1)
[UM2553]	*STM32CubeIDE quick start guide* - User manual(1)
[UM2609]	*STM32CubeIDE user guide* - User manual(1)
[AN4992]	*STM32 MCUs secure firmware install (SFI) overview* - Application note(1)
[AN5156]	*Introduction to STM32 microcontrollers security* - Application note(1)
[AN5347]	*Arm® TrustZone® features for STM32L5 and STM32U5 Series* - Application note(1)
[AN5435]	*STSAFE-A110 generic sample profile description - Application note*(1)
[AN5447]	*Overview of Secure Boot and Secure Firmware Update solution on Arm® TrustZone® STM32 microcontrollers* - Application note(1)
[PSA_API]	PSA developer APIs - developer.arm.com/architectures/architecture-security-features/platform-security#implement(2)
[RFC7049]	Concise binary object representation (CBOR) - tools.ietf.org/html/rfc7049(2)
[RFC8152]	CBOR object signing and encryption (COSE) - tools.ietf.org/html/rfc8152(2)

1. *Available on www.st.com. Contact STMicroelectronics when more information is needed.*
2. *This URL belongs to a third party. It is active at document publication. However, STMicroelectronics shall not be liable for any change, move, or inactivation of the URL or the referenced material.*

Table 3. Open-source software resources

Reference	Open-source software resource
[TF-M]	TF-M (*Trusted firmware-M*) Arm Limited driven open-source software framework: www.trustedfirmware.org/(1)
[MCUboot]	MCUboot open-source software: mcuboot.com/(1)
[mbed-crypto]	mbed-crypto open-source software: github.com/ARMmbed/mbedtls(1)
[PSA]	PSA certification website: www.psacertified.org(1)

1. *This URL belongs to a third party. It is active at document publication. However, STMicroelectronics shall not be liable for any change, move, or inactivation of the URL or the referenced material.*

3 STM32Cube overview

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

STM32Cube includes:

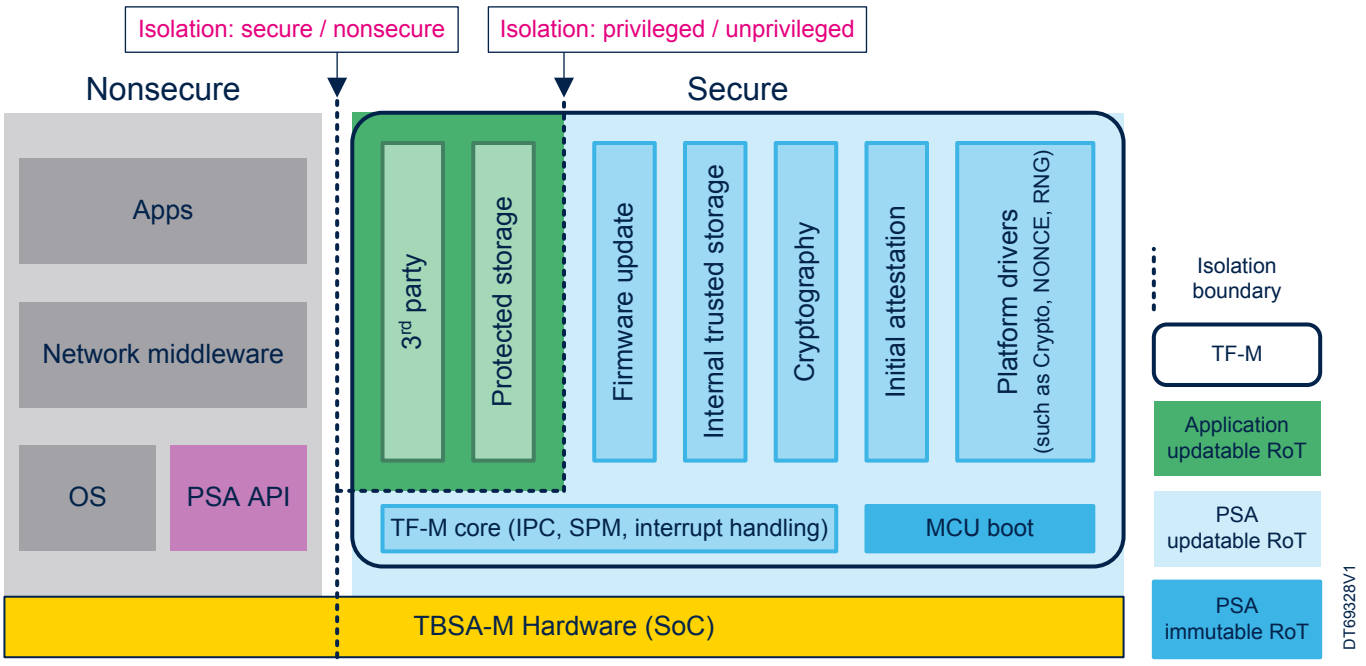
- A set of user-friendly software development tools to cover project development from conception to realization, among which are:
  - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards
  - STM32CubeIDE, an all-in-one development tool with peripheral configuration, code generation, code compilation, and debug features
  - STM32CubeCLT, an all-in-one command-line development toolset with code compilation, board programming, 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 STM32CubeU5 for the STM32U5 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 ThreadX, FileX / LevelX, NetX Duo, USBX, USB-PD, touch library, network library, mbed-crypto, TFM, and OpenBL
  - 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

4 Arm® Trusted Firmware-M (TF-M) introduction

TF-M (refer to [TF-M]) is an Arm Limited driven open-source software framework providing a reference implementation of the PSA standard on the Arm® Cortex®-M33 (TrustZone®) processor:

- PSA immutable RoT (root of trust): immutable *"Secure Boot and Secure Firmware Update"* application executed after any reset. This application is based on the MCUboot open-source software (refer to [MCUboot]).
- PSA updatable RoT: *"secure"* application implementing a set of secure services isolated in the secure/privileged environment that can be called by the nonsecure application at a nonsecure application runtime via the PSA APIs (refer to [mbed-crypto]):
  - Firmware update service: TF-M firmware update (FWU) service implements PSA firmware update APIs that allow an application to install a new firmware.
  - Internal trusted storage service: TF-M internal trusted storage (ITS) service implements PSA internal trusted storage APIs allowing the writing of data in a microcontroller built-in flash memory region that is isolated from nonsecure or from unprivileged applications by means of the hardware security protection mechanisms.
  - Cryptography service: the TF-M cryptography service implements the PSA Crypto APIs that allow an application to use cryptography primitives such as symmetric and asymmetric ciphers, hash, message authentication codes (MACs), authenticated encryption with associated data (AEAD), randomization, and key derivation. It comes with a PSA cryptography driver interface to make use of dedicated hardware. It is based on the Mbed Crypto open-source software (refer to [mbed-crypto]).
  - Initial attestation service: the TF-M initial attestation service allows the application to prove the device identity during an authentication process to a verification entity. The initial attestation service can create a token on request, which contains a fix set of device-specific data.
- Application updatable RoT: secure services that are isolated in the secure/unprivileged environment and that can be called by the nonsecure application at a nonsecure application runtime.
  - Protected storage service: The TF-M protected storage (PS) service implements PSA protected storage APIs allowing data encryption and writing the result in a possibly untrusted storage. The PS service implements an AES-GCM-based AEAD encryption policy, as a reference, to protect data integrity and authenticity.
  - Third-party: RoT applications that implement additional product-specific secure services.

Figure 1. TF-M overview



5 Secure Boot and Secure Firmware Update services (PSA immutable RoT)

5.1 Product security introduction

A device deployed in the field operates in an untrusted environment, and 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. However, this is prone to attack if not executed in a secure way.

The consequences may be damaging, for example firmware cloning, malicious software download, or device corruption. Security solutions must therefore be designed in order to protect sensitive data (potentially even the firmware itself), and critical operations.

Typical countermeasures are based on cryptography (with associated key) and on memory protection mechanisms:

- Cryptography ensures integrity (the assurance that data has not been corrupted), authentication (the assurance that a certain entity is what 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 nonsecure processes.

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

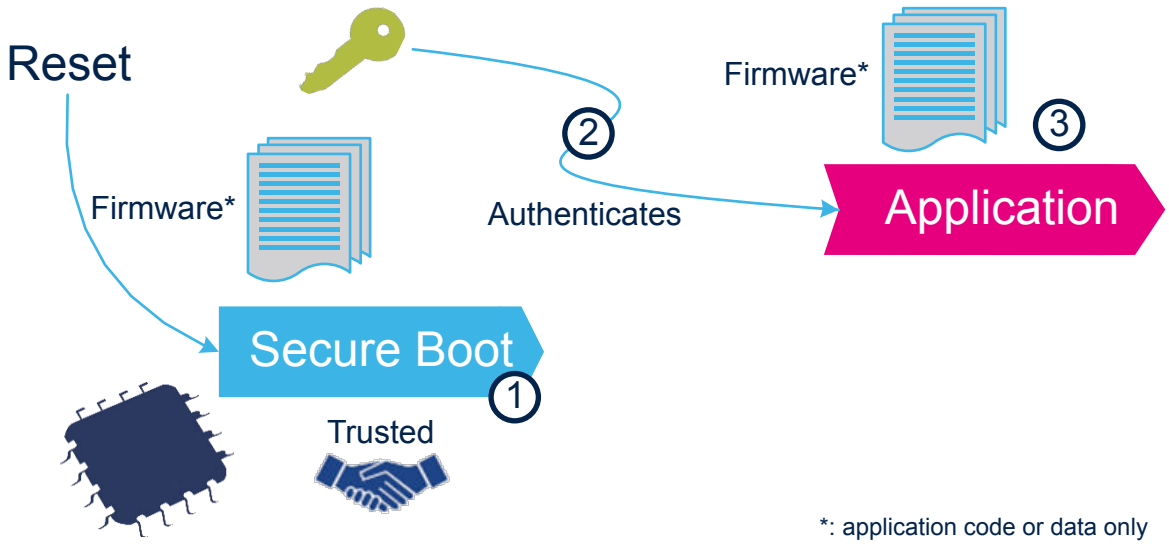
5.2 Secure Boot

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

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 2. Secure Boot root of trust



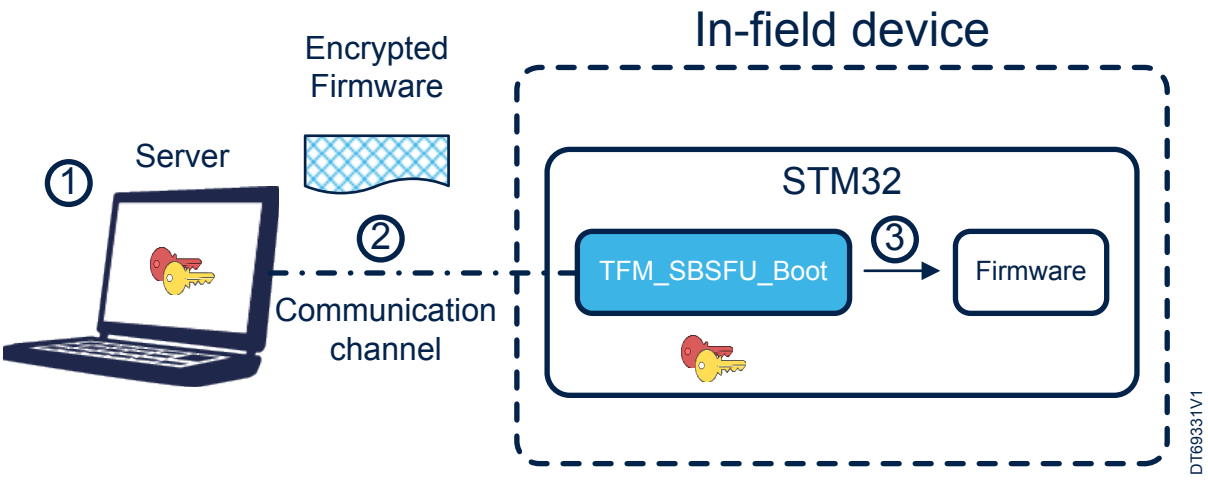
5.3 Secure Firmware Update

Secure Firmware Update 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 3, two entities are typically involved in a firmware update process:

- Server
  - Can be an OEM manufacturer server or 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 code running the 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 3. Typical in-field device update scenario



The 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 is done on the complete firmware image.

Firmware update is vulnerable to the threats presented in Section 5.1 Product security introduction: cryptography is used to ensure confidentiality, integrity, and authentication.

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.

Integrity is verified 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.

5.4 Cryptography operations

The TFM_SBSFU_Boot application example is delivered with configurable cryptographic schemes (solution for firmware authentication and firmware encryption):

- RSA-2048 asymmetric cryptography for image authenticity verification, AES-CTR-128 symmetric cryptography with key RSA-OAEP encrypted for image confidentiality, and SHA256 cryptography for image integrity check.
- RSA-3072 asymmetric cryptography for image authenticity verification, AES-CTR-128 symmetric cryptography with key RSA-OAEP encrypted for image confidentiality, and SHA256 cryptography for image integrity check.
- ECDSA-256 asymmetric cryptography for image authenticity verification, AES-CTR-128 symmetric cryptography with key ECIES-P256 encrypted for image confidentiality, and SHA256 cryptography for image integrity check.

For more information on the cryptographic scheme, refer to the [MCUboot] open-source website.

6 Secure services at runtime

The secure services at runtime are a set of services that can be called at a nonsecure application runtime. They manage critical assets that are isolated from the nonsecure application. A nonsecure application cannot access directly to any of the critical assets but can only use the secure services that use the critical assets. The secure services are provided with two levels of isolation through the privileged/unprivileged mode usage (the processor can limit or exclude access to some resources by executing code in the privileged or unprivileged mode):

- Privileged secure services: secure services executed in privileged mode. Such type of services can access any assets in the system (secure or nonsecure, privileged or unprivileged). These services are in PSA updatable RoT partition: firmware update service, internal trusted storage service, secure cryptographic service, and initial attestation service.
- Unprivileged secure services: secure services executed in unprivileged mode. Such type of services can access any assets in the system except the assets stored in a privileged area. These services are in application updatable RoT partition: protected storage and third-party service.

6.1 Protected storage service (PS)

The TF-M protected storage (PS) service implements PSA protected storage APIs (refer to [PSA_API] for more information).

The service is backed by hardware isolation of the flash memory access domain. In the current version, it relies on hardware to isolate the flash memory area from nonsecure accesses.

The current PS service design relies on the hardware abstraction level provided by TF-M. The PS service provides a nonhierarchical storage model, as a filesystem, where a linearly-indexed list of metadata manages all the assets.

The PS service implements an AES-GCM based AEAD encryption policy, as a reference, to protect data confidentiality, integrity, and authenticity.

Additionally, it implements nonvolatile counters as a rollback protection mechanism against malicious attacks.

The design addresses the following high-level requirements as well:

- Confidentiality: Resistance to unauthorized accesses through hardware/software attacks.
- Access authentication: Mechanism to establish the requester's identity (a nonsecure entity, a secure entity, or a remote server).
- Integrity: Resistance to tampering by either the normal users of a product, package, or system or others with physical access to it. If the content of the secure storage is changed maliciously, the service is able to detect it.
- Reliability: Resistance to power failure scenarios and incomplete write cycles.
- Configurability: High-level configurability to scale the memory footprint up or down to cater for a variety of devices with varying security requirements.
- Performance: Optimized to be used for resource-constrained devices with very small silicon footprint, the PPA (power, performance, area) should be optimal.
- Modularity: The PS partition is placed in an unprivileged; The filesystem is in a privileged area. This implies dependencies with other services: cryptography, internal trusted storage API, and platform service.

For more information about the hardware isolation mechanism, refer to Section 7 Protection measures and security strategy.

6.2 Internal trusted storage service (ITS)

The TF-M internal trusted storage (ITS) service implements PSA internal trusted storage APIs (for more information, refer to [PSA_API]).

The service is backed by hardware isolation of the flash memory access domain and relies on hardware to isolate the flash memory area from nonsecure access and application updatable RoT at higher levels of isolation.

Contrary to the PS service, the ITS service does not implement any encryption policy. The confidentiality of data is ensured by means of the hardware isolation of the internal flash memory access domain.

The current ITS service design relies on a hardware abstraction provided by TF-M. The ITS service provides a nonhierarchical storage model, as a filesystem, where a linearly-indexed list of metadata manages all the assets.

The design addresses the following high-level requirements as well:

- Confidentiality: Resistance to unauthorized accesses through hardware/software attacks, by means of the hardware isolation of the flash memory access domain
- Access authentication: Mechanism to establish the requester's identity (a nonsecure entity, a secure entity, or a remote server).
- Integrity: Resistance to tampering by attackers with physical access is provided by the internal flash memory device itself. Resistance to tampering by nonsecure or application updatable RoT attackers is provided by a hardware isolation mechanism.
- Reliability: Resistance to power failure scenarios and incomplete write cycles.
- Configurability: High level of configurability to scale the memory footprint up or down to cater for a variety of devices with varying requirements.

For more information about the hardware isolation mechanism, refer to Section 7 Protection measures and security strategy.

6.3 Secure cryptographic service

The TF-M secure cryptographic service provides an implementation of the PSA Crypto API in a PSA updatable RoT secure partition in TF-M. It is based on mbed-crypto, which is a reference implementation of the PSA Crypto API.

The service can rely on alternate implementations or on dedicated cryptographic drivers that may target a secure element. The storage of cryptographic data (such as persistent keys) is needed so that the TF-M secure cryptographic service has dependencies with the internal trusted storage API.

For more details on the PSA Crypto API or the mbed-crypto implementation, refer directly to the [mbed-crypto] GitHub repository.

The service can be used by other services running in the secure processing environment (SPE), or by applications running in the nonsecure processing environment (NSPE), to provide cryptographic functionalities.

6.4 Initial attestation service

The TF-M initial attestation service allows the application to prove the device identity during an authentication process to a verification entity. The initial attestation service can create an entity attestation token (EAT) on request, which contains a fix set of device-specific data. The device must contain an attestation key pair, which is unique per device. The token is signed with the private part of the attestation key pair. The public part of the key pair is known by the verification entity. The public key is used to verify the token authenticity. The data items in the token are used to verify the device integrity and assess its trustworthiness. Attestation key provisioning is out of scope for the initial attestation service and is expected to take part during product manufacturing.

Attestation keys can be installed within the MCU or within a secure element, which is, in that case, driven from the TF-M secure cryptographic service. Therefore, the TF-M initial attestation service may have dependencies with the PSA Crypto API. For more details on the various options in the implementation to provision the device with the attestation keys, refer to Section 12 Integrator role description.

6.5 Firmware update service

The TF-M firmware update (FWU) service implements PSA firmware update APIs (refer to [PSA_API] for more information). It provides a standard and platform-agnostic interface for updating firmware. It cooperates with the bootloader and has dependencies with the PSA Crypto API and platform services.

7 Protection measures and security strategy

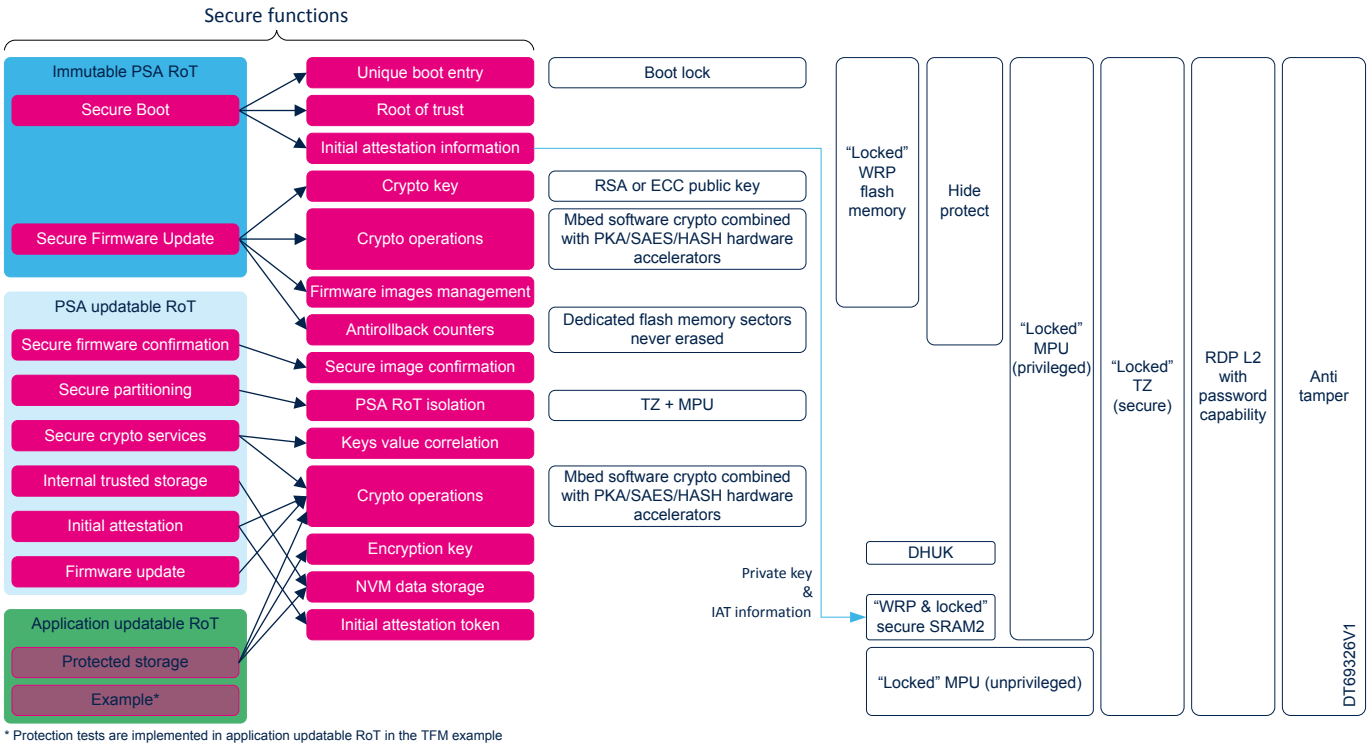
Cryptography ensures integrity, authentication, and confidentiality. However, the use of cryptography alone is not enough. To resist possible attacks, a set of measures and system-level strategy are needed for protecting critical operations, sensitive data (such as a secret key), and the execution flow.

The STM32CubeU5 TFM example uses a security strategy based on the following concepts:

- Ensure single-entry point at reset: force code execution to start with Secure Boot code
- Make TFM_SBSFU_Boot code and TFM_SBSFU_Boot “secrets” immutable: no possibility to modify or alter them once security is fully activated
- Create protected/isolated domains:
  - Secure/privileged: to execute PSA immutable RoT code and then PSA updatable RoT code. Both codes use associated secrets and secure privileged STM32U5 peripherals.
    Note that the immutable PSA RoT piece of code is hidden when its execution is completed.
  - Secure/unprivileged: to execute application updatable RoT using associated secrets, and secure unprivileged STM32U5 peripherals.
- Limit execution surface according to application state:
  - From product reset until the installed application is verified: only TFM_SBSFU_Boot code execution allowed
  - Once the installed application is verified OK: application code (secure part and nonsecure part) execution allowed
- Remove JTAG access to the device.
- Use four build time FIH profile settings to strengthen the critical functions call path of TFM_SBSFU_Boot and TFM_Appli code against fault injection attacks. High profile uses a random delay based on the TRNG to mitigate sensitive code execution.

Figure 4 gives a high-level view of the security mechanisms activated on the STM32U5 series.

Figure 4. TFM application using STM32U5 security peripherals



* Protection tests are implemented in application updatable RoT in the TFM example

7.1 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 TFM_SBSFU_Boot application example, device life cycle (managed through RDP option bytes), boot lock, protected SRAM2 protections, and antitamper are used to protect the product against outer attacks:

- **Device life cycle**: Read protection level 2 achieves the highest protection level. Read protection level 2 with OEM2 password capability is used to ensure that a JTAG debugger cannot access the device, except to inject the OEM2 password. In RDP level 2, when the OEM2 password is injected on the JTAG port, the RDP level is regressed to level 1. The OEM2 password must first have been provisioned when the RDP level is 0.
- **Boot lock**: BOOT_LOCK option byte is used to fix the entry point to a memory location defined in the option byte. In the TFM application example, the boot entry point after reset is fixed on the TFM_SBSFU_Boot code.
- **Protected SRAM2**: SRAM2 is automatically protected against intrusion once the system is configured in RDP level 1. SRAM2 content is erased as soon as an intrusion is detected. Moreover, SRAM2 content can be write protected (content is frozen but can be read) until the next reset by activating the lock bit. In the TFM application example, the system has been configured to use the protected SRAM2 to share and to freeze initial attestation information between the TFM_SBSFU_Boot application and the secure application.
- **Antitamper**: the antitamper protection is used to protect sensitive data from physical attacks. It is activated at the start of TFM_SBSFU_Boot, and remains active during TFM_Appli and TFM_Loader applications. In case of tamper detection, sensitive data in SRAM2, caches, and cryptographic peripherals are immediately erased, and a reboot is forced. Both external active tamper pins and internal tamper events are used.

Other STM32U5 peripherals could be used to protect the product against outer attacks, but the current TFM example does not use them:

- **Debug**: the debug protection consists in deactivating the DAP (debug access port). Once deactivated, the JTAG pins are no longer connected to the STM32U5 internal bus. DAP is automatically disabled with RDP level 2.
- **Watchdog** IWDG (independent watchdog) is a free-running down-counter. Once running, it cannot be stopped. It must be refreshed periodically before it causes a reset. This mechanism could be used to control the TFM_SBSFU_Boot execution duration.

7.2 Protections against inner attacks

Inner attacks refer to attacks triggered by code running into the STM32. Attacks may be due to either malicious firmware that exploits bugs or security breaches, or unwanted operations. In the TFM application example, the **TZ** (TrustZone®), **MPU** (memory protection unit), **SAU** (security attribution unit), **GTZC** (global TrustZone® controller), **WRP** (write protect), and **HDP** (hide protection) protections preserve the product from inner attacks:

- **TZ**, secure **MPU**, and **GTZC** are combined to put in place different protected environments with different privileges and different access rights:
  - **TZ**: The Cortex®-M33 CPU core supports two modes of operation (secure and nonsecure). When the Cortex®-M33 is in nonsecure mode, it cannot access any STM32U5 resources configured in secure.
  - **MPU:** The MPU is a memory protection mechanism that allows specific access rights to be defined for any memory-mapped resources of the device: flash memory, SRAM, and peripheral registers. MPU attributes are only set for CPU access. Other bus controller requests (such as DMA once) are not filtered by the MPU. This protection is dynamically managed at runtime. Secure MPU is used to control CPU access in secure mode and nonsecure MPU is used to control CPU access in nonsecure mode.
  - **SAU**: The SAU is a hardware unit coupled to the core (as the MPU), responsible for setting the secure attribute of the AHB5 (advanced high-performance bus) transaction.
  - **GTZC**: provides mechanisms to configure any memories and peripherals to be secure or nonsecure and to be privileged or unprivileged.

TZ and GTZC configuration can start with static settings from **SECWM** (secure watermark) option bytes values. It can also be updated dynamically at runtime by the secure privileged applications. Secure privileged applications can lock the GTZC, the secure MPU configuration, and the secure SAU configuration until the next reset by activating the lock bits. Once TZ, MPU, SAU, and GTZC are configured, the applications can only use or access the memories and the peripherals corresponding to their execution mode, which depends on the Cortex®-M33 CPU core mode (secure or nonsecure and privileged or unprivileged).

In the TFM application example, the system has been defined to put in place different protected execution environments according to the product execution states:

- System state: execution of the TFM_SBSFU_Boot application (application executed after product reset)
  - Execution environment: secure privileged, to execute the immutable RoT (TFM_SBSFU_Boot code corresponding to the immutable PSA RoT part).

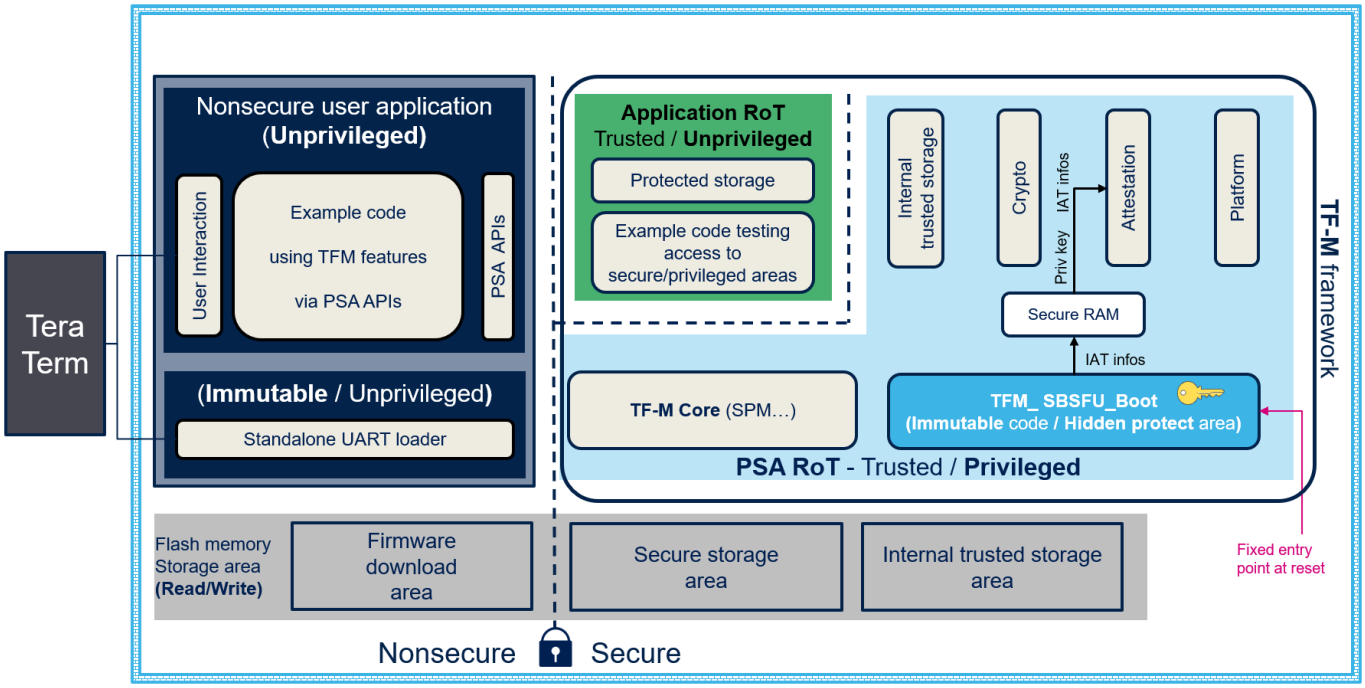
During the TFM_SBSFU_Boot code execution, only the flash memory area corresponding to the TFM_SBSFU_Boot code can be executed by the CPU in secure mode. The other memories areas (flash memory and SRAMs) are in read/write access rights only. Before launching the verified application, the TFM_SBSFU_Boot application reconfigures the system so that the execution surface is extended with the flash memory area corresponding to the verified application (both secure part and nonsecure part), the other memories areas (flash memory and SRAMs) are in read/write access rights only.

- System state: execution of the application, application executed (executing first the secure part of the application) once the Secure Boot has verified it is OK.
  - Execution environment: secure privileged, to execute the secure privileged part of the application (corresponding to the PSA updatable RoT part), and to store nonvolatile data related to PS and ITS secure services.
  - Execution environment: secure unprivileged, to execute the secure unprivileged part of the application (corresponding to the application updatable RoT part).
  - Execution environment: nonsecure unprivileged (to execute the nonsecure part of the application).

The secure privileged part of the application starts by reconfiguring the system to put in place the protected execution environments listed above that are used during application execution. The execution surface is extended for all the secure parts. Once the system reconfiguration is completed, GTZC, the secure MPU configuration, and the secure SAU configuration are locked until the next reset by activating lock bits. The nonsecure application execution is started in privileged mode and is able to reconfigure the nonsecure MPU and lock it if needed.

- **WRP**: 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. In the TFM example, the system has been configured to make the TFM_SBSFU_Boot code, the TFM_SBSFU_Boot personalized data, and the TFM_Loader code as immutable data. Additionally, the write protection is locked, so that it cannot be removed unless RDP regression to level 0 is performed.
- **HDP**: when the HDP secure hide protection is activated, any accesses to the protected flash memory area (fetch, read, programming, erase) are rejected until the next product reset. All the code and secrets located inside the protected flash memory area are fully hidden. In the TFM example, the system has been configured to hide the TFM_SBSFU_Boot code, the TFM_SBSFU_Boot personalized data located in flash memory, and the TFM_SBSFU_Boot nonvolatile counters area located in flash memory just before the TFM_SBSFU_Boot application launches the verified application.
- **Secure backup register:** Secure backup registers can only be accessed by a secure privileged application. In the TFM example, the secure backup registers are not used.
- **Interruptions and exceptions:**
  - During TFM_SBSFU_Boot execution, the only enabled interruption is the tamper interruption, to trigger a reset in case of a tamper event.
    Regarding exceptions, the NMI manages the flash memory double ECC errors (by skipping any instructions reading a corrupted flash memory address), whereas all other exceptions trigger a reset.
  - During TFM_Appli execution, the GTZC interruption is enabled to prevent nonsecure accesses to a secure area, including through DMA usage.
  - **Secure vector table lock bit:** The secure vector table address can be locked until the next reset by activation lock bit. In the TFM example, the secure application locks the secure vector table during the initialization phase. The nonsecure application is able to lock the nonsecure vector table if needed.

Figure 5. System protection overview



Refer to Memory protections for more details on memory protections implementation.

8 Package description

The STM32CubeU5 MCU Package proposes two different examples of applications, based on the TF-M reference implementation.

- TFM: application with full TF-M services.
- SBSFU: application with only the Secure Boot and Secure Firmware Update services of the TF-M.

This document focuses on the TFM application only. Refer to [AN5447] for more information on the SBSFU application.

This section details the TFM application in the STM32CubeU5 MCU Package and the way to use it.

8.1 TFM application description

The main features of the Secure Boot and Secure Firmware Update application are:

- Configurable asymmetric cryptography for image authentication:
  - RSA-2048
  - RSA-3072
  - EC-256
- SHA256 cryptography for image integrity check.
- Retention of a hash reference porting on each image (boot time acceleration).
  Image verification consists mainly of computing a hash over the image (integrity check) and then generating a signature over this hash (authentication check). With this feature, it is possible to avoid the computation of a signature by reference of its hash that is stored in a fixed location (so-called HASH REF). This area contains the hash of which the signature has already passed the verification process so that the next signature verification can be bypassed. This feature is an optimization (under the `MCUBOOT_USE_HASH_REF` define) that is applied on each image and is efficient from the second boot.
- AES-CTR cryptography for image encryption, with symmetric key encrypted in RSA-OAEP or ECIES-P256 provided in the image itself. Image encryption is configurable (for example, it can be deactivated).
- Two cryptography modes: Full software cryptography or a mix of software and hardware-accelerated cryptography to accelerate operations and reduce the memory footprint (with or without DPA resistance against side-channel and timing attacks).
- Configurable slots mode:
  - Single primary slot mode, which enables maximizing image size. The downloaded image is in the same memory slot as the installed image. The new downloaded image overwrites the previous installed image. The devices with 512 Kbytes of flash memory only support this configuration.
  - Primary and secondary slots mode, which enables safe image programming. The downloaded image and installed image are in different memory slots.
- Image programming resistant to asynchronous power down and reset.
- Flexible number of application images:
  - Either one application image (secure and nonsecure binaries combined in a single image) with:
    - Unique key pair
    - Antirollback version check
  - Or two application images (a secure image and a nonsecure image) with:
    - Dedicated key pairs per firmware image
    - Dedicated antirollback version check per firmware image
    - Images version dependency management
- Flexible number of data images: one data image (secure or nonsecure) or two images (secure and nonsecure) with the policies defined on application images (authenticity and integrity verification, antirollback version check, decryption).
- Integration of the full entropy TRNG source (RNG hardware peripheral) for random numbers generation (boot seed generation, tamper protection) or random delays (FIH).

- Configurable firmware image upgrade strategy, for primary and secondary slots mode:
  - Overwrite strategy, for which the image in the secondary slot overwrites the image in the primary slot.
  - Swap strategy, for which the image in primary and secondary slots are swapped. After the swap, the new image in the primary slot must be confirmed by the user application, else, at the next boot, the images are swapped back.
- System flash memory configuration:
  - Internal flash memory: all firmware slots located in the internal flash memory (secure and nonsecure applications primary and secondary slots).
- Integration of hardware security peripherals and mechanisms in order to implement a root of trust. RDP, BOOT_LOCK, TZ, MPU, GTZC, SAU, WRP, SECWM, HDP, and TAMPER are combined to achieve the highest security level.
- IDE integrated image tool to prepare the image, provided both as a Windows® executable and a Python™ source code.
- Activation of ICACHE peripheral for internal flash memory access to improve boot time performance.

The main features of the secure services at runtime are:

- PSA level 2 isolation in secure side [PSA].
- Support of nonsecure interrupts in secure application (with priority levels control).
- Cryptography
  - Large set of cryptography primitives such as symmetric and asymmetric ciphers, hash, messages authentication codes (MACs) and authenticated encryption with associated data (AEAD), key random generation, and key derivation.
  - Configurable algorithms list support at compilation stage (AES-CBC, AES-CFB, AES-CTR, AES-OFB, AES-CCM, AES-GCM, RSA, ECDSA, ECDH, SHA1, SHA256, SHA512)
  - Two cryptography modes: Full software cryptography or a mix of software and hardware-accelerated cryptography to accelerate operations and reduce the memory footprint (with or without DPA resistance against side-channel and timing attacks).
  - Opaque key APIs management.
  - Entropy via the true random number generator (RNG hardware peripheral).
  - PSA driver interface with the secure element: STSAFE.
- Initial attestation
  - Entity token encoded with CBOR (concise binary object representation) .
  - Entity token signature (SHA256 and ECDSA) compliant with COSE (CBOR object signing and encryption) .
  - Entity token signed with either the STM32U5 microcontroller or the STSAFE secure element.
- Protected storage
  - AES-GCM-based AEAD encryption in secure flash memory region, using HUK derived from 256-bit nonvolatile device-unique secret in flash memory (for a mix of software and hardware cryptography), or provisioned HUK (for software cryptography).
  - Restricted access through opaque UID on 64 bits.
  - Resistant to asynchronous power down and reset.
- Internal trusted storage
  - Same as protected storage, with no encryption.

The STM32CubeU5 MCU Package includes sample applications that the developer can use to start experimenting with the code.

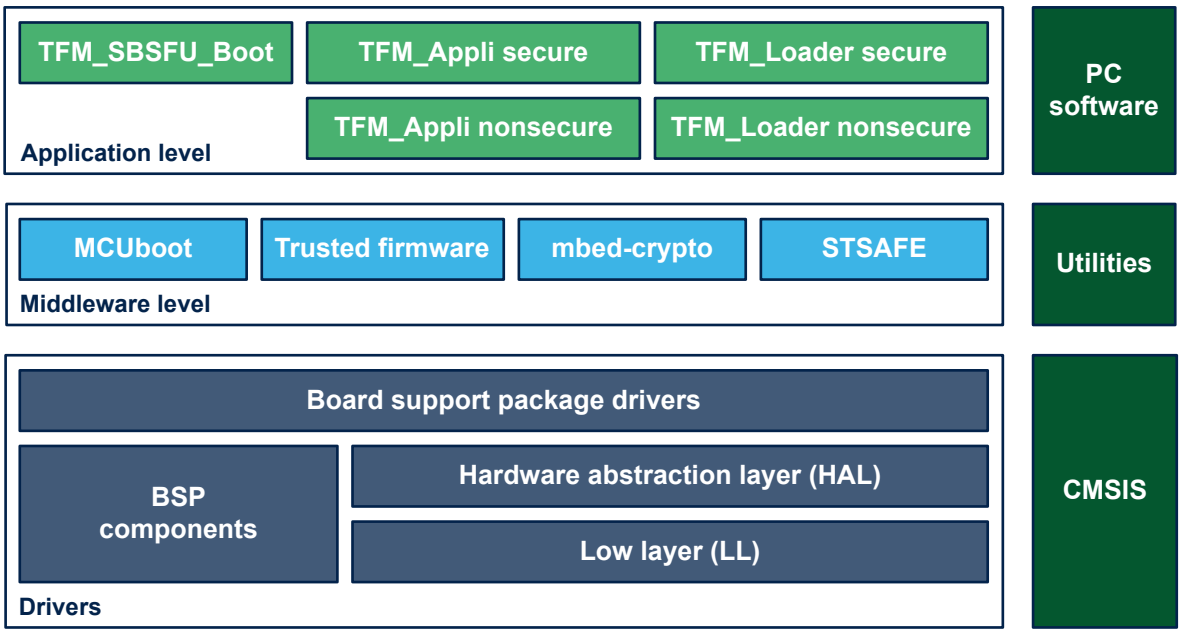
Table 4. Features configurability in TF-M-based examples in the STM32CubeU5 MCU Package

Feature	SBSFU_Boot (B-U585I-IOT02A)	TFM_SBSFU_Boot (B-U585I-IOT02A)
Crypto schemes	RSA-2048 RSA-3072 EC-256	RSA-2048 RSA-3072 EC-256

Feature	SBSFU_Boot (B-U585I-IOT02A)	TFM_SBSFU_Boot (B-U585I-IOT02A)
Image encryption	None AES-CTR	None AES-CTR
Cryptography modes	Software Mix of hardware and software	Software Mix of hardware and software
Slot modes	Primary only slot Primary and secondary slots	Primary only slot Primary and secondary slots
Images number modes	1 image 2 images	1 image 2 images
Flash memory configuration	Internal flash memory	Internal flash memory
Image upgrade strategy	Overwrite only Swap	Overwrite only Swap
Local loader	None Ymodem	None Ymodem
Antitamper	None Internal tampers only Internal and external tampers	None Internal tampers only Internal and external tampers

8.2 TFM application architecture description

Figure 6. **TFM application architecture**



8.2.1 Board support package (BSP)

This layer offers a set of APIs relative to the hardware components in the hardware boards (such as LCD, audio, microSD™, and MEMS drivers). It is composed of two parts:

- Component
  This is the driver relative to the external device on the board and not to the STM32. The component drivers provide specific APIs to the BSP driver external components and could be portable on any other board.

- BSP driver
  It allows linking the component driver to a specific board and provides a set of user-friendly APIs. The API naming rule is `BSP_FUNCT_Action()`.
  Example: `BSP_LED_Init()`, `BSP_LED_On()`

The BSP is based on a modular architecture that allows an easy porting on any hardware by just implementing the low-level routines.

8.2.2 Hardware abstraction layer (HAL) and low-layer (LL)

The STM32CubeU5 HAL and LL are complementary and cover a wide range of applications requirements:

- The HAL drivers offer high-level function-oriented highly portable APIs. They hide the MCU and peripheral complexity to the end user.
  The HAL drivers provide generic multi-instance feature-oriented APIs, which simplify the user application implementation by providing a ready-to-use process. As an example, for the communication peripherals (I2S, UART, and others), they provide APIs allowing initializing and configuring the peripheral, managing data transfer based on polling, interrupt or DMA process, and handling communication errors that may raise during communication.
  The HAL driver APIs are split into two categories:
  - Generic APIs, which provide common and generic functions to all the STM32 series of microcontrollers and microprocessors.
  - Extension APIs, which provide specific and customized functions for a specific product line or a specific part number.
- The low-layer APIs provide low-level APIs at register level, with better optimization but less portability. They require a deep knowledge of the MCU and peripheral specifications.
  The LL drivers are designed to offer a fast lightweight expert-oriented layer, which is closer to the hardware than the HAL. Contrary to the HAL, the LL APIs are provided only for the peripherals where optimized access is a key feature. They are not proposed either for the peripherals requiring heavy software configuration, complex upper-level stack, or both.
  The LL drivers feature:
  - A set of functions to initialize the peripheral main features according to the parameters specified in the data structures
  - A set of functions used to fill initialization data structures with the reset values corresponding to each field
  - Function for peripheral deinitialization (peripheral registers restored to their default values)
  - A set of inline functions for direct and atomic register access
  - Full independence from HAL and capability to be used in standalone mode (without HAL drivers)
  - Full coverage of the supported peripheral features

8.2.3 mbed-crypto library

The mbed-crypto library is an open-source middleware. It is a C library that implements cryptographic primitives. It supports symmetric and asymmetric cryptography as well as hash computation.

It includes a reference implementation of the PSA cryptography API.

It is used by MCUboot middleware during the "Secure Boot" operation or during the "Secure Firmware Update" operation. It is also used by the TFM middleware to implement cryptographic services.

8.2.4 MCUboot middleware

MCUboot is an open-source code. It is a secure bootloader for 32-bit microcontrollers. The goal of the MCUboot is to:

- Define a common infrastructure for the bootloader and the system flash memory layout on the microcontroller system.
- Provide a secure bootloader that enables easy software upgrade.

8.2.5 Trusted Firmware-M middleware (TF-M)

TF-M is an open-source middleware. It contains:

- The TF-M core services at runtime: inter-process communication (IPC), secure partition manager (SPM), and interrupt handling.

- The TF-M secure services at runtime: initial attestation, cryptography (relying on the mbed-crypto middleware for the cryptographic part), protected storage, and internal trusted storage.

8.2.6 STSAFE

STSAFE is a source code (middleware and high-level API) built on the STM32Cube ecosystem software. It is designed specifically for the STSAFE-Axxx secure elements, which provide authentication and data management services.

Each secure element comes with a default personalization profile (refer to [AN5435] for further details):

- a preprovisioned device certificate signed with ST root CA (self-signed CA certificate) in the STMicroelectronics factory
- a unique asymmetric key pair (private key and public key)
- a unique serial number per chip

This profile simplifies the life cycle of an IoT device:

- the registration services in the network infrastructures
- the device authentication in the field
- the secure connection with a remote server

The STSAFE code contains:

- A core module that integrates the minimum command set to drive the secure element.
- A service module and a cryptography module for establishing a communication channel (hardware setup and secure respectively) between the microcontroller and the secure element.
- Several configuration files about the sourcing of pairing keys and miscellaneous optimizations.
- A high-level API with extended and enablement procedures to manage keys and certificate.

8.2.7 TFM_SBSFU_Boot application

This application manages the TF-M Secure Boot and Secure Firmware Update services. It also manages the first level of security protections on the platform required during TFM_SBSFU_Boot application execution.

8.2.8 TFM_Appli secure application

This application manages the secure runtime services offered to the nonsecure application. It also finalizes the security protections required during the application execution.

8.2.9 TFM_Appli nonsecure application

This application is a sample code of a nonsecure user application. It demonstrates how to use the TF-Msecure services available in the TFM_Appli secure application.

8.2.10 TFM_Loader nonsecure application

This application is a sample code of a standalone local loader using the Ymodem protocol. This application permits the download of a new version of the secure firmware image (TFM_Appli secure application) and of the nonsecure firmware image (TFM_Appli nonsecure application). To store the downloaded image, this application directly accesses nonsecure flash memory areas, and relies on the TFM_Loader secure application to access secure flash memory areas indirectly.

8.2.11 TFM_Loader secure application

This application manages the secure flash memory access offered to the TFM_Loader nonsecure application.

8.3 Memory layout

8.3.1 Flash memory layout

The STM32CubeU5 TFM application relies on a flash memory layout defining different regions:

- HASF REF region: region where the SHA256 references are stored (one reference per image).
- BL2 NVCNT region: region where TFM_SBSFU_Boot gets nonvolatile information about last installed images (secure and nonsecure) versions for the antirollback feature.

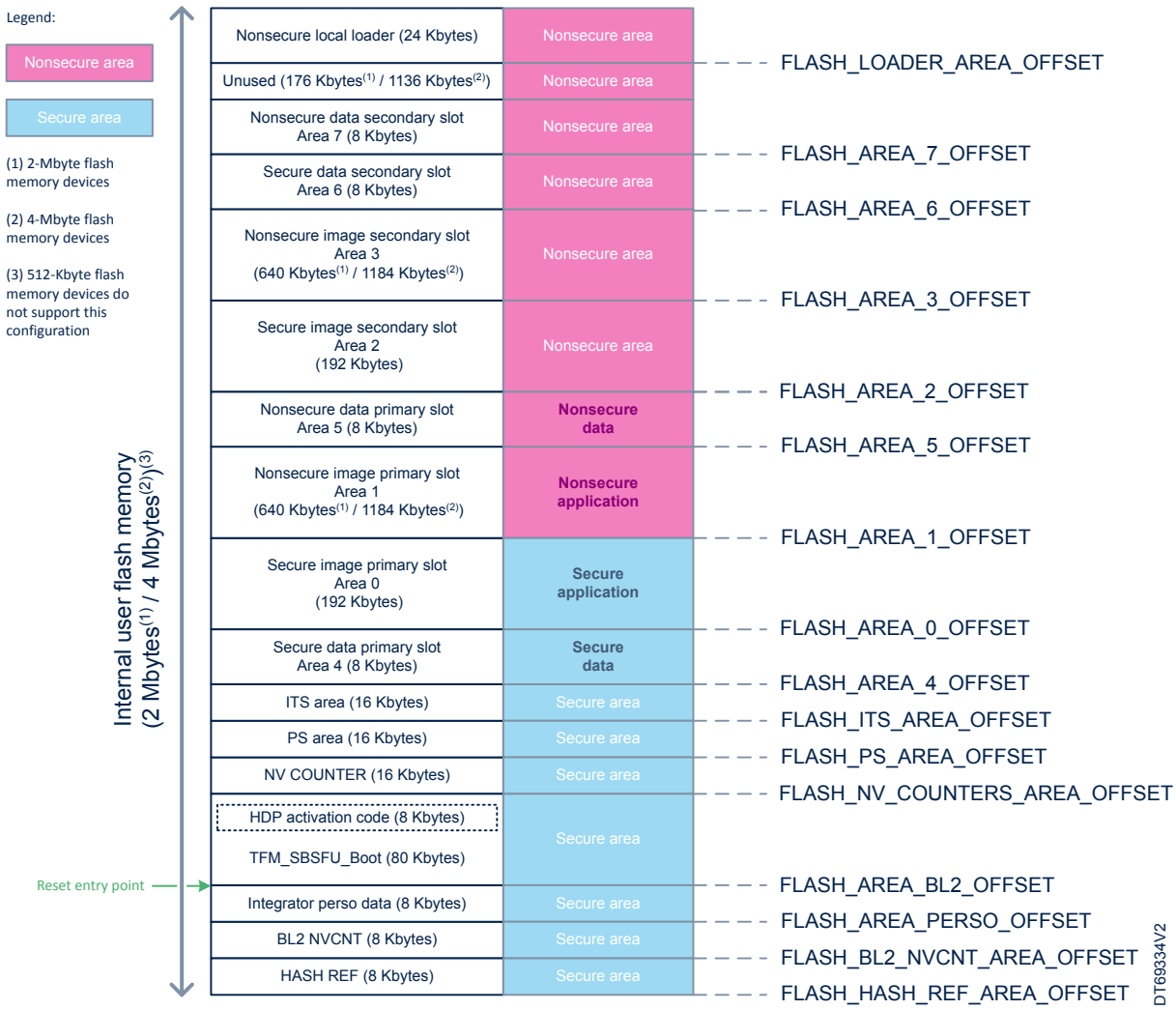
- SCRATCH region: region used by TFM_SBSFU_Boot to store the image data temporarily during the image swap process (not used in overwrite-only mode).
- Integrator personalized data region: region to personalize TF-M data specific to the integrator or specific to the STM32U5 microcontroller (the keys used by the SBSFU application, and the keys and information used by the TFM secure application).
- TFM_SBSFU_Boot binary region: region to program the TFM_SBSFU_Boot code binary that manages the function "Secure Boot" and the function "Secure Firmware Update.
- HDP activation code: code used to hide all boot code and secrets before starting the application.
- NV COUNTER region: region where the secure application manages the nonvolatile counters used by PS services.
- PS area: region where the encrypted data of the protected storage service is stored.
- ITS area region: region where the data of the internal trusted storage service is stored in clear.
- Secure image primary slot region: region to program secure image of “active” firmware.
- Nonsecure image primary slot region: region to program nonsecure image of “active” firmware.
- Secure image secondary slot region: region to program secure image of “new” firmware.
- Nonsecure image secondary slot region: region to program nonsecure image of “new” firmware.
- Secure data primary slot region: region to program the secure image of “active” data.
- Nonsecure data primary slot region: region to program the nonsecure image of “active” data.
- Secure data secondary slot region: region to program the secure image of “new” data.
- Nonsecure data secondary slot region: region to program the nonsecure image of “new” data.
- Nonsecure local loader: region to program TFM_Loader nonsecure code binary.
- Secure local loader: region to program TFM_Loader secure code binary.

The flash memory layout depends on the slot mode, number of images (application and data), image upgrade strategy, and local loader activation. By default, in the TFM application, the configuration (refer to Section 8.1) of these features is the following:

- Slot mode: primary and secondary slots
- Image number mode: four images (two applications image and two data images)
- Image upgrade strategy: overwrite only mode
- Local loader: Ymodem

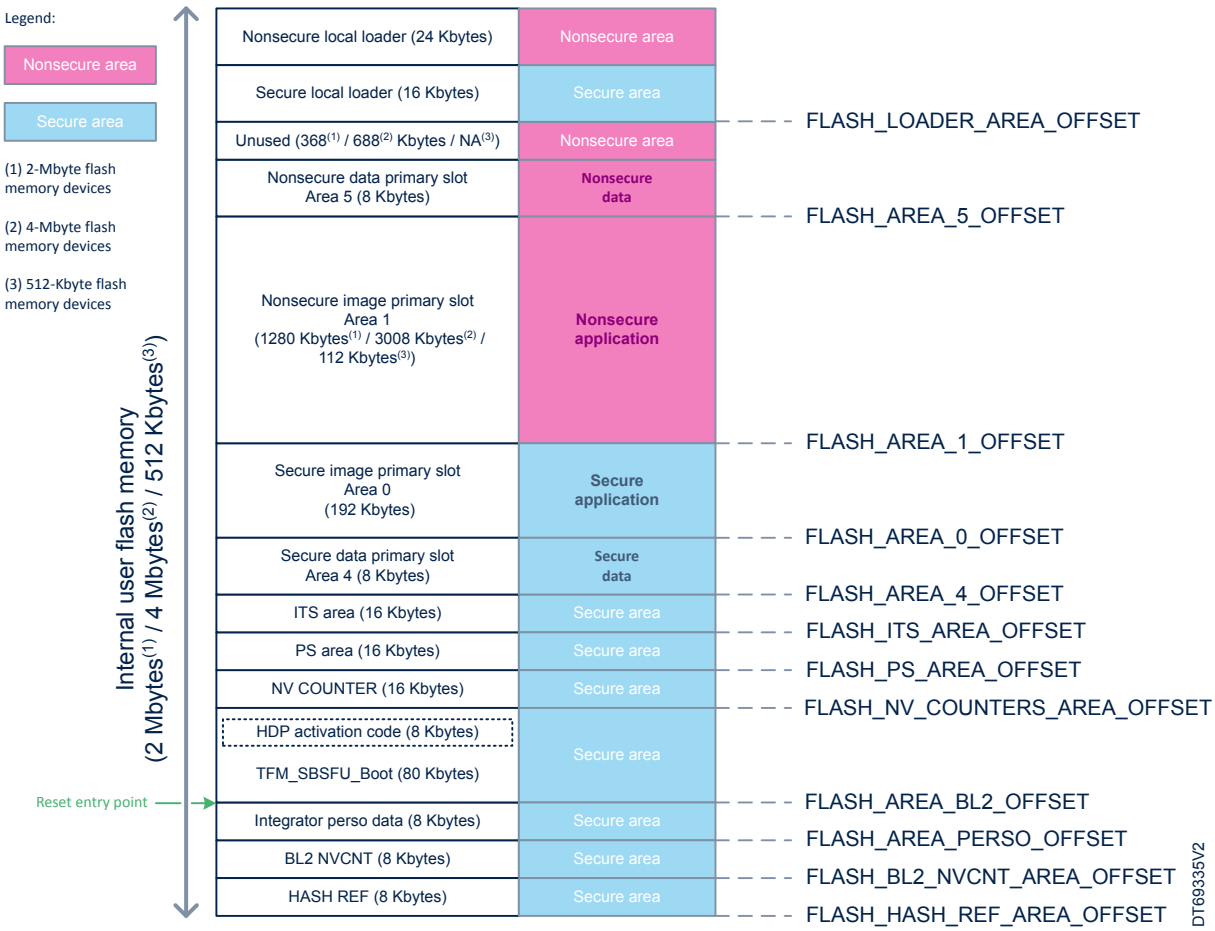
In the TFM application default configuration, the flash memory layout is described in Figure 7.

Figure 7. STM32U5 TFM flash memory layout (default configuration)



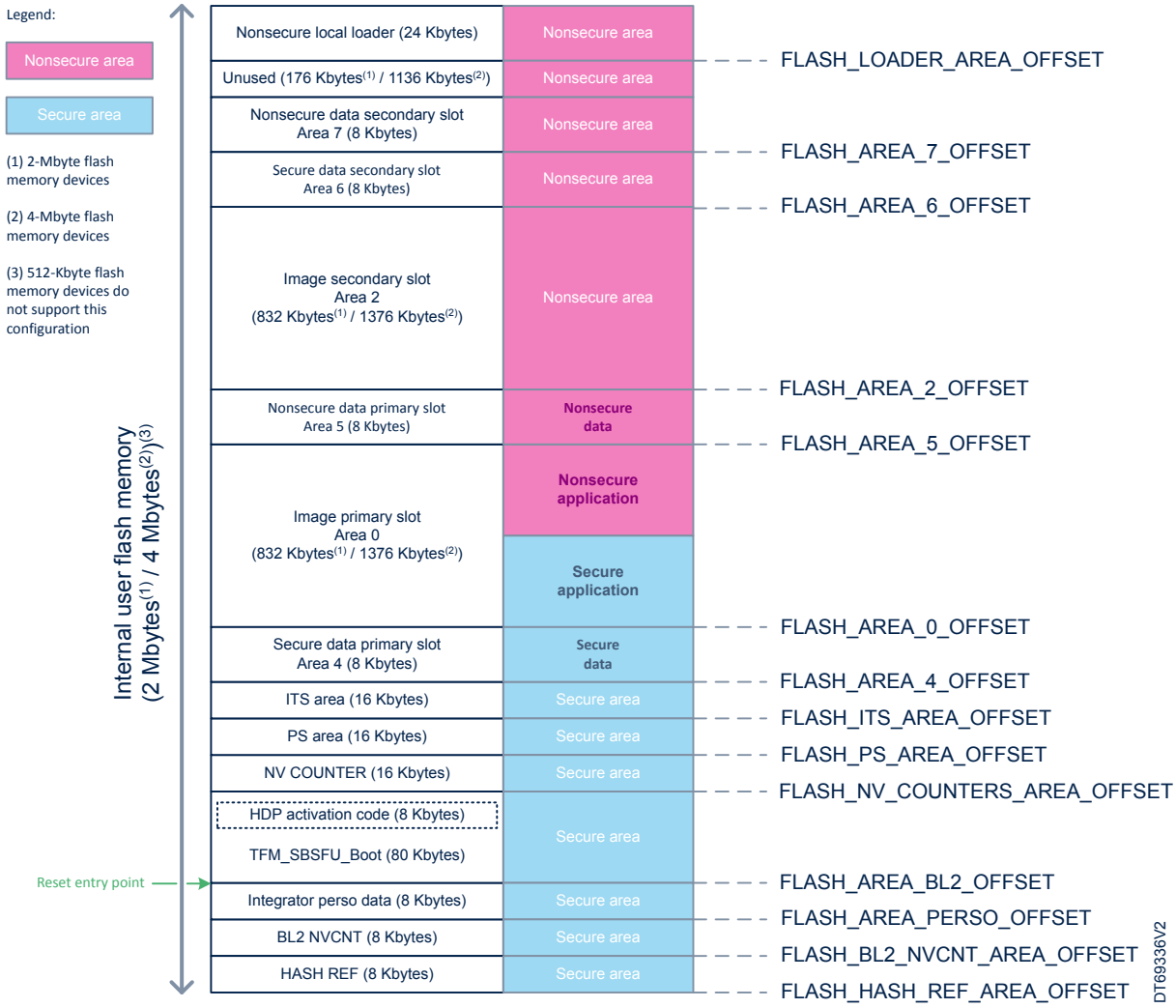
In the case of the primary only slot configuration, the secondary slot areas 2 and 3 (application code) are not present and the secondary slot areas 6 and 7 (data code) are not present. Additionally, a local loader secure region is introduced, to permit the download of a secure image in primary slot area 0 (secure area). In this configuration, the flash memory layout is described in Figure 8.

Figure 8. STM32U5 TFM flash memory layout (primary only slot)



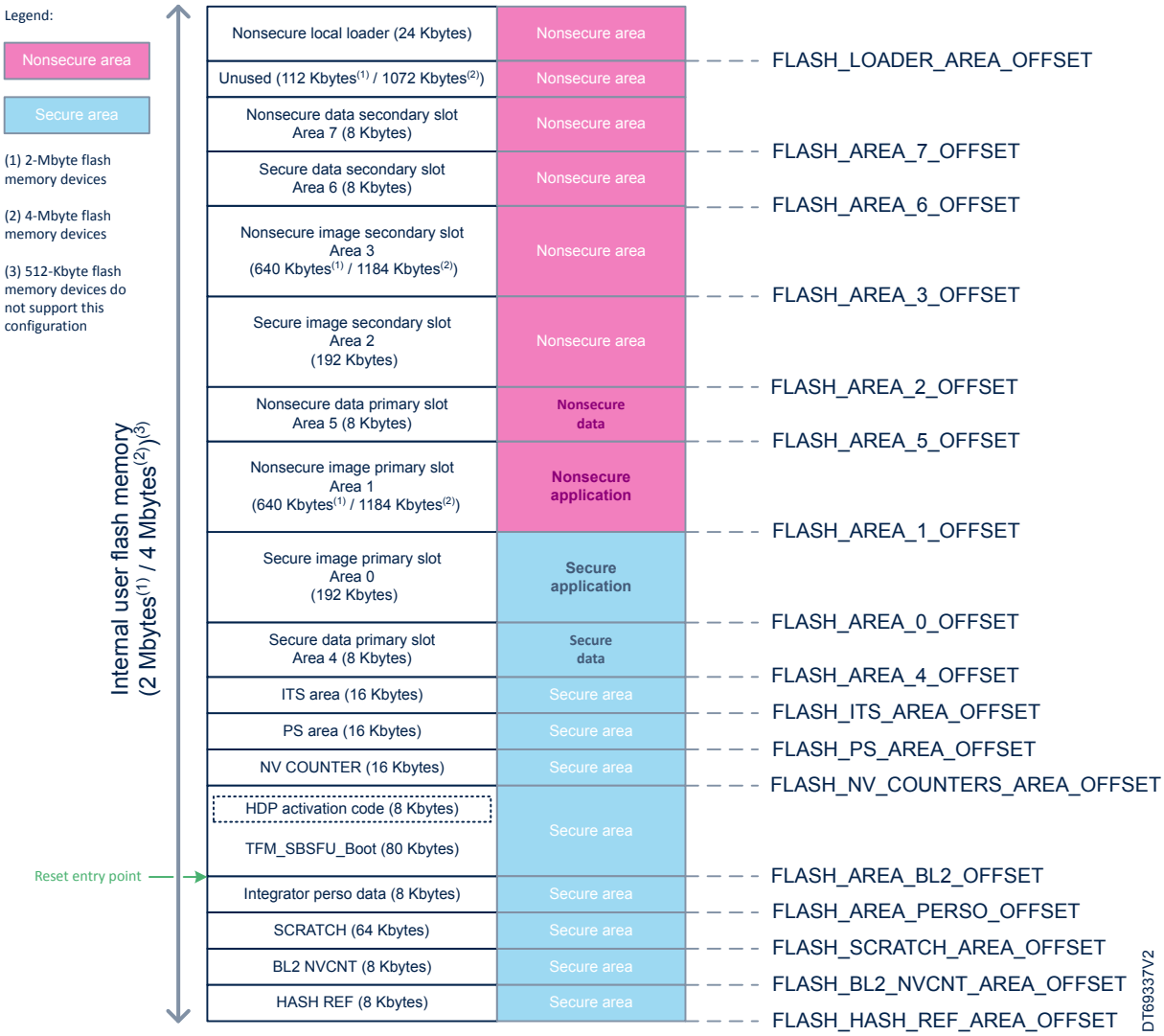
In the case of one image configuration, the primary slot areas 1 and 3 are not present. Slot areas 0 and 2 receive the assembled image with secure and nonsecure binaries. In this configuration, the flash memory layout is described in Figure 9.

Figure 9. STM32U5 TFM flash memory layout (one image)



In the case of the swap mode strategy, a SCRATCH region is introduced to permit the swap of images. In this configuration, the flash memory layout is described in Figure 10.

Figure 10. STM32U5 TFM flash memory layout (swap mode)



In the case of no local loader configuration, the local loader area is unused.

All these configuration items can be combined, so that the corresponding flash memory layout changes are combined as well.

The mechanism of firmware images update depends on the number of images, image upgrade strategy, and slots mode configurations. The procedure is described in the figures below, according to the configuration (for simplification, the data images are not illustrated but the same mechanisms apply to them).

Figure 11. New firmware download and install procedure for overwrite mode, two firmware images configuration, and for primary and secondary slot configuration

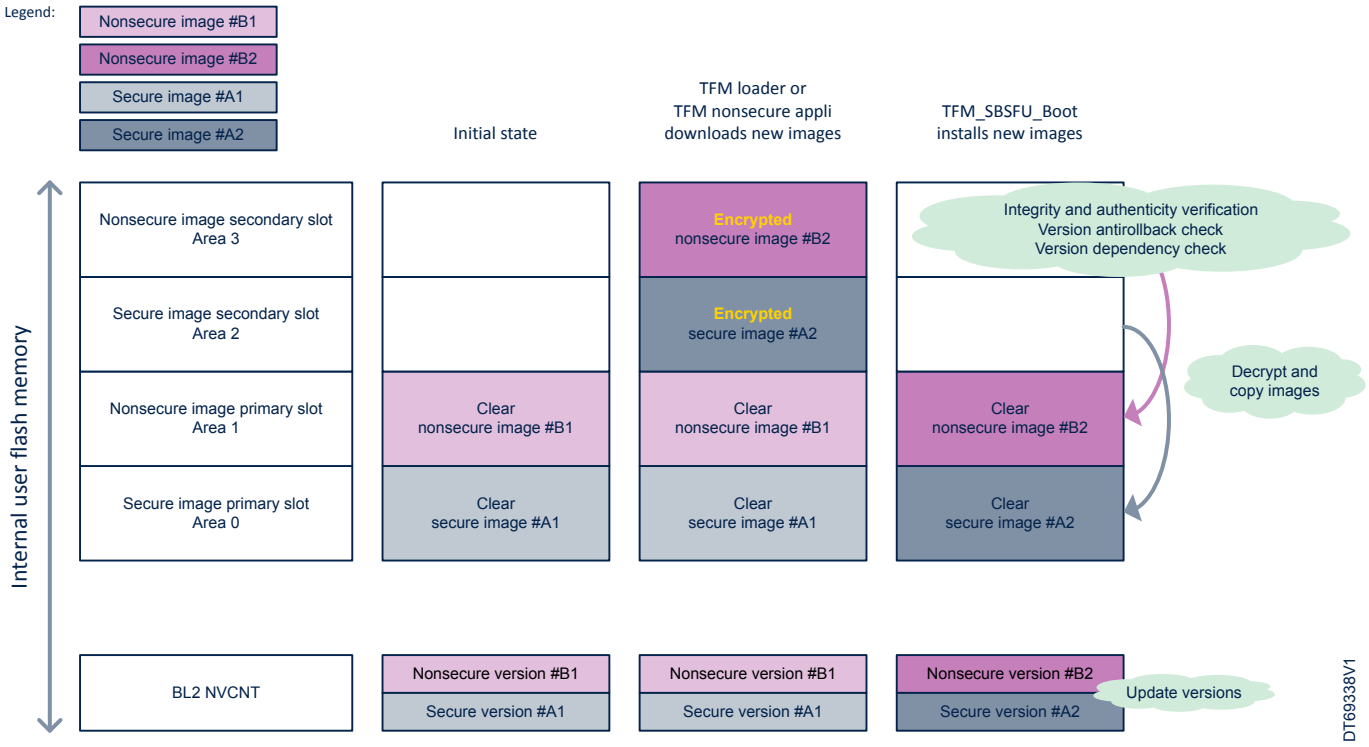


Figure 12. New firmware download and install procedure for overwrite mode, two firmware images configuration and for primary only slot configuration

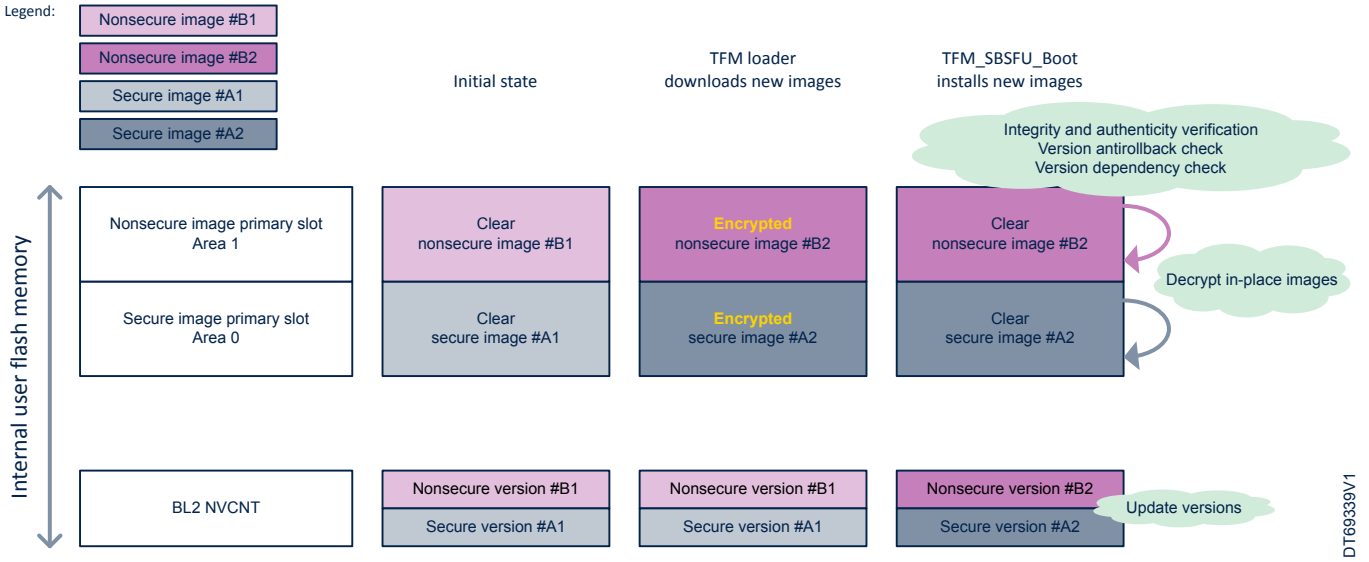


Figure 13. New firmware download and install procedure for overwrite mode, one firmware image configuration and for primary and secondary slot configuration

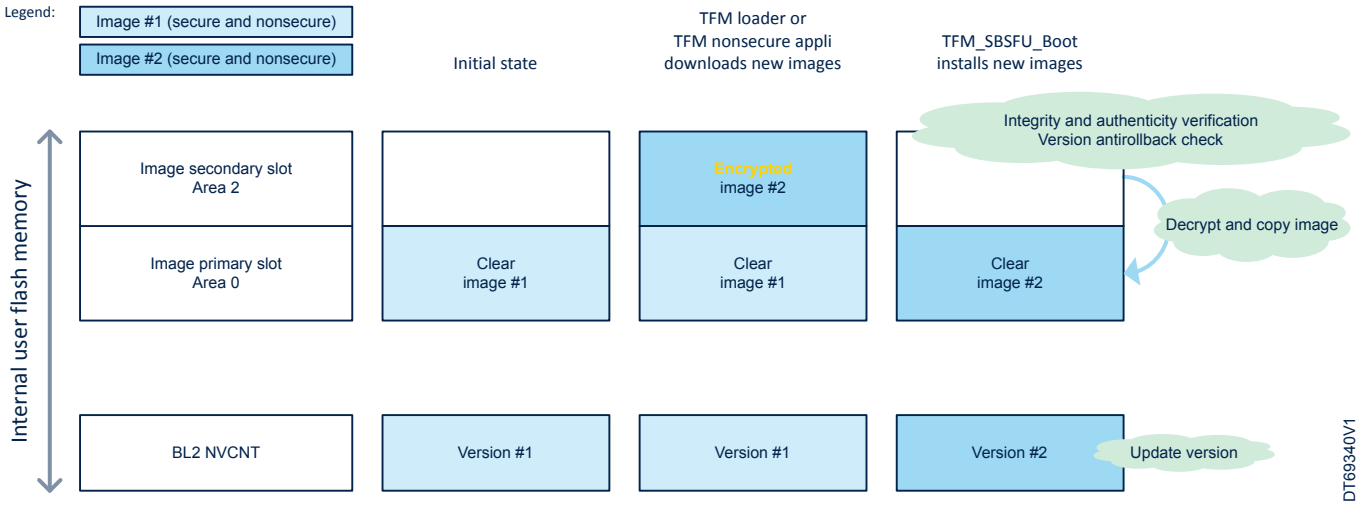


Figure 14. New firmware download and install procedure for overwrite mode, one firmware image configuration and for primary only slot configuration

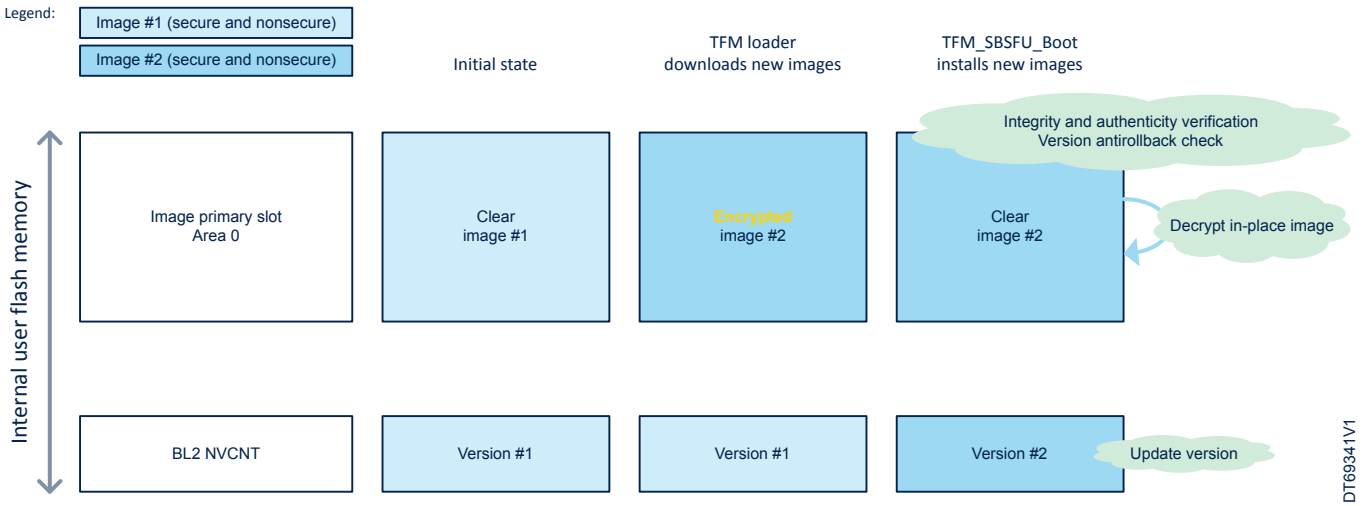


Figure 15. New firmware download and install procedure for swap mode, with images confirmation

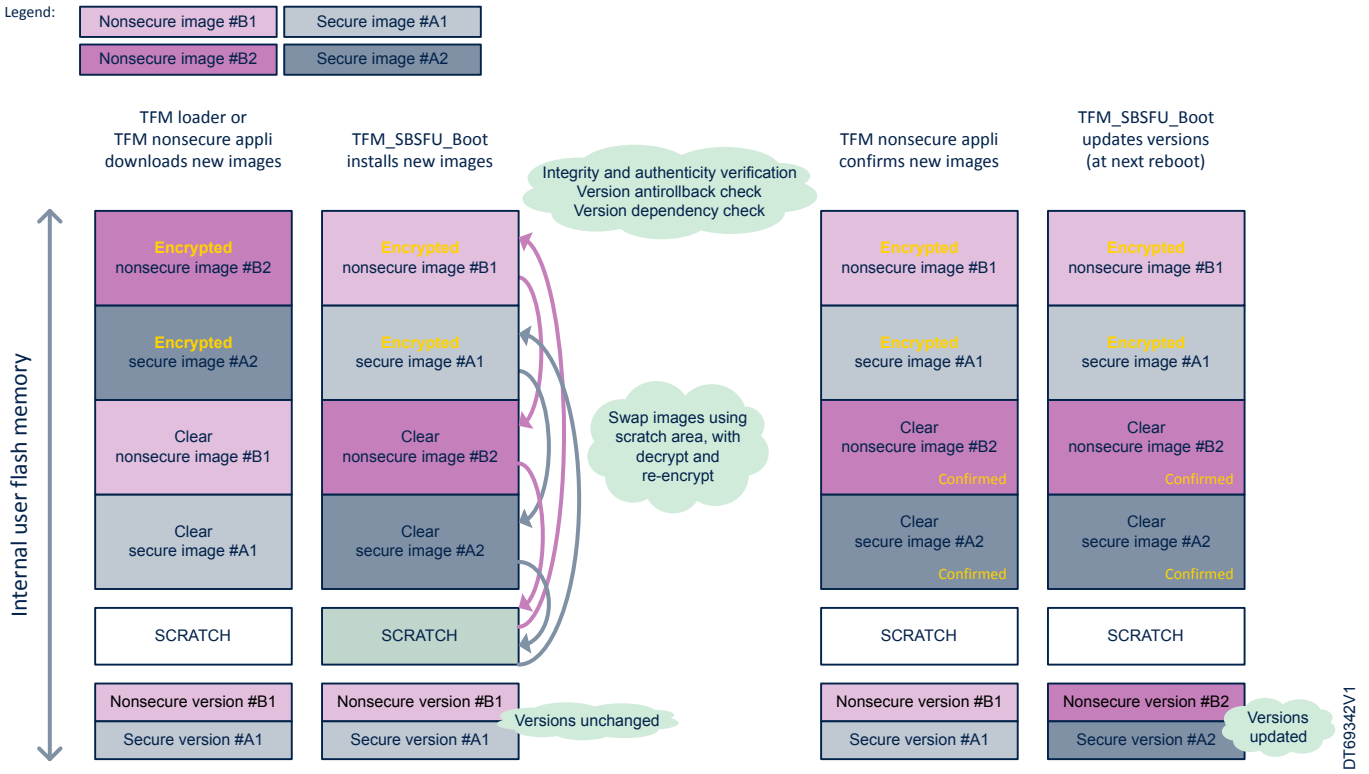
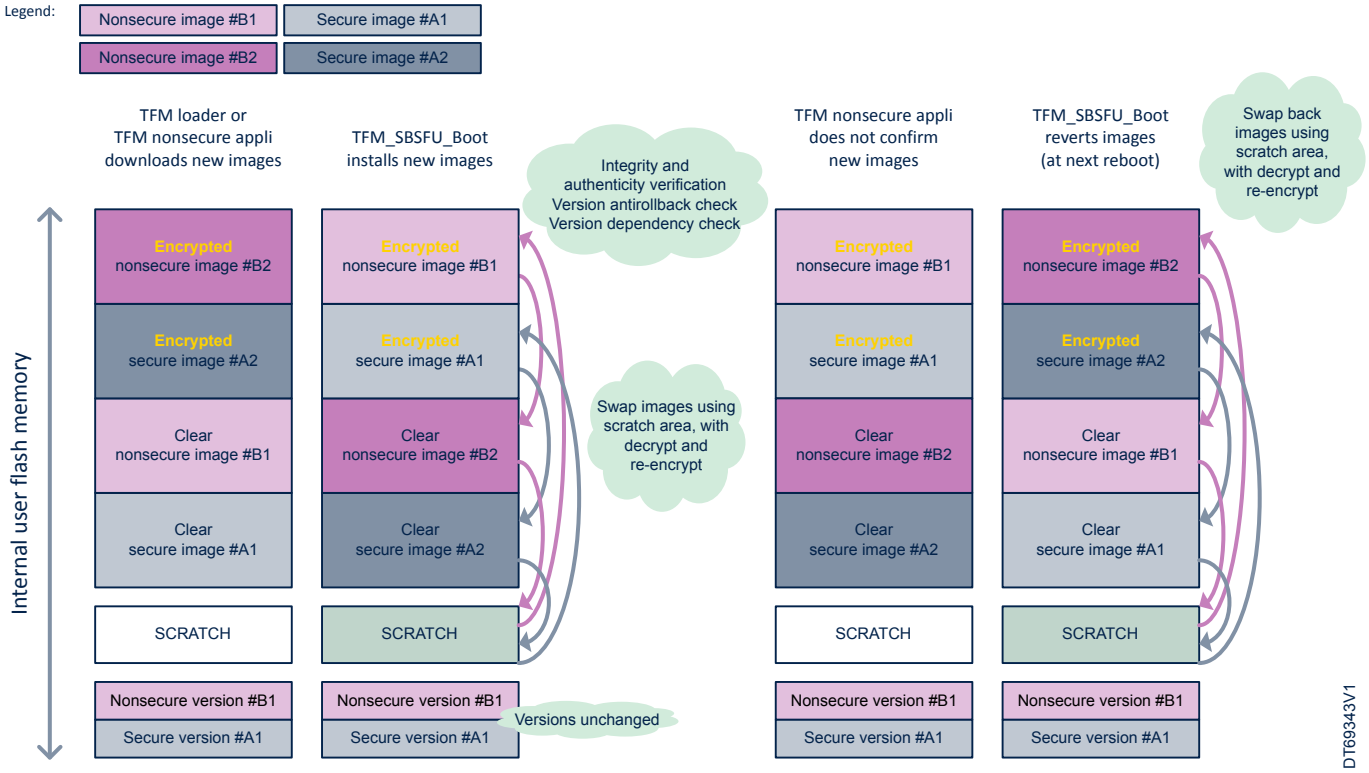


Figure 16. New firmware download and install procedure for swap mode, with images not confirmed



The image slots (secure/non-secure and primary/secondary image slots) contain signed images. A signed image consists in a binary encapsulated by a header (1 Kbyte) and TLV (type-length-value) records containing image metadata (< 1 Kbyte).

To trigger an image installation request, a magic data must be written at the very end position in the image slot (by the application in charge of the download).

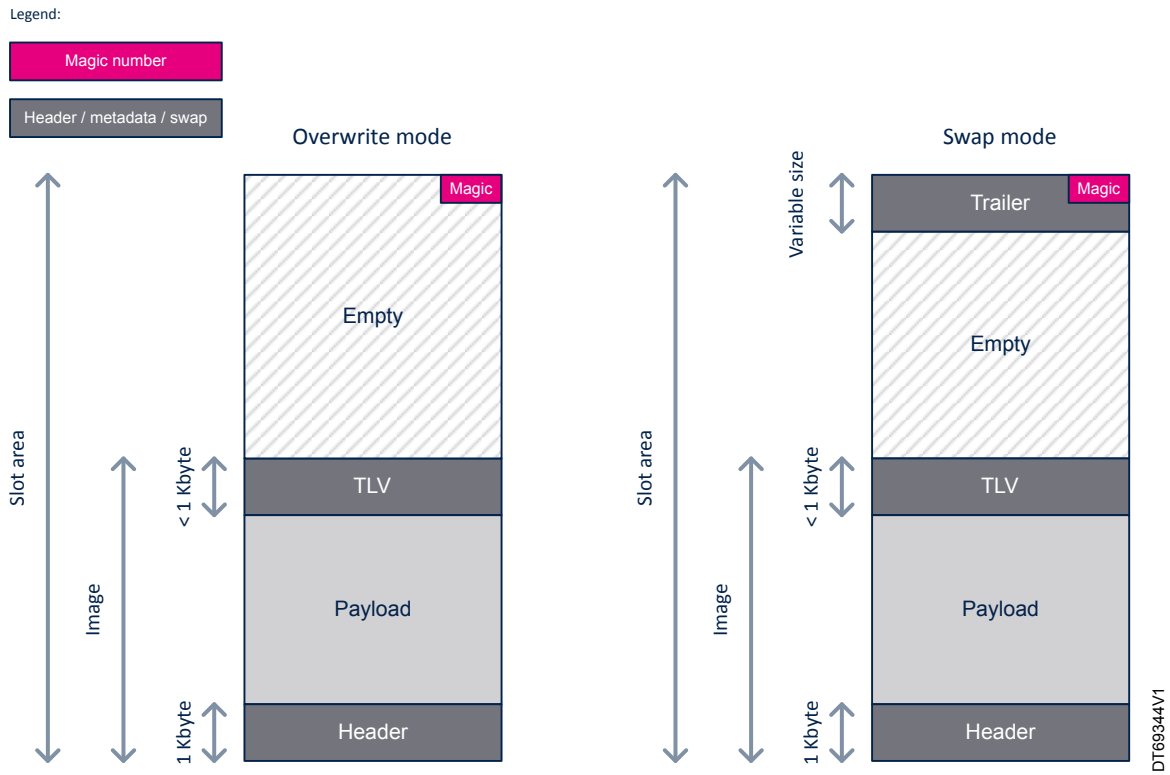
In overwrite mode, the image size is limited to the slot area size minus the magic data size (16 bytes). In swap mode only, a trailer is reserved at the end of the slot, for the swap process, so that the image size is limited to the slot area size minus the trailer size. The trailer size depends on the image and flash memory properties.

Note: Trailer size computation example in the case of a nonsecure slot of a 2-Mbyte flash memory device:

- *Input: the flash memory supports 16-byte wide data read and write operation (not less), with a total of 160 sectors (8-Kbyte page size).*
- *Result: the trailer size is 7680 bytes in direct application of the formula `BOOT_MAX_IMG_SECTORS * min-write-size * 3`.*

For more details, refer to [MCUboot].

Figure 17. Firmware image and slot area



The flash memory layout is common to all IDEs even if the size of generated binaries depends on the compiler (see Note below). The memory layout is defined in two files:

- `Projects\B-U585I-IOT02A\Applications\TFM\Linker\flash_layout.h`
- `Projects\B-U585I-IOT02A\Applications\TFM\Linker\region_defs.h`

Note: It is recommended for the integrator to optimize the default flash memory layout depending on the IDE used and TFM application configuration (refer to Memory footprint).

8.3.2 SRAM layout

The STM32CubeU5 TFM application relies on a dynamic SRAM layout: the SRAM layout is redefined between the TFM_SBSFU_Boot execution and application execution. The SRAM layout defines the following regions:

- TFM_SBSFU_Boot shared area: region where TFM_SBSFU_Boot stores the secure data needed by the secure application in privileged mode for the initial attestation service (boot seed, software measurements, implementation ID, EAT private key, instance ID, life cycle) and secure cryptographic service (provisioned HUK for the Crypto software).
- TFM_SBSFU_Boot volatile area: region used by TFM_SBSFU_Boot for volatile data.
- Secure application privileged volatile area: region used by secure applications in privileged mode for volatile data.
- Secure application unprivileged volatile area: region used by secure applications in unprivileged mode for volatile data.
- Nonsecure application volatile data: region used by nonsecure applications in privileged mode for volatile data.

Figure 18. STM32U5 user SRAM mapping (1 of 2)

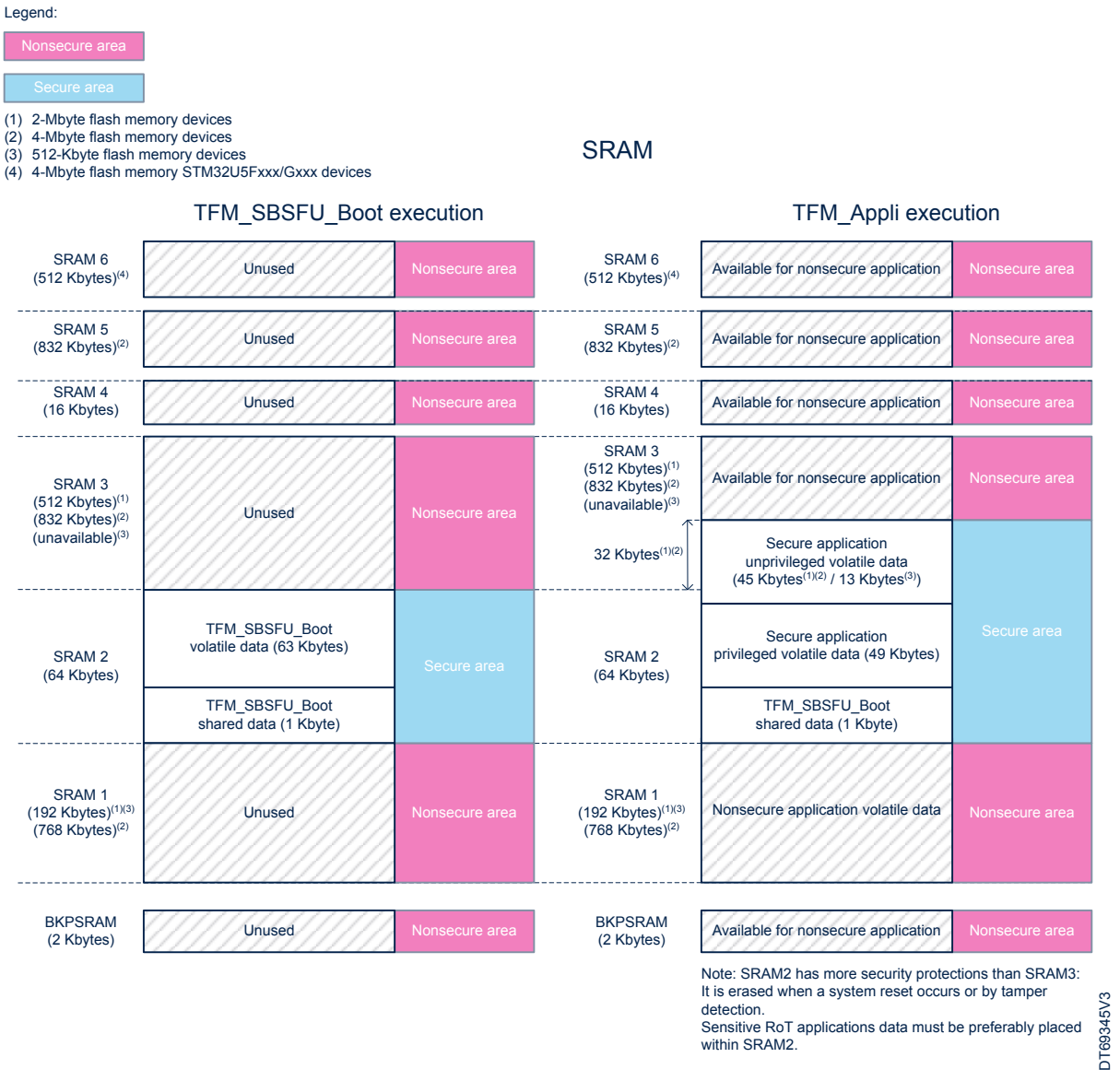


Figure 19. **STM32U5 user SRAM mapping (2 of 2)**

8.4 Folder structure

Figure 20. **Projects file structure (1 of 3)**

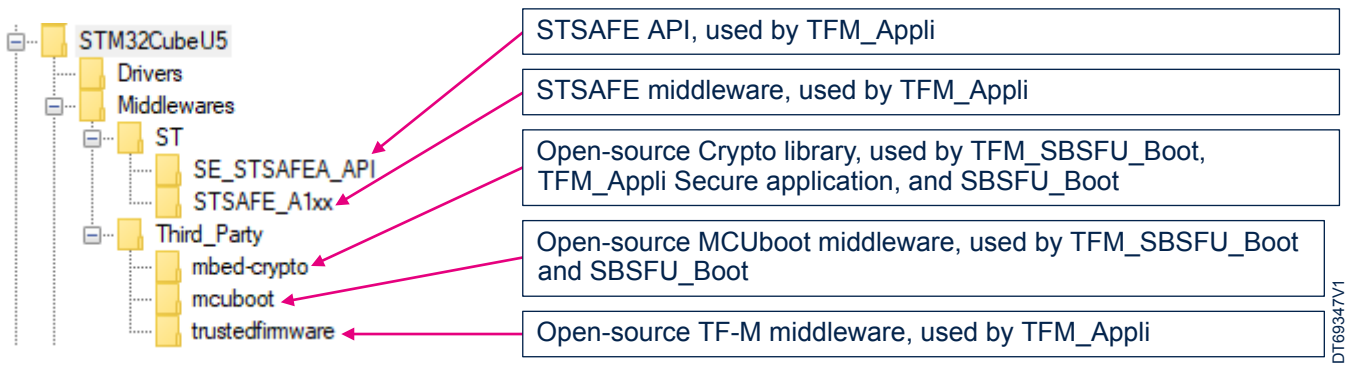


Figure 21. **Projects file structure (2 of 3)**

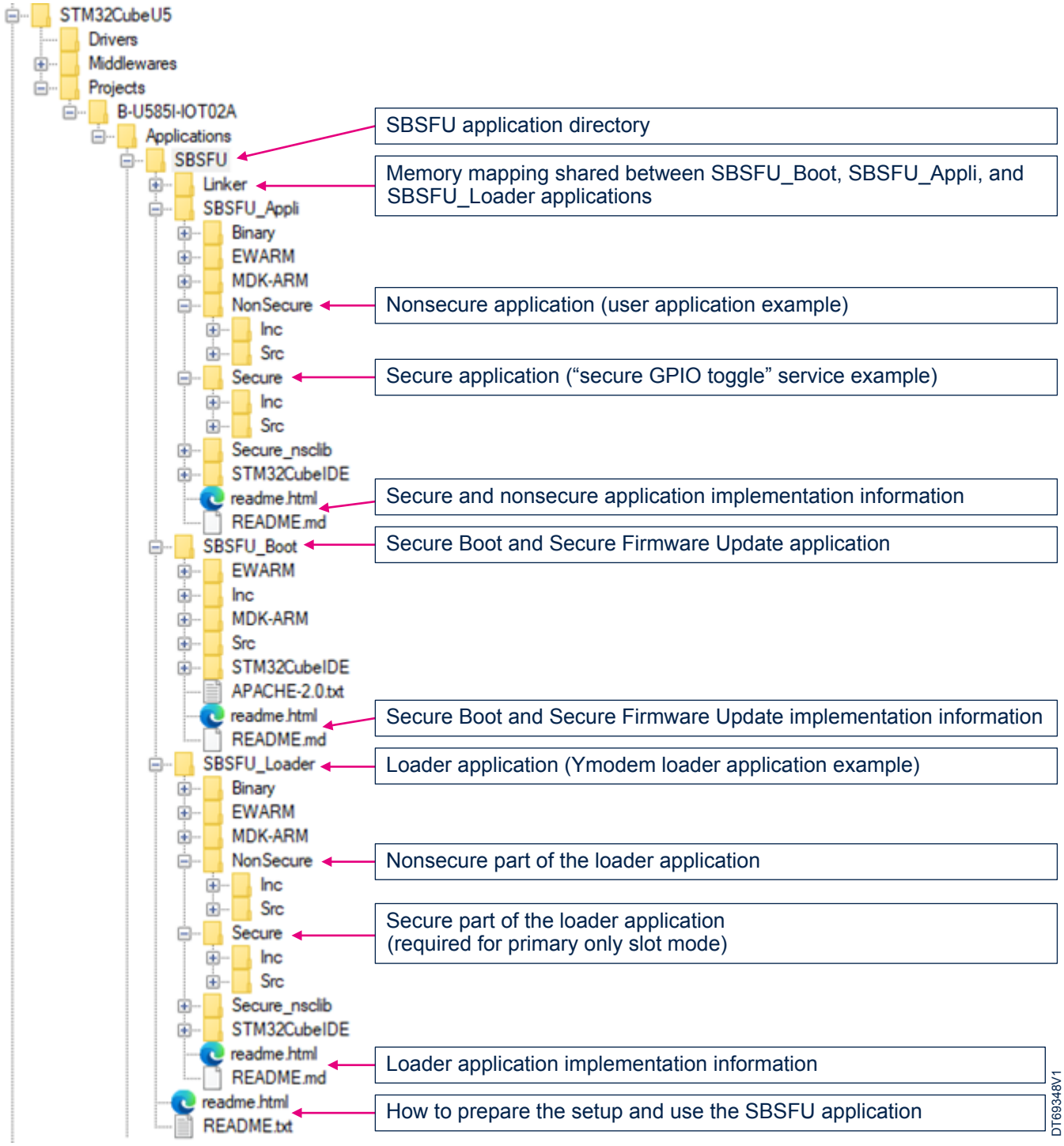
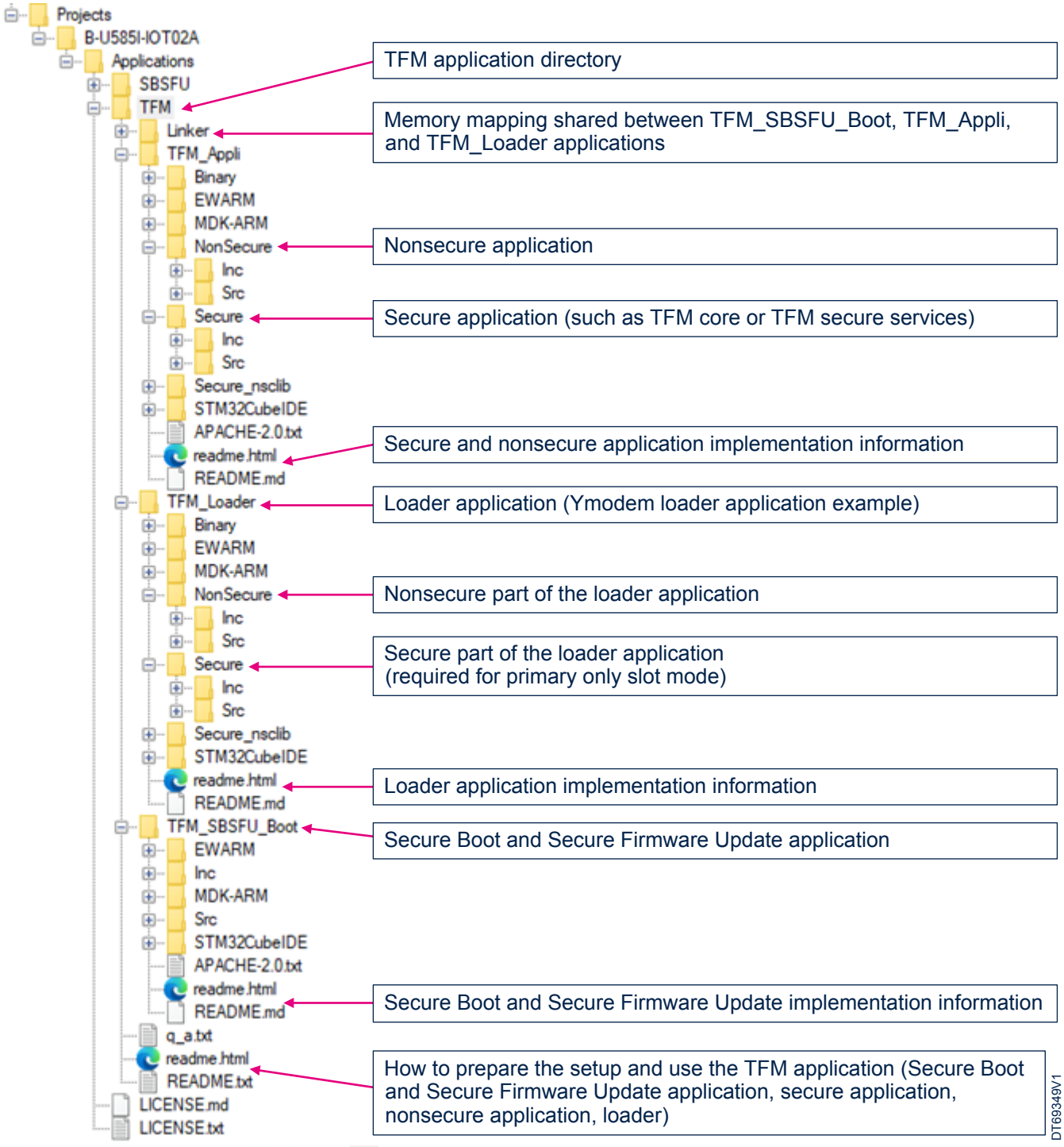


Figure 22. **Projects file structure (3 of 3)**



8.5 APIs

Detailed technical information about the PSA functional APIs is provided in [PSA_API].

9 Hardware and software environment setup

This section describes the hardware and software setup procedures.

9.1 Hardware setup

To set up the hardware environment, connect the ST-LINK USB port of the development board to a personal computer via a USB cable. This connection with the PC allows the user to:

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

Antitamper

- For the B-U585I-IOT02A and STM32U5A9J-DK Discovery kits, the antitamper protection is enabled with active tamper pins usage by default. Refer to Appendix A Development hardware boards for their antitamper protection setups.
- For the NUCLEO-U545RE-Q Nucleo-64 board and the STM32U5G9J-DK2 Discovery kit, the antitamper protection must be disabled as the boards do not support it. Refer to Antitamper in Section 12.1 Configuration.

Boot

- For the B-U585I-IOT02A, STM32U5A9J-DK, and STM32U5G9J-DK2 Discovery kits, it is possible to boot them from the bootloader, or from the flash memory in the development mode (refer to Appendix A Development hardware boards).
- For the NUCLEO-U545RE-Q Nucleo-64 board, it is only possible to boot it by default from the flash memory.

9.2 Software setup

This section lists the minimum requirements for the developer to:

- set up the SDK on a Windows® 10 host
- run the sample scenario
- customize the TFM application delivered in the STM32CubeU5 MCU Package

9.2.1 STM32CubeU5 MCU Package

Copy the STM32CubeU5 MCU Package to the host Windows® hard disk at `C:\data` (for example), or any other path short enough without any spaces.

To get the right certification configuration, for release STM32CubeU5 v1.3.0, the path must be `C:\Packages\en.stm32cubeu5-v1-3-0`, to get such path `C:\Packages\en.stm32cubeu5-v1-3-0\Middlewares\`.

9.2.2 Development toolchains and compilers

Select one of the integrated development environments supported by the STM32CubeU5 MCU Package (refer to the release notes in the STM32CubeU5 MCU Package for the list of supported IDEs).

Allow for the system requirements and setup information provided by the selected IDE provider.

9.2.3 Software tools for programming STM32 microcontrollers

STM32CubeProgrammer (STM32CubeProg) is an all-in-one multi-OS software tool for programming STM32 microcontrollers and microprocessors. It provides an easy to use and efficient environment for reading, writing, and verifying device memory. It can operate 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. 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 (STM32CubeProg) software tool on *www.st.com*.

9.2.4 Terminal emulator

A terminal emulator software is needed to run the application.

It displays some debug information to understand the operations done by the embedded applications. It also permits the interaction with the nonsecure application to trigger some operations.

The example in this document is based on Tera Term, an open-source free software terminal emulator that can be downloaded from the osdn.net/projects/ttssh2/ webpage. Any other similar tool can be used instead (Ymodem protocol support is required).

9.2.5 Python™

The firmware images are produced during the build process using the tool imgtool, located in the MCUboot middleware. Two versions of the tool imgtool are available: the Windows® executable and the Python™ version. By default, the Windows® executable is selected. It is possible to switch to the Python™ version by:

- installing Python™ (Python™ 3.6 or newer) with the required modules from `Middlewares\Third_Party\mcuboot\scripts\requirements.txt`: `pip install -r requirements.txt`
- having Python™ in the execution path variable
- deleting `imgtool.exe` in `Middlewares\Third_Party\mcuboot\scripts\dist\imgtool`

10 Installation procedure

To get a complete installation with security fully activated, the STM32U5 product preparation must be done in four steps:

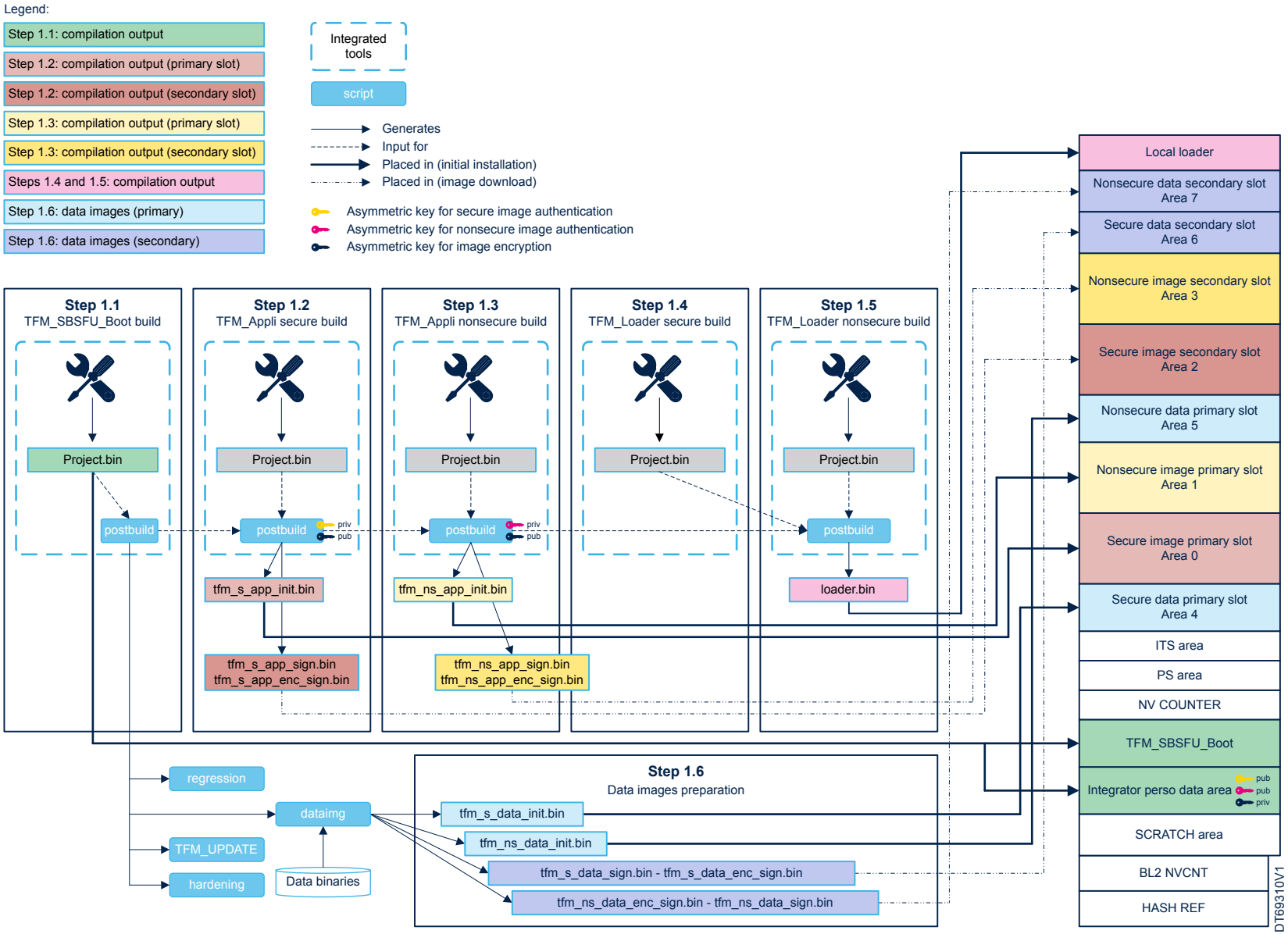
- Step 1: Software compilation (refer to Section 10.1)
- Step 2: STM32U5 device initialization (refer to Section 10.2)
- Step 3: Software programming into the STM32U5 microcontroller internal flash memory (refer to Section 10.3)
- Step 4: Configuring STM32U5 static security protections (refer to Section 10.4)

10.1 Application compilation process

The compilation process is performed in six steps. The six steps are firstly presented in Section 10.1.1 and then further described in Section 10.1.2.

10.1.1 Application compilation overview

Figure 23. Compilation process overview



10.1.2 Application compilation steps

Build the TFM-related projects provided in the STM32CubeU5 MCU Package strictly following the order described in the six steps below.

Step 1.1: build the TFM_SBSFU_Boot application

The TFM_SBSFU_Boot project is in:

`\Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\`

It can be built in the development mode or in the production mode. The build configuration mode can be selected via the project compile switch TFM_DEV_MODE (managed as a preprocessor symbol of the TFM_SBSFU_Boot project):

- Switch TFM_DEV_MODE enabled: development mode
- Switch TFM_DEV_MODE disabled: production mode

By default, this switch is enabled, so that the build configuration is development mode. The development mode makes the development process simpler (see the *Note* below), whereas the production mode is required for security in production. The differences between the two modes are described below:

Table 5. Development versus production mode

Security setting	Development mode	Production mode
BOOT_LOCK static protection	Not required.	Required.
Static protections configuration	Automatically configured by TFM_SBSFU_Boot code at first execution.	Only checked by TFM_SBSFU_Boot code: boot fails if static protections are not at expected values. The user must configure the static protections.
RDP level	RDP level 1.	RDP level 2 (with password).
WRP lock protection	Not required.	Required.
NSBOOTADD0/1 configuration	Not required.	Configured to SECBOOTADD0 value.
TFM_SBSFU_Boot logs on terminal emulator	Enabled.	Disabled.
Error handling	Nonsecure code is executed (presently an infinite loop). This prevents from multiple resets and allows the developers to debug.	The system reset is initiated to start with secure boot code again. This assumes that the executable secure code is functional and present. An exception concerns the control of the expected RDP level when checking the static protections: an infinite loop is executed in the secure part to prevent multiple resets during the RDP regression.

Note: Additionally, before modifying the TFM application, it is recommended to disable the protections (in the `boot_hal_cfg.h` file of the TFM_SBSFU_Boot project). In particular, setting RDP level 0 permits the debug of the TFM application.

Protections can be disabled with the following flags:

```
/* Static protections */
#define TFM_WRP_PROTECT_ENABLE /*!< Write Protection */
#define TFM_HDP_PROTECT_ENABLE /*!< HDP protection */
#define TFM_SECURE_USER_SRAM2_ERASE_AT_RESET /*!< SRAM2 clear at Reset */

#ifdef TFM_DEV_MODE
#define TFM_OB_RDP_LEVEL_VALUE OB_RDP_LEVEL_1 /*!< RDP level */
#else
#define TFM_OB_RDP_LEVEL_VALUE OB_RDP_LEVEL_2 /*!< RDP level */
#endif /* TFM_DEV_MODE */

#define NO_TAMPER            (0) /*!< No tamper activated */
#define INTERNAL_TAMPER_ONLY (1) /*!< Only Internal tamper activated */
#define ALL_TAMPER           (2)/*!< Internal and External tamper activated */
#define TFM_TAMPER_ENABLE ALL_TAMPER

#ifdef TFM_DEV_MODE
#define TFM_OB_BOOT_LOCK 0 /*!< BOOT Lock expected value */
#define TFM_ENABLE_SET_OB
    /*!< Option bytes are set by TFM_SBSFU_Boot when not correctly set */
#define TFM_ERROR_HANDLER_NON_SECURE
    /*!< Error handler is in Non Secure , this allows regression without jumping */
#else
#define TFM_WRP_LOCK_ENABLE /*!< Write Protection Lock */
#define TFM_OB_BOOT_LOCK 1 /*!< BOOT Lock expected value */
#define TFM_NSBOOT_CHECK_ENABLE
    /*!< NSBOOTADD0 and NSBOOTADD1 must be set to TFM_SBSFU_Boot Vector */
#endif /* TFM_DEV_MODE */

/* Run time protections */
#define TFM_FLASH_PRIVONLY_ENABLE /*!< Flash Command in Privileged only */
#define TFM_BOOT_MPU_PROTECTION
    /*!< TFM_SBSFU_Boot uses MPU to prevent execution outside of TFM_SBSFU_Boot code */
```

Build the project, using the selected IDE.

This step creates the Secure Boot and Secure Firmware Update binary including provisioned user data such as keys and IDs. Check that the binary is correctly created at this location:

- EWARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\EWARM\B-U585I-IOT02A\Exe\Project.bin`
- MDK-ARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\MDK-ARM\B-U585I-IOT02A\Exe\Project.bin`
- STM32CubeIDE: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\STM32CubeIDE\Release\TFM_SBSFU_Boot.bin`

Step 1.2: build the TFM_Appli secure application

The TFM_Appli secure project is in `\Projects\B-U585I-IOT02A\Applications\TFM\TFM_Appli\`, together with the TFM_Appli nonsecure project.

Build the TFM_Appli secure project, using the selected IDE.

This step creates the TFM secure binary. Check that the binary is correctly created at this location:

- EWARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Appli\EWARM\Secure\B-U585I-IOT02A_S\Exe\Project.bin`
- MDK-ARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Appli\MDK-ARM\Secure\B-U585I-IOT02A_S\Exe\Project.bin`
- STM32CubeIDE: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Appli\STM32CubeIDE\Secure\Release\TFM_Appli_Secure.bin`

Additionally, by means of the postbuild command integrated in the IDE project, it also produces the following application images:

- The clear TFM secure signed application image for initial installation in `TFM_Appli\Binary\tfm_s_app_init.bin`
- The encrypted TFM secure signed application image for download in `TFM_Appli\Binary\tfm_s_app_enc_sign.bin`
- The clear TFM secure signed application image for download in `TFM_Appli\Binary\tfm_s_app_sign.bin`

The postbuild command is relying on the tool imgtool located in the MCUboot middleware. The produced firmware images can be parsed using the command `imgtool verify`, to extract Header and TLV details.

Note: If the firmware location does not fulfill the conditions indicated in Section 9.2.1 STM32CubeU5 MCU Package, an error may occur during the postbuild script. Any postbuild script error is reported in an `output.txt` file.

For more information on the signed and encrypted binary formats, please refer to the [MCUboot] open-source website.

Step 1.3: build the TFM_Appli nonsecure application

The TFM_Appli nonsecure project is in `\Projects\B-U585I-IOT02A\Applications\TFM\TFM_Appli\`, together with the TFM_Appli secure project.

Build the TFM_Appli nonsecure project, using the selected IDE.

This step creates the TFM nonsecure binary. Check that the binary is correctly created at this location:

- EWARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Appli\EWARM\NonSecure\B-U585I-IOT02A_NS\Exe\Project.bin`
- MDK-ARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Appli\MDK-ARM\NonSecure\B-U585I-IOT02A_NS\Exe\Project.bin`
- STM32CubeIDE: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Appli\STM32CubeIDE\NonSecure\Release\TFM_Appli_NonSecure.bin`

Additionally, by means of the postbuild command integrated in the IDE project, it also produces the following application images:

- The clear TFM nonsecure signed application image for initial installation in `TFM_Appli\Binary\tfm_ns_app_init.bin`
- The encrypted TFM nonsecure signed application image for download in `TFM_Appli\Binary\tfm_ns_app_enc_sign.bin`
- The clear TFM nonsecure signed application image for download in `TFM_Appli\Binary\tfm_ns_app_sign.bin`

The postbuild command is relying on the tool imgtool located in MCUboot middleware. The produced firmware images can be parsed using the command `imgtool verify`, to extract Header and TLV details.

Note: If the firmware location does not fulfill the conditions indicated in Section 9.2.1 STM32CubeU5 MCU Package, an error may occur during the postbuild script.

For more information on the signed and encrypted binary formats, please refer to the [MCUboot] open-source website.

Step 1.4: build the TFM_Loader secure application

The TFM_Loader secure project is in `\Projects\B-U585I-IOT02A\Applications\TFM\TFM_Loader`, together with the TFM_Loader nonsecure project.

Build the TFM_Loader secure project, using the selected IDE.

This step creates the TFM loader secure binary. Check that the binary is correctly created at this location:

- EWARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Loader\EWARM\Secure\B-U585I-IOT02A_S\Exe\Project.bin`
- MDK-ARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Loader\MDK-ARM\Secure\B-U585I-IOT02A_S\Exe\Project.bin`
- STM32CubeIDE: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Loader\STM32CubeIDE\Secure\Release\TFM_Loader_Secure.bin`

Step 1.5: build the TFM_Loader nonsecure application

The TFM_Loader nonsecure project is in: `\Projects\B-U585I-IOT02A\Applications\TFM\TFM_Loader`.

Build the TFM_Loader nonsecure project, using the selected IDE.

This step creates the TFM_Loader nonsecure binary. Check that the binary is correctly created at this location:

- EWARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Loader\EWARM\NonSecure\B-U585I-IOT02A_NS\Exe\Project.bin`
- MDK-ARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Loader\MDK-ARM\NonSecure\B-U585I-IOT02A_NS\Exe\Project.bin`
- STM32CubeIDE: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Loader\STM32CubeIDE\NonSecure\Release\TFM_Loader_NonSecure.bin`

Additionally, by means of the postbuild command integrated in the IDE project, it also produces the TFM_Loader image in `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Loader\Binary\loader.bin`. In the case of primary and secondary slot configuration, the TFM_Loader image is produced from the TFM_Loader nonsecure binary only. In the case of a primary-only slot configuration, the TFM_Loader image is produced from the assembled TFM_Loader secure and nonsecure binaries.

Step 1.6: data images preparation

The TFM_SBSFU_Boot project is delivered with the default raw data binaries of a secure data image (s_data.bin) and a nonsecure data image (ns_data.bin). Both binaries are available in: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\Src\`. They are provisioned by default with an EAT private key and dummy data respectively as an example.

To prepare the data images, execute the script (before programming):

- EWARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\EWARM\dataimg.bat`
- MDK-ARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\MDK-ARM\dataimg.bat`
- STM32CubeIDE: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\STM32CubeIDE\dataimg.sh`

The script produces the nonsecure and secure binaries in `TFM_Appli\Binary`:

- Nonsecure binaries:
  - Clear TFM nonsecure signed data image for initial installation (tfm_ns_data_init.bin)
  - Encrypted TFM nonsecure signed data image for download (tfm_ns_data_enc_sign.bin)
  - Clear TFM nonsecure signed data image for download (tfm_ns_data_sign.bin)
- Secure binaries:
  - Clear TFM secure signed data image for initial installation (tfm_s_data_init.bin)
  - Encrypted TFM secure signed data image for download (tfm_s_data_enc_sign.bin)
  - Clear TFM secure signed data image for download (tfm_s_data_sign.bin)

Note: Ignore this step for configurations without data images. Otherwise, run the step at least once. It must then be applied each time a modification is operated on the configuration that affects the images production (crypto scheme for instance).

10.2 STM32U5 device initialization

The STM32U5 microcontroller initialization consists in the following operations:

- Enabling the TrustZone® mode
- Disabling the security protections in the option bytes
- Erasing the flash memory
- Setting a default OEM2 password (RDP regression)

This can be achieved by using the STM32CubeProgrammer (STM32CubeProg) tool.

To make the device initialization procedure easier, execute an automatic script relying on the STM32CubeProgrammer CLI in the STM32CubeU5 MCU Package:

- EWARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\EWARM\regression.bat`
- MDK-ARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\MDK-ARM\regression.bat`
- STM32CubeIDE: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\STM32CubeIDE\regression.sh`

When using this automatic script, the user must check that there is no error reported during the script execution.

As an alternative, it is possible to initialize and verify manually the option bytes configuration by means of the STM32CubeProgrammer GUI through the steps below.

Step 2.1 - Connection: connect to target with hot plug mode selected

Figure 24. STM32CubeProgrammer connection menu

DT69309V1

Step 2.2 - Option bytes settings: menu *Option bytes*/*User Configuration*

The following option bytes values must be set:

- RDP level 0
- SWAP_BANK: unchecked (bank1 and bank2 are not swapped)
- DBANK: checked (dual-bank mode with 64-bit data)
- SRAM2-RST: unchecked (SRAM2 erased when a system reset occurs)
- TZEN: checked (global TrustZone® security enabled)
- HDP1: disabled (hide protection area)
- HDP2: disabled (hide protection area)
- NSBOOTADD0: 0x100000 (0x08000000) (NS user flash memory address)
- NSBOOTADD1: 0x17F200 (0x0BF90000) (system bootloader address)
- SECBOOTADD0: 0x1800C0 (0x0C006000) (Secure Boot base address 0)
- BOOT_LOCK: unchecked (boot based on the pad/option bit configuration)
- nSWBOOT0: checked (BOOT0 taken from the PH3/BOOT0 pin)
- SECWM1: enabled on complete bank 1 (secure Area 1)
- WRP1A: disabled and unlocked (bank 1 write protection for area A)
- WRP1B: disabled and unlocked (bank 1 write protection for area B)
- SECWM2: enabled on complete bank 2 (secure Area 2)
- WRP2A: disabled and unlocked (bank 2 write protection for area A)
- WRP2B: disabled and unlocked (bank 2 write protection for area B)

Figure 25. STM32CubeProgrammer *Option bytes* screen (*Read Out Protection*)

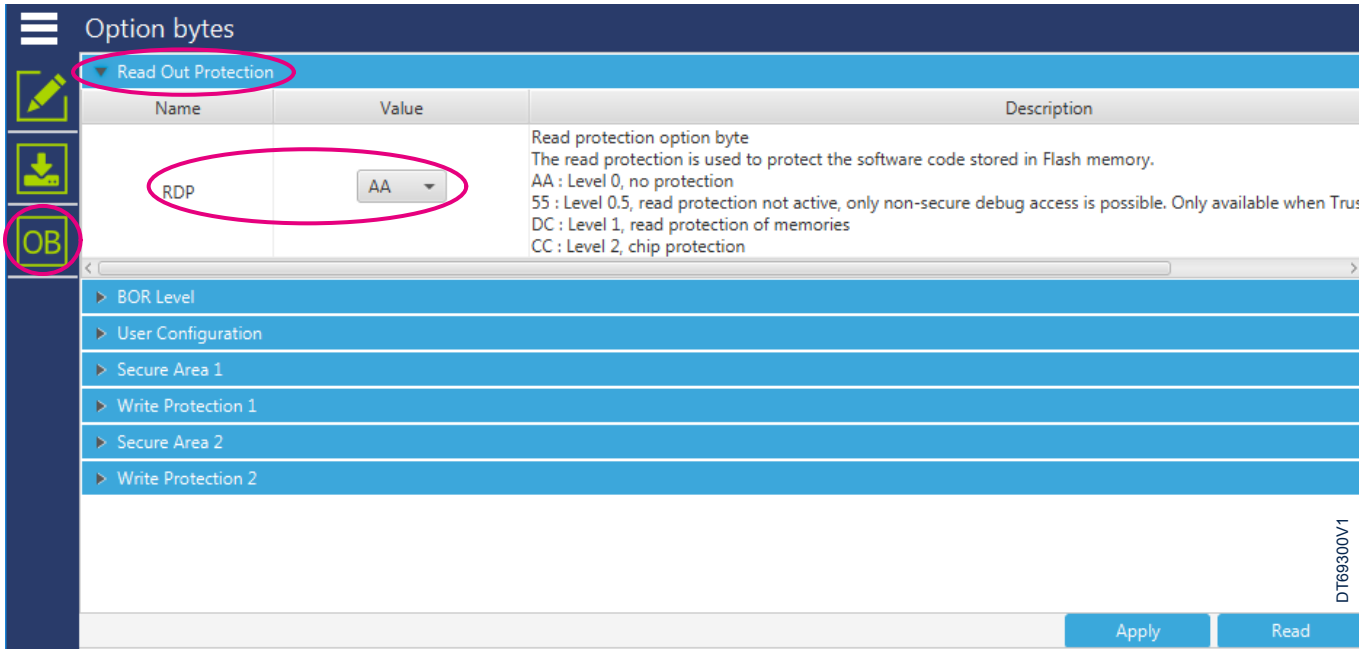


Figure 26. **STM32CubeProgrammer *Option bytes* screen (*User Configuration* - part 1)**

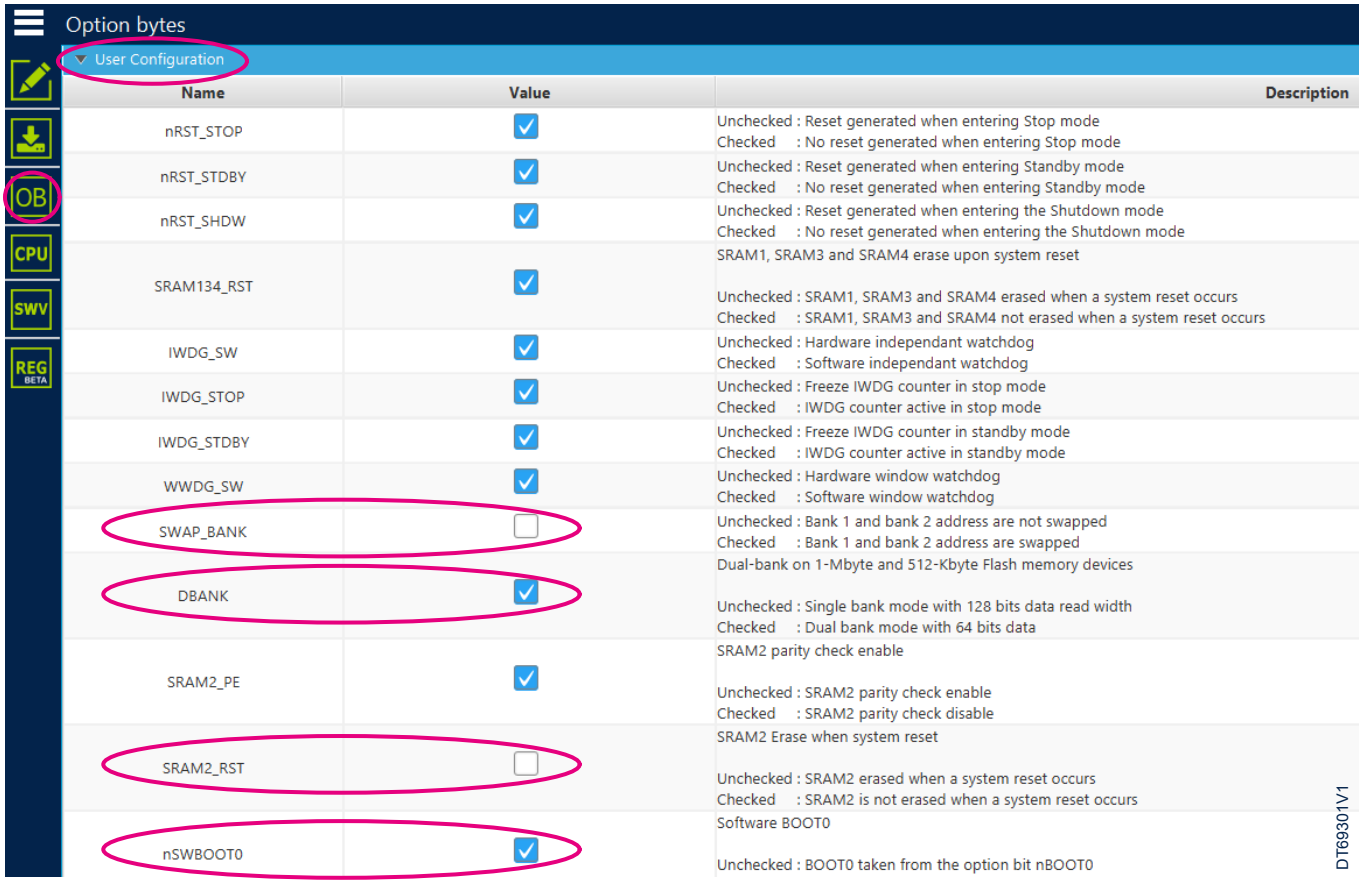


Figure 27. **STM32CubeProgrammer *Option bytes* screen (*User Configuration* - part 2)**

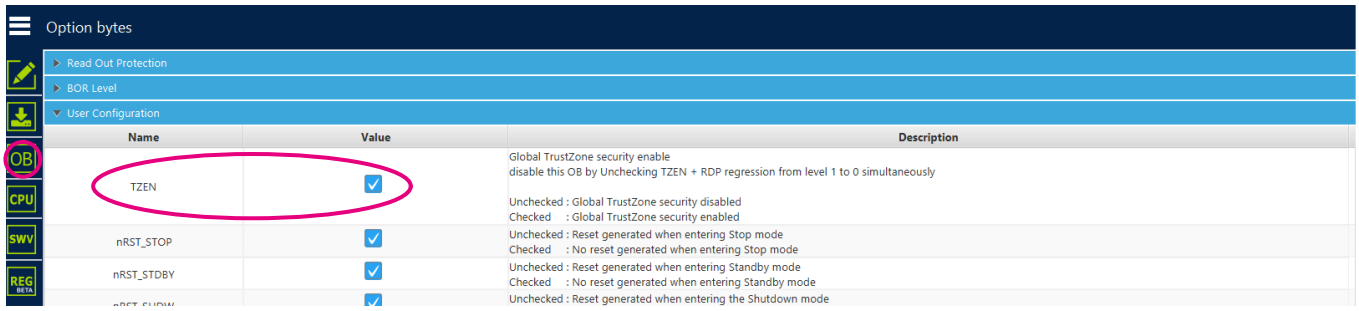


Figure 28. **STM32CubeProgrammer *Option bytes* screen (*Boot Configuration*)**

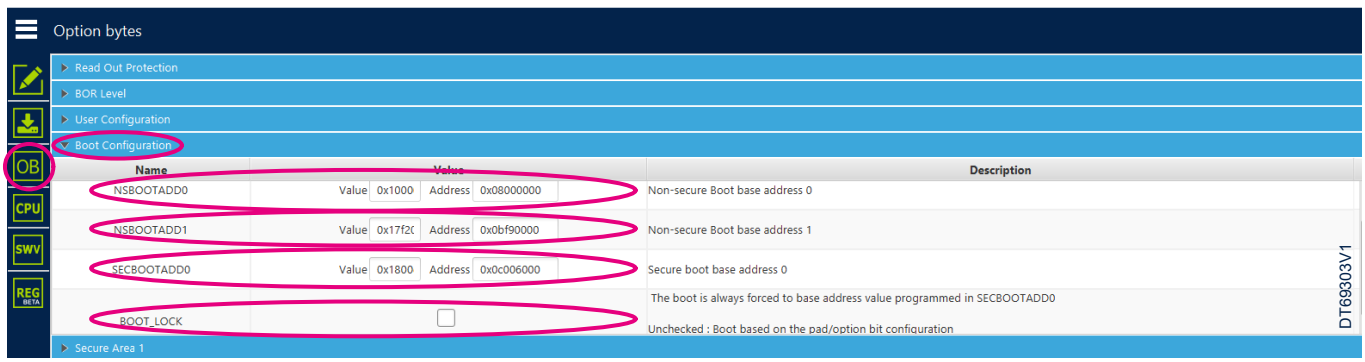


Figure 29. **STM32CubeProgrammer *Option bytes* screen (*Secure Area 1*)**

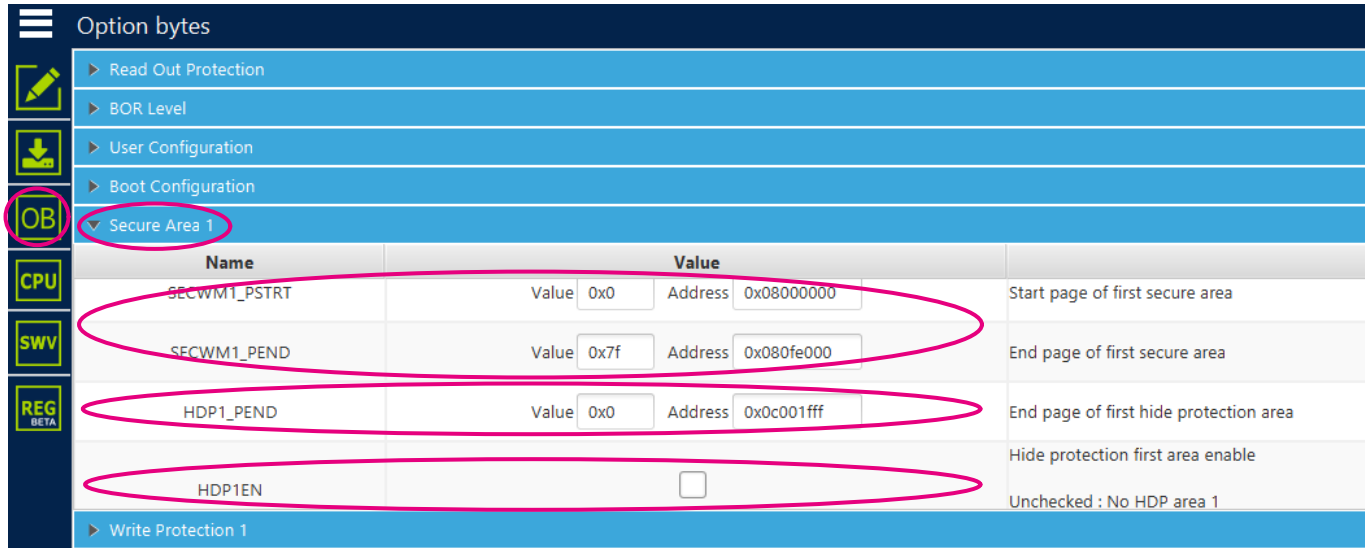


Figure 30. **STM32CubeProgrammer *Option bytes* screen (*Write Protection 1*)**

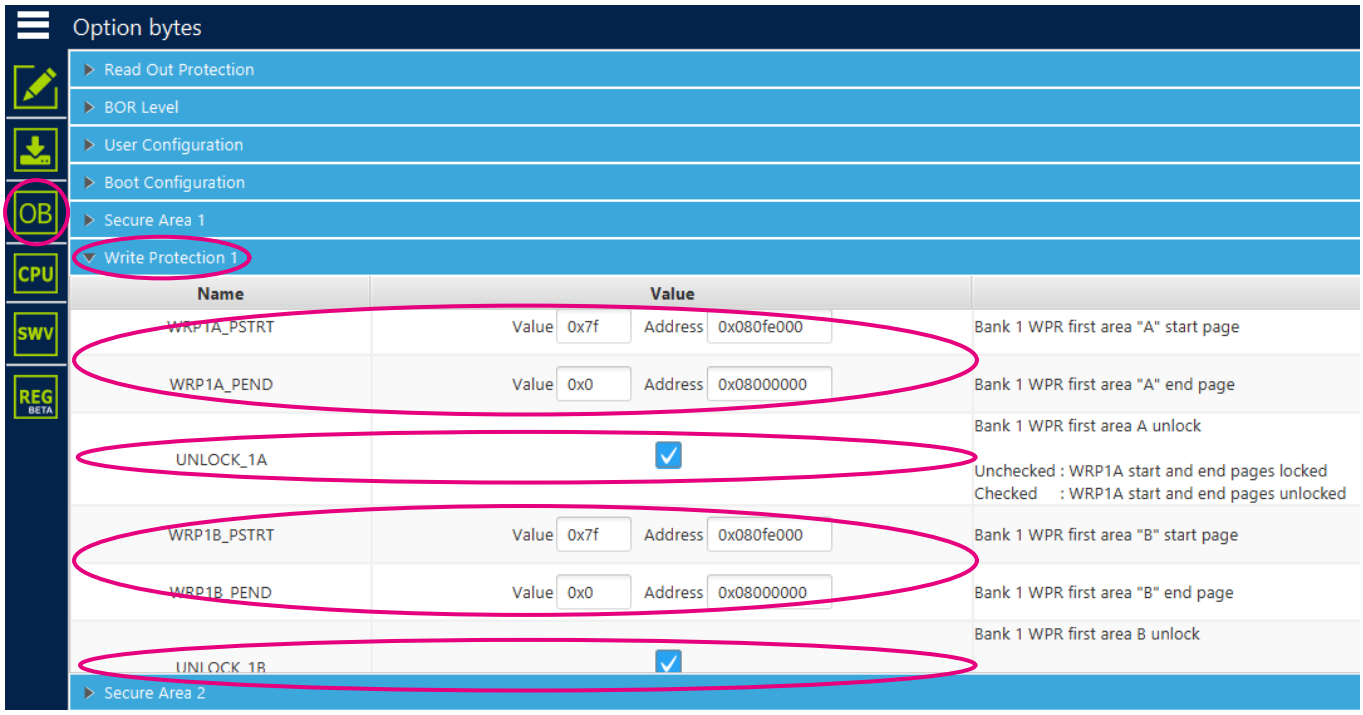


Figure 31. **STM32CubeProgrammer *Option bytes* screen (*Secure Area 2*)**

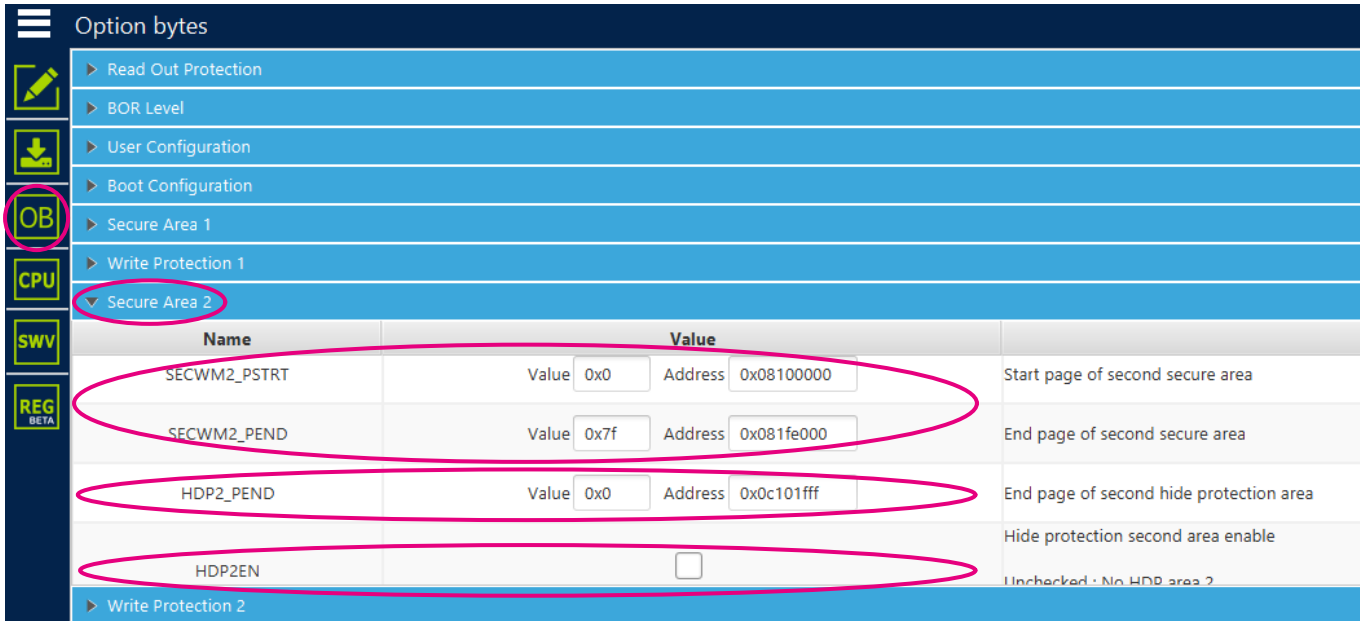


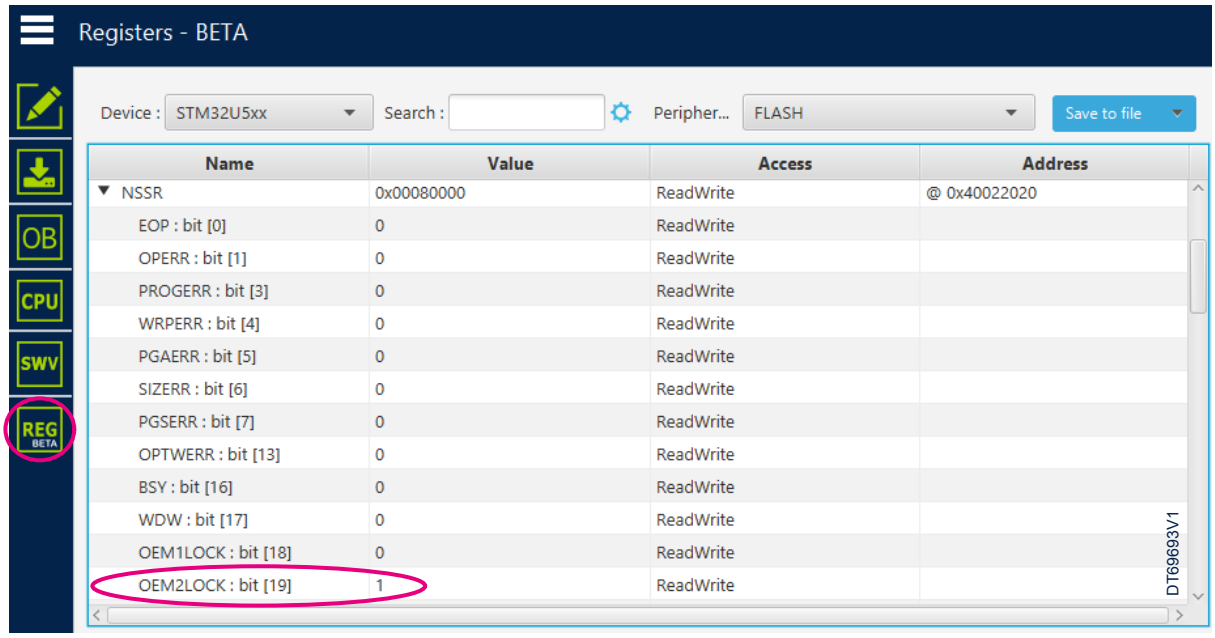
Figure 32. **STM32CubeProgrammer *Option bytes* screen (*Write Protection 2*)**

Step 2.3 - Check OEM2 password provisioning state

The OEM2LOCK option bit is set in FLASH_NSSR after having provisioned an OEM2 password. If a password is already provisioned other than the default one proposed in the regression script, apply one of the two following solutions:

- either clear it and then use the default password that is proposed in the regression script (development)
- or simply update the regression script with this new password (personalization before production)

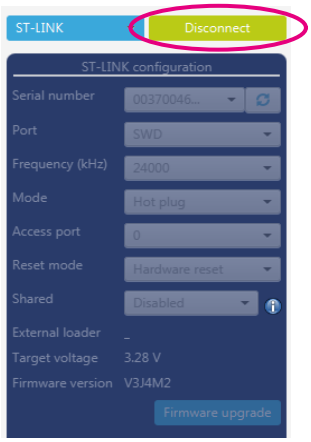
Figure 33. **STM32CubeProgrammer flash memory nonsecure status register screen (OEM2LOCK)**



Refer to [AN5347] for password provisioning and clearing methods.

Step 2.4 - Disconnect

Figure 34. **STM32CubeProgrammer disconnect**



10.3 Software programming into STM32U5 internal flash memory

To make the programming of the generated binaries in internal flash memory easier, execute the automatic script relying on the STM32CubeProgrammer (STM32CubeProg) CLI in the STM32CubeU5 MCU Package:

- EWARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\EWARM\TFM_UPDATE.bat`
- MDK-ARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\MDK-ARM\TFM_UPDATE.bat`
- STM32CubeIDE: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\STM32CubeIDE\TFM_UPDATE.sh`

The script programs all the generated binaries/images into the flash memory. Data format (clear or encrypted) and flash memory location depend on the system configuration used. The script is dynamically updated during the postbuild of TFM_SBSFU_Boot compilation (see Section 10.1 Step 2.1), according to the flash memory layout and to the application configuration. This ensures that the binaries are programmed at the correct flash memory location.

It is important to check that no error is reported during the script execution.

10.4 Configuring STM32U5 static security protections

In development mode (see Section 10.1 Application compilation process), the static security protections are automatically configured in option bytes by the TFM_SBSFU_Boot code at the first start of the application. No further action is expected from the user.

In production mode (see Section 10.1 Application compilation process), the user must first choose the 64-bit RDP level 2 password (OEM2 password). It is automatically provisioned using the STM32CubeProgrammer (STM32CubeProg) CLI command in the regression script. Then, the user must configure the static security protections in the option bytes. The TFM_SBSFU_Boot code only checks the static protections and allows the boot procedure only if the static protections are correctly configured. To make the programming of the static protections easier, an automatic script is available in the STM32CubeU5 MCU Package:

- EWARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\EWARM\hardening.bat`
- MDK-ARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\MDK-ARM\hardening.bat`
- STM32CubeIDE: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\STM32CubeIDE\hardening.sh`

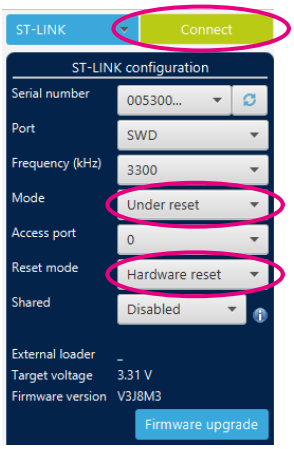
The user must check that no error is reported during the script execution. In case of errors reported, the script must be executed again so that no errors are reported.

This script, relying on the STM32CubeProgrammer CLI, sets the following static protections HDP1, SECWM1, WRP1, SECWM2, WRP2, BOOT_LOCK, and NSBOOTADD0/1 (see Note) according to the flash memory layout and application configuration. The protections setting performed by the hardening script can be verified manually by means of the STM32CubeProgrammer GUI (as described in the steps below). As a second step, the setting of the static protections WRP1A lock (UNLOCK_1A option byte), WRP2A lock (UNLOCK_2A option byte), and RDP must be performed manually (as described in the steps below).

Note: According to the recommendation in section 3 of [RM0456], the nonsecure boot address (NSBOOTADD0 and NSBOOTADD1) must be set in user flash memory.

Step 4.1 - Connection: connect under reset

Figure 35. STM32CubeProgrammer connection menu



DT69311V1

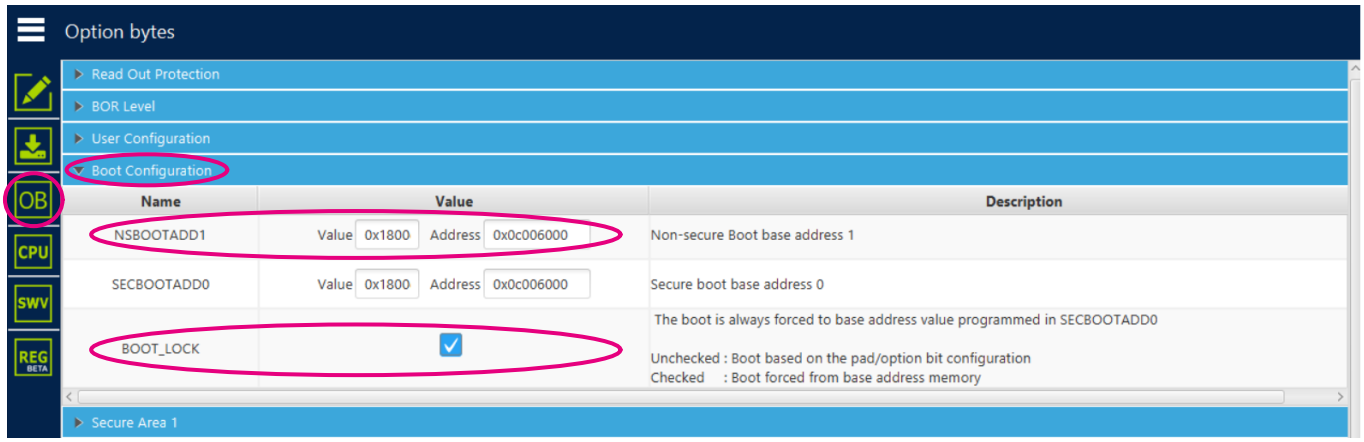
Step 4.2 - Option bytes settings: menu *Option bytes* / *User Configuration*

The following option byte values have been set first by the hardening script, according to flash memory layout and application configuration, the protection settings can be verified manually by means of the STM32CubeProgrammer (STM32CubeProg) GUI:

- HDP1 (hide protect enable)
- WRP1A/WRP2A (write protect)
- SECWM1/SECWM2 (secure flash memory area)
- NSBOOTADD0 (nonsecure boot address 0) = SECBOOTADD0
- NSBOOTADD1 (nonsecure boot address 1) = SECBOOTADD0
- BOOT_LOCK activated (boot entry point fixed to SECBOOTADD0)

The following figures describe, for 2-Mbyte flash memory devices, the option bytes configuration set by the hardening script for default flash memory layout and default application configuration.

Figure 36. STM32CubeProgrammer option bytes screen (*Boot Configuration*)



Note: Once BOOT_LOCK is set, the programmed application must offer the possibility to execute some code in the nonsecure area, to enable the connection to the target and reinitialize the device.

Figure 37. STM32CubeProgrammer option bytes screen (*Secure Area 1*)

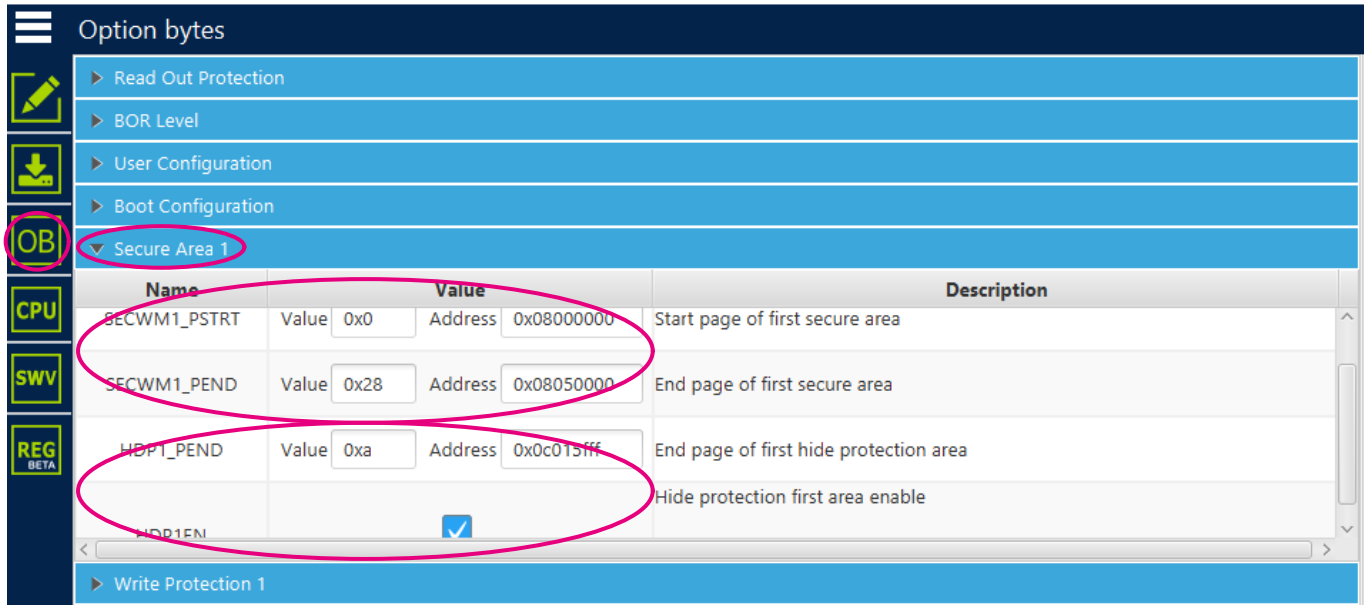


Figure 38. **STM32CubeProgrammer option bytes screen (*Write Protection 1*)**

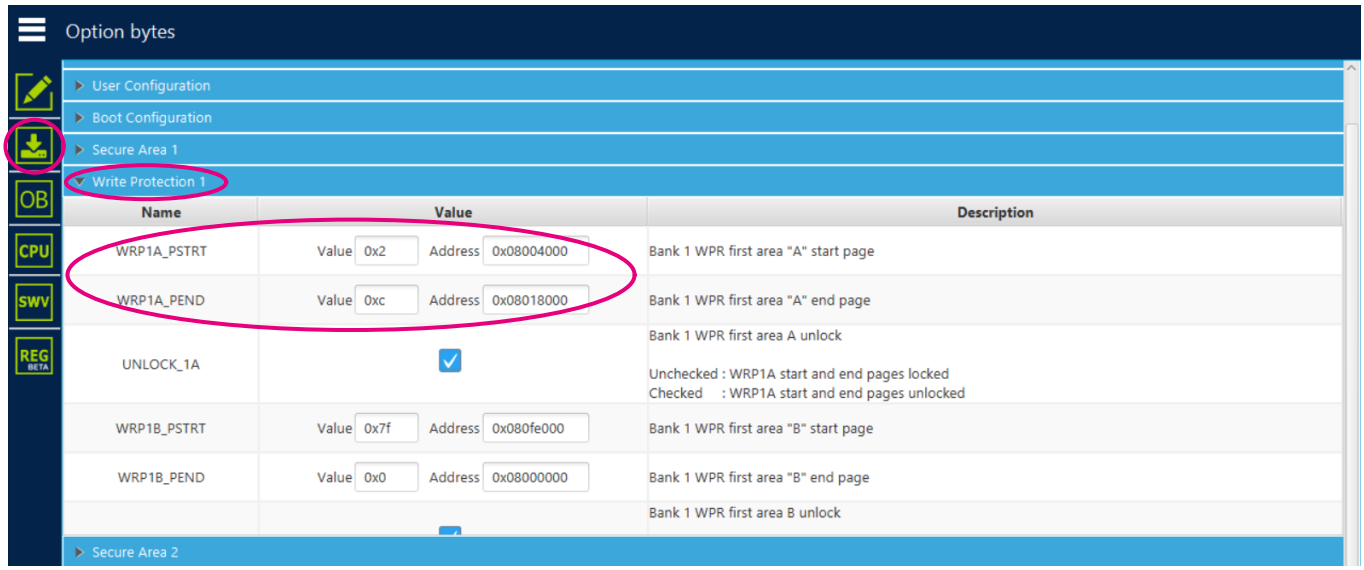


Figure 39. **STM32CubeProgrammer option bytes screen (*Secure Area 2*)**

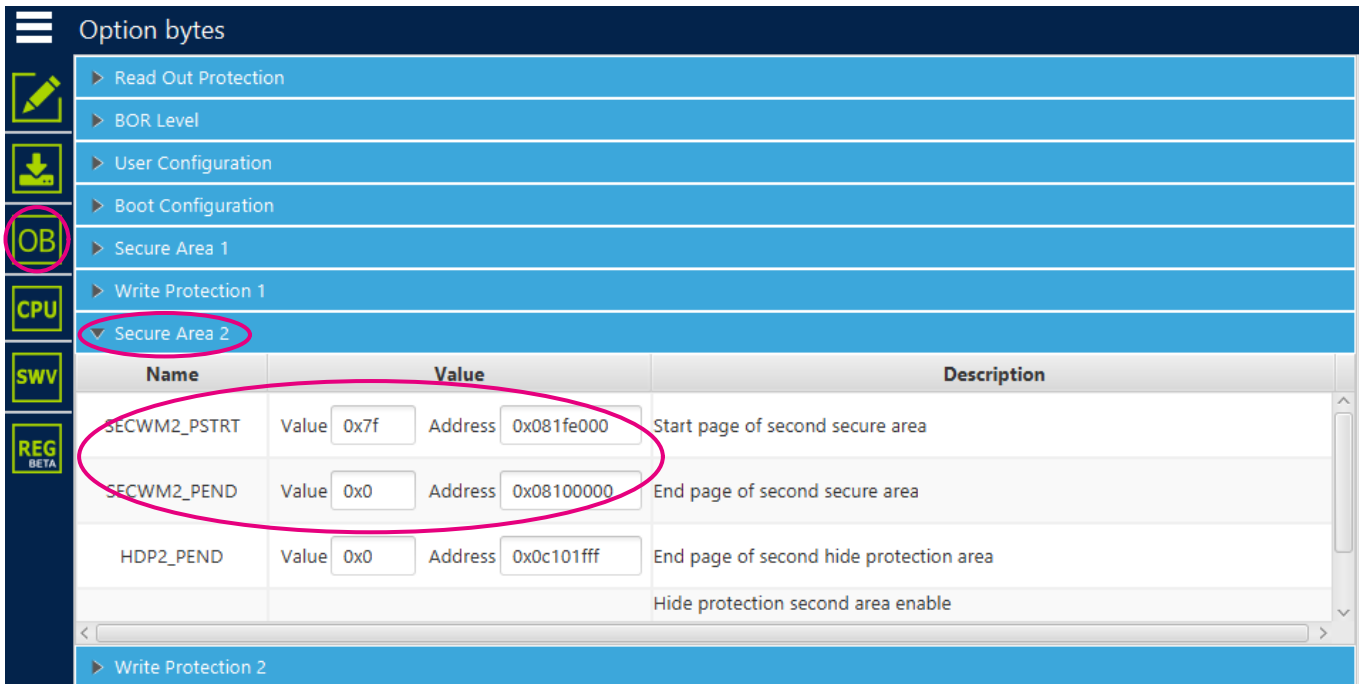
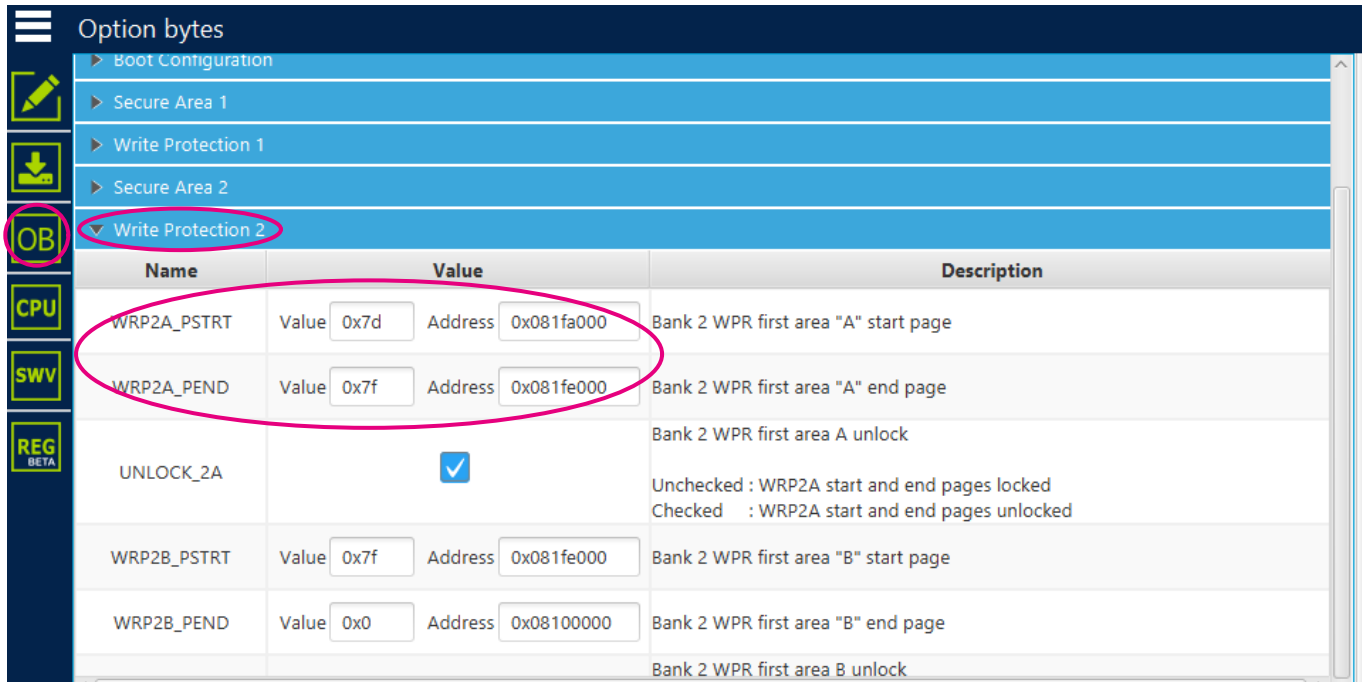


Figure 40. **STM32CubeProgrammer option bytes screen (*Write Protection 2*)**



As a second step, the WRP1A and WRP2A lock must be set manually:

- WRP1A locked (UNLOCK_1A unchecked)
- WRP2A locked (UNLOCK_2A unchecked)

Figure 41. **STM32CubeProgrammer option bytes screen (WRP1A lock)**

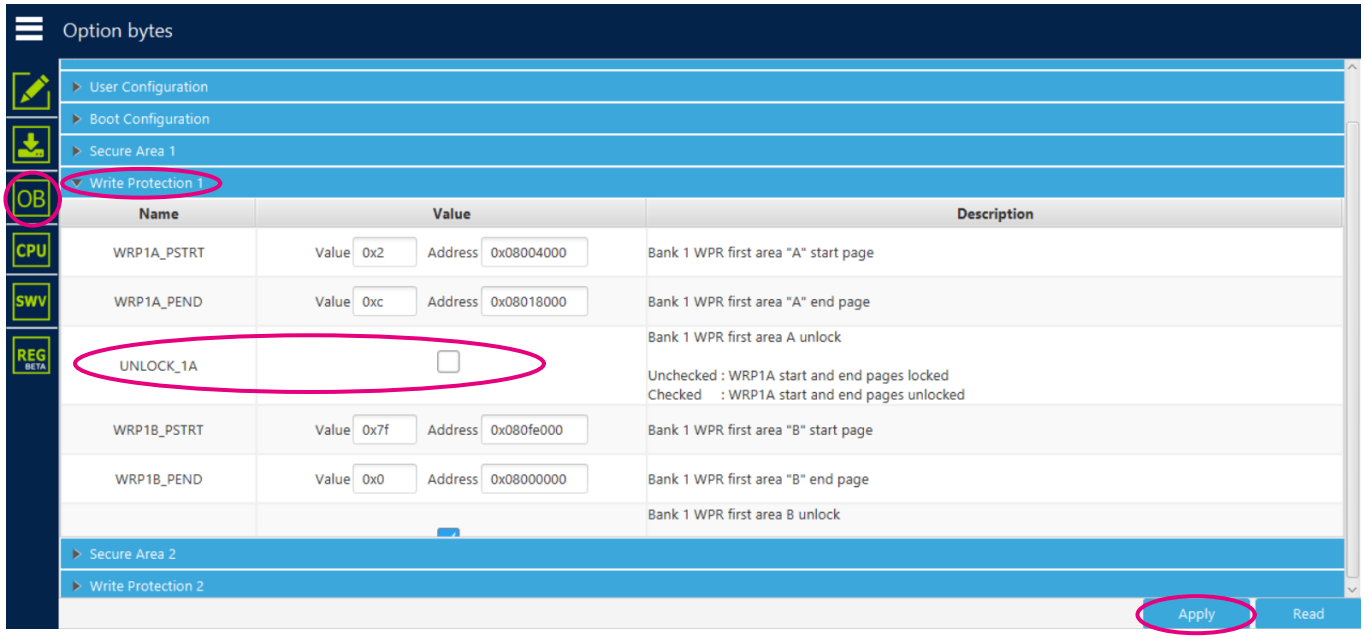
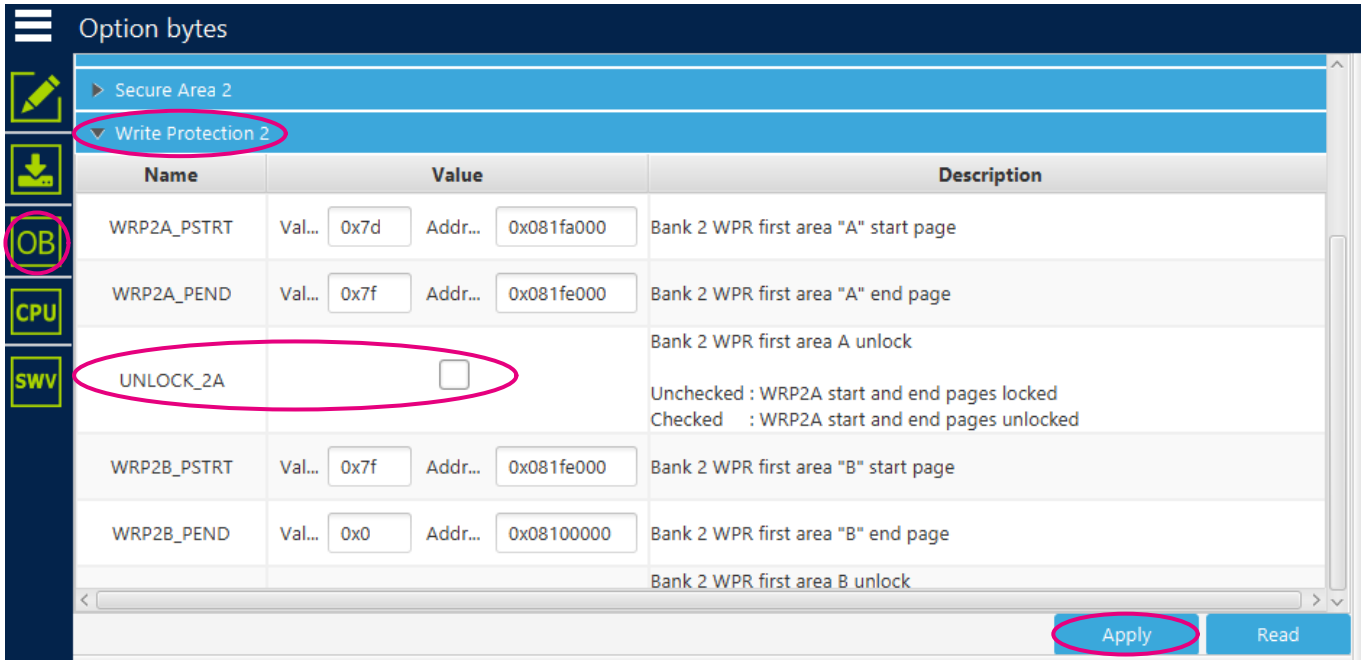


Figure 42. **STM32CubeProgrammer option bytes screen (WRP2A lock)**



And finally, the RDP must be set manually:

- RDP level 2 (JTAG connection only allowed to inject RDP 2 password and obtain device identification)

Figure 43. **STM32CubeProgrammer option bytes screen (RDP)**

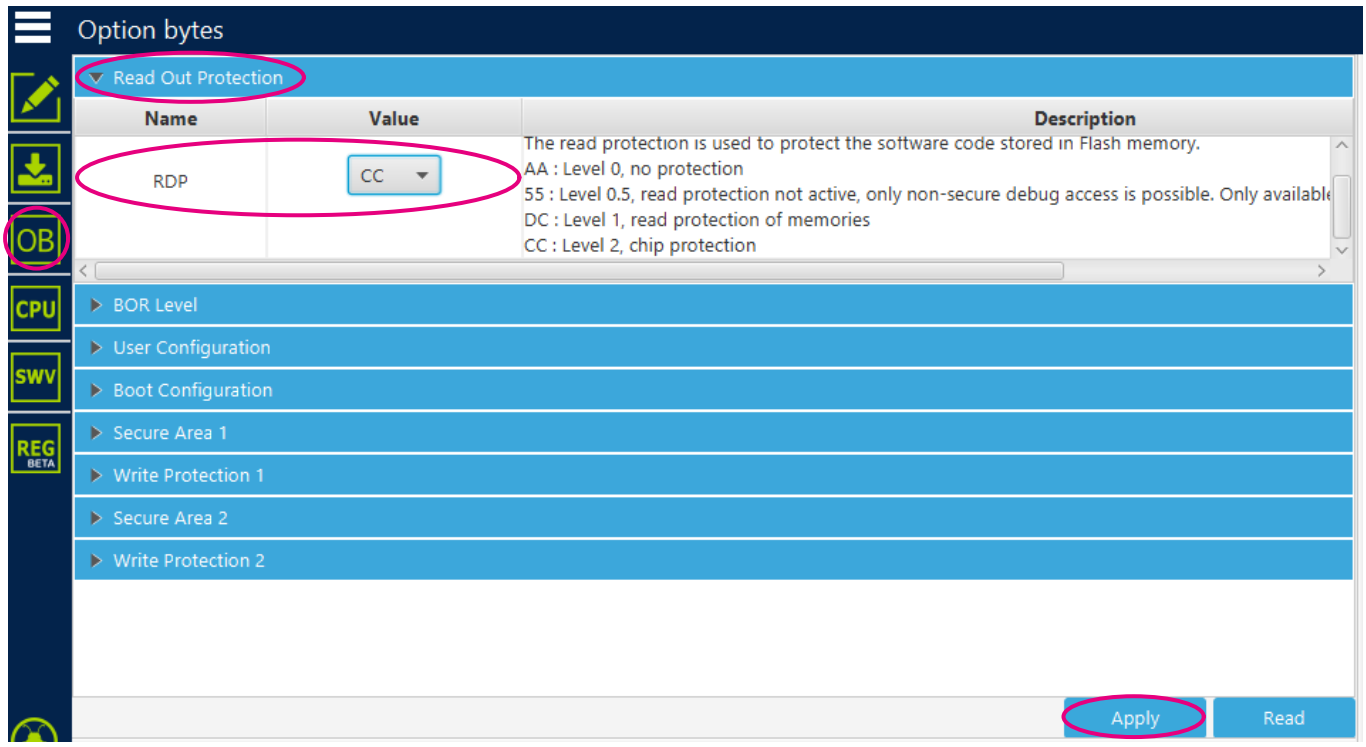
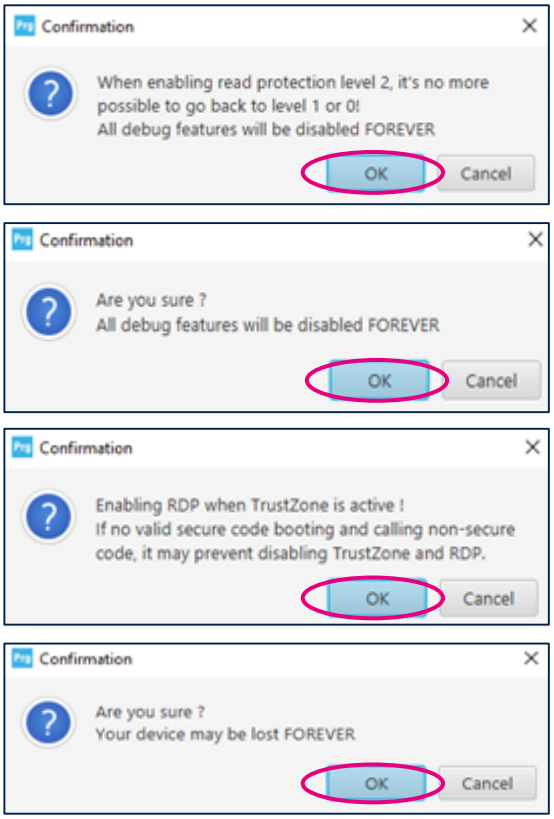


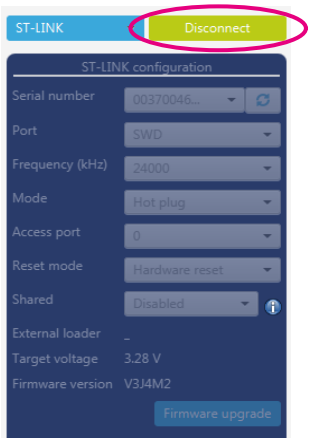
Figure 44. **STM32CubeProgrammer option bytes screen (RDP confirmation)**



DT69320V1

Step 4.3 - Disconnect

Figure 45. **STM32CubeProgrammer disconnect**



DT69308V1

At this step, the device is in freeze state due to the intrusion detection after the RDP level change. As a result, the connection with the device is lost. To recover from the intrusion detection, remove the IDD jumper from the development board, then put it back in place. Refer to Appendix A Development hardware boards for the position of the IDD jumper on the various development boards.

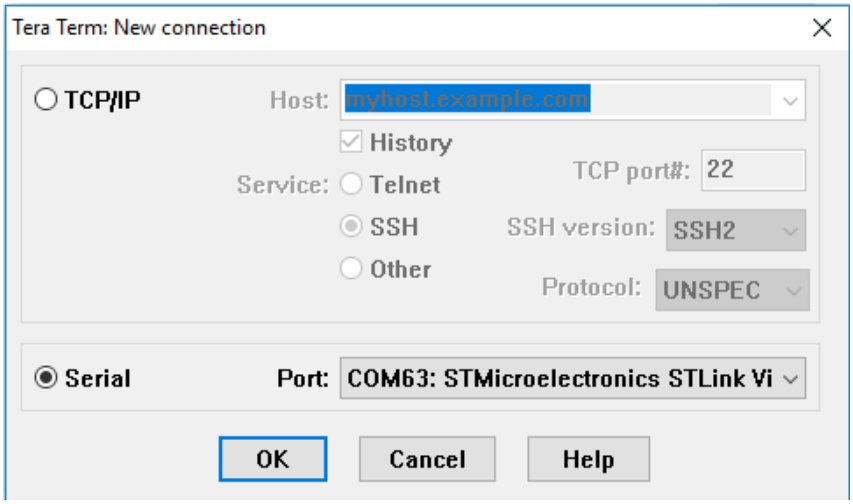
10.5 Tera Term connection preparation procedure

Tera Term connection is achieved by applying in sequence the steps described from Section 10.5.1 to Section 10.5.3.

10.5.1 Tera Term launch

The Tera Term launch requires that the port is selected as *COMxx: STMicroelectronics STLink Virtual COM port*. Figure 46 illustrates an example based on the selection of port COM63.

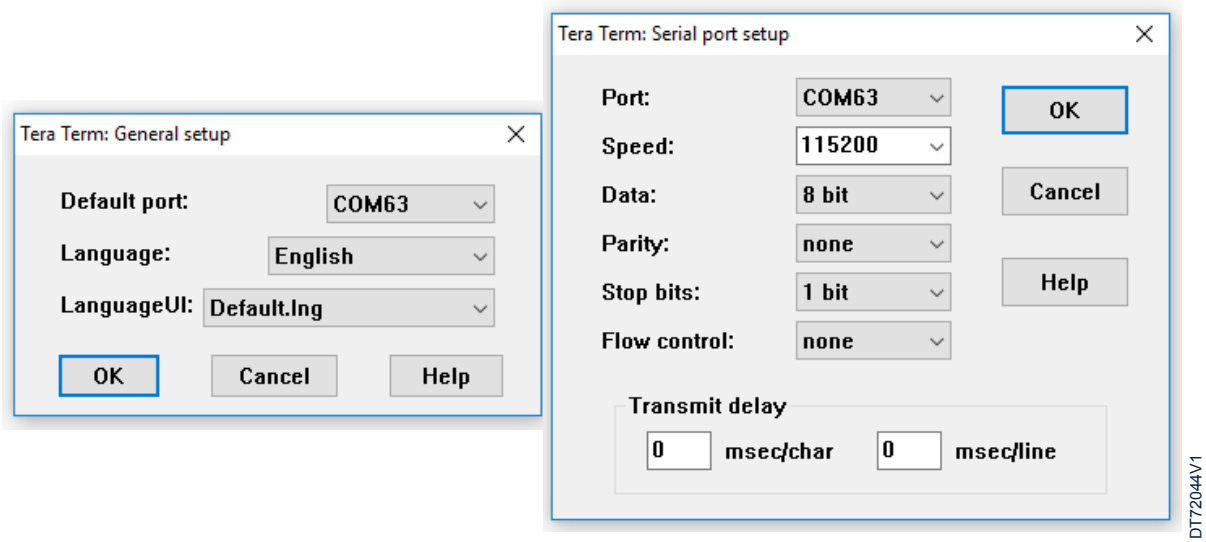
Figure 46. Tera Term connection screen



10.5.2 Tera Term configuration

The Tera Term configuration is performed through the *General* and *Serial port setup* menus. Figure 47. Tera Term setup screens illustrates the *General setup* and *Serial port setup* menus.

Figure 47. Tera Term setup screens



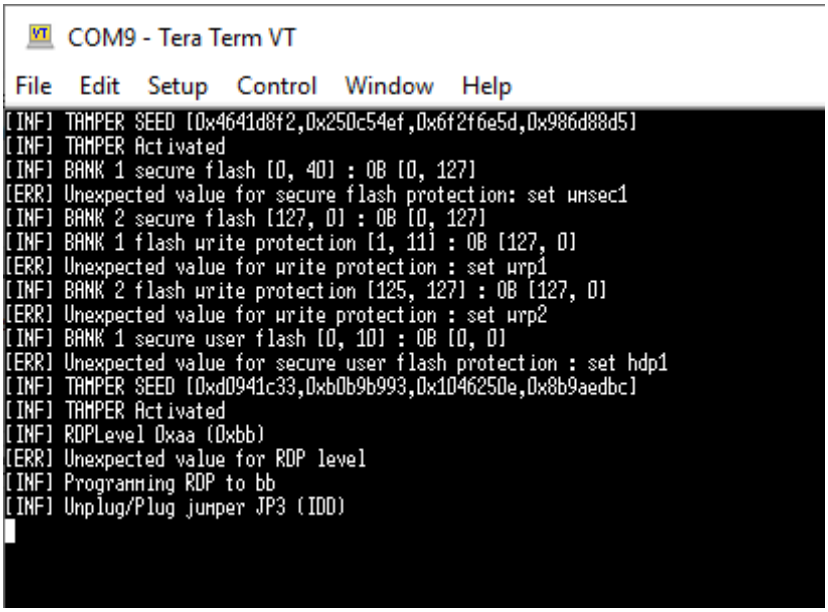
Important: After plugging and unplugging the USB cable, the Tera Term serial port setup menu may have to be validated again to restart the connection. ***Press the [Reset] button to display the welcome screen****.*

10.5.3 ST-LINK disable

The security mechanisms managed by TFM_SBSFU_Boot forbid a 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 V3J8M3 onwards:

- Reset the board after programming binaries (see Section 10.3 Software programming into STM32U5 internal flash memory) by pressing the reset button.
  Refer to Appendix A Development hardware boards for the position of the reset button on the various development boards.
- The TFM_SBSFU_Boot application starts:
  – In the development mode, some information is displayed on the terminal emulator. TFM_SBSFU_Boot configures the security mechanisms the option bytes are not at the correct values. At this time, the microcontroller detects an intrusion due to RDP level 1, so that the execution freezes.

Figure 48. Information example displayed on Tera Term in development mode



  – In the production mode, TFM_SBSFU_Boot logs are disabled so that nothing is visible on the terminal emulator at this step. After the completion of the static protections configuration (see Section 10.4 Configuring STM32U5 static security protections), the microcontroller detects an intrusion due to RDP level 2, so that the execution freezes.
- Remove the IDD jumper from the development board, then put it back in place.
  Refer to Appendix A Development hardware boards for the position of the IDD jumper on the various development boards.

Caution: This step is mandatory, whatever the mode (development/production mode), to recover from intrusion, as soon as RDP is leaving level 0.

- The TFM_SBSFU_Boot application starts with the static protections correctly configured. Then it jumps to the TFM_Appli displaying the user application main menu on the terminal emulator.

Figure 49. Information example displayed on Tera Term in development mode

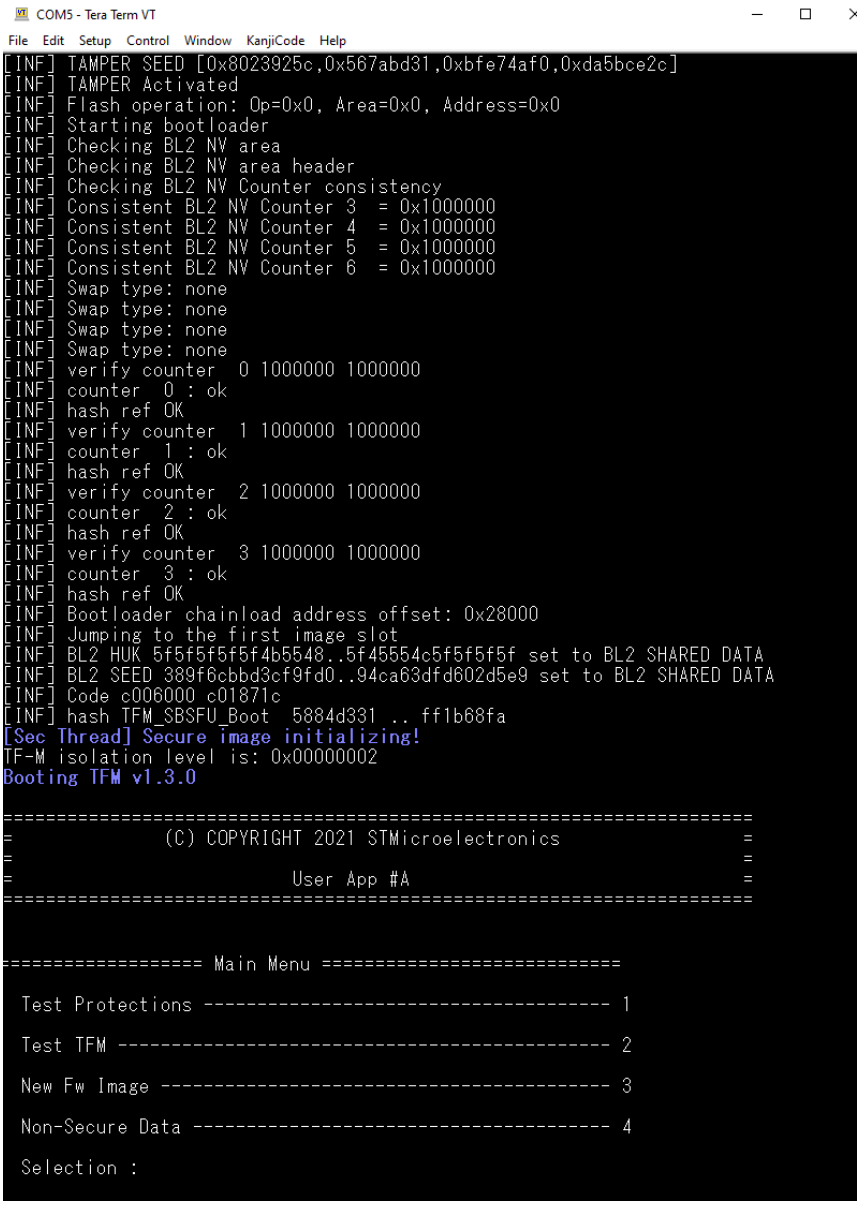
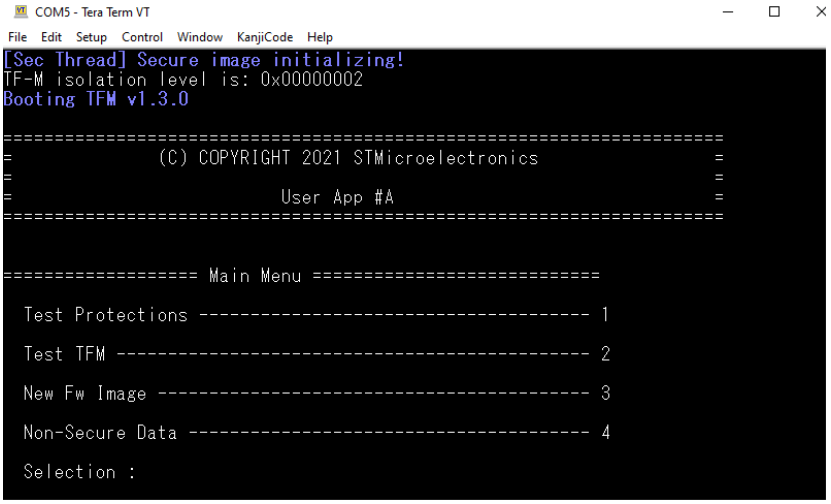


Figure 50. Display on Tera Term in production mode



10.6 STM32U5 device reinitialization

Once the TFM application is running on the device (in development mode or in production mode), it is possible to reinitialize the device to install a new TFM application.

- In development mode (RDP level 1), the device reinitialization can be achieved by running the regression script previously described in Section 10.2 STM32U5 device initialization. The RDP then switches to level 0 with a flash memory mass erase.
- In production mode (RDP level 2 with the OEM2 password provisioned) the device reinitialization can be achieved with these steps:
  1. Inject the OEM2 password, to switch to RDP level 1.
     For the OEM2 password value example 0xFACEB00C 0xDEADBABE, the command is: `./STM32_Programmer_CLI -c port=SWD mode=UR --hardRst -unlockRDP2 0xFACEB00C 0xDEADBABE`
  2. Remove the IDD jumper, then put it back in place to recover from intrusion. Refer to Appendix A Development hardware boards for the position of the IDD jumper.
  3. Run the regression script previously described in the section Section 10.2 STM32U5 device initialization. The RDP then switches to level 0 with a flash memory mass erase. If errors are reported during the script execution, the script must be executed again so that no errors are reported.

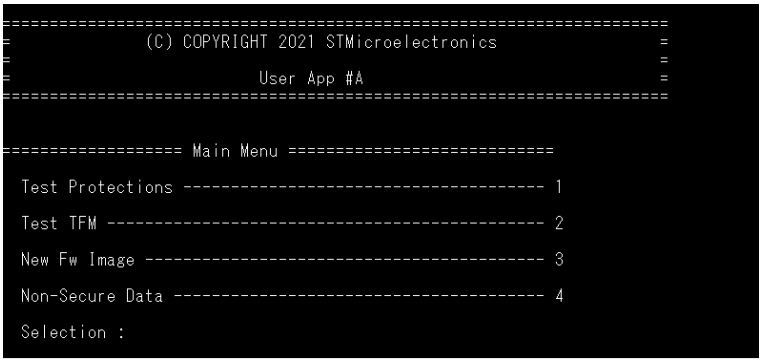
At this stage, the device is reinitialized (with TrustZone® option bit enabled), and the TFM installation procedure can be performed again (starting from Section 10.1 Application compilation process).

11 Step-by-step execution

11.1 Welcome screen display

After the installation procedure, the welcome screen of the TFM nonsecure application is displayed on Tera Term:

Figure 51. TFM nonsecure application welcome screen

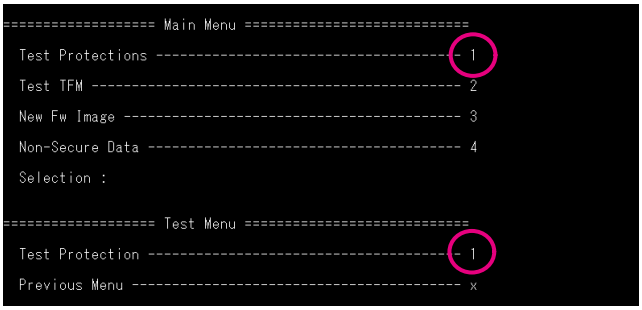


11.2 Test protections

By pressing ‘1’, the user enters the test protection menu.

Press then ‘1’ to trigger protected areas (flash memory, SRAM, peripherals) accesses attempts from nonsecure, and secure unprivileged code and DMA.

Figure 52. Test protection menu

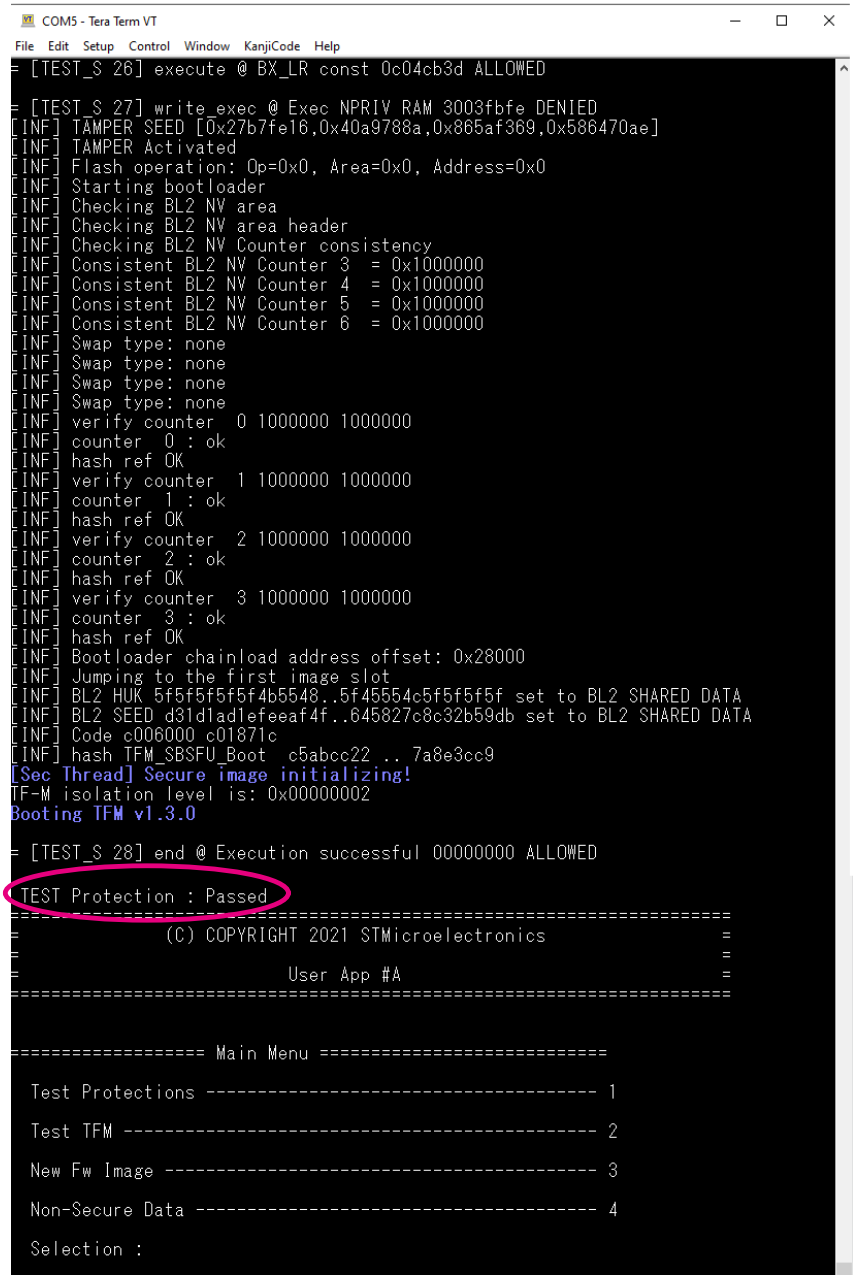


Several access attempts are performed in a row. For each access attempt, the effective behavior is checked against the expected behavior. The result can be one of the following ones:

- DENIED (provoking reset)
- SILENT (read as zero, write no effect)
- ALLOWED

At the end of the sequence, the global test status ‘Passed’ is displayed if all executed protection tests have succeeded.

Figure 53. Test protection results



11.3 Test TFM

By pressing ‘2’, the user enters the TFM test menu.

Figure 54. TFM test menu

This menu permits the test of some of the TF-M secure services at runtime.
The user can select the TFM test to run by pressing the corresponding key:

- ‘1’: Test AES-GCM crypto services
- ‘2’: Test AES-CBC crypto services
- ‘3’: Test AES-CCM crypto services.
- ‘4’: Test UID creation in protected storage area
- ‘5’: Test UID read and check in protected storage area
- ‘6’: Test UID removal in protected storage area
- ‘7’: Test initial attestation service
- ‘8’: Test UID creation in internal trusted storage area
- ‘9’: Test UID read and check in internal trusted storage area
- ‘a’: Test UID removal in internal trusted storage area
- ‘b’: Test SHA224 crypto services
- ‘c’: Test SHA256 crypto services
- ‘d’: Test Persistent key import services
- ‘e’: Test Persistent key export services

- ‘f’: Test Persistent key destruction services

Additionally, the following menu item is not activated by default:

- ‘s’: Test STSAFE (refer to Section 12.1 Configuration for its activation)

When pressing ‘0’, all TFM test examples are executed in a row and the overall result is displayed in the log.

Figure 55. TFM test results

Additional information regarding the EAT service token response

The entity token is CBOR encoded. It is possible to decode it by following these steps:

1. Ensure that a Python™ version is installed as indicated in Section 9.2.5 Python™
2. Copy and paste the token response obtained in the terminal emulator into the text file `Middlewares\Third_Party\trustedfirmware\tools\iat-verifier\st_tools\eat.txt`
3. Navigate to `Middlewares\Third_Party\trustedfirmware\tools\iat-verifier` and execute the installation of the required packages: `python setup.py install`
4. Decode the token response, from `Middlewares\Third_Party\trustedfirmware\tools\iat-verifier\st_tools`
   - With the private key
     a. Encode the EAT into the CBOR format: `python build.py cbor ./eat.txt ./eat.cbor`
     b. Decode the EAT: `check_iat -k ../../../../../../Projects/B-U585I-IOT02A/Applications/TFM/TFM_SBSFU_Boot/Src/tfm_initial_attestation_key.pem ./eat.cbor -p`
   - With the public key (case of STSAFE)
     a. Encode the EAT into the CBOR format: `python build.py cbor ./eat.txt ./eat.cbor`
     b. Copy and paste the certificate obtained in the terminal emulator (Test STSAFE menu) into the text file `certificate.txt`
     c. Encode the certificate into a (PEM) Base64 encoded file with distinct headers and footers. Use OpenSSL:
     `cat certificate.txt | xxd -r -p | openssl.exe x509 -inform DER -out certificate.pem -outform PEM`
     d. Retrieve the public key from the certificate:
     `openssl x509 -pubkey -noout -in certificate.pem >> pubkey.pem`
     e. Decode the EAT: `check_iat -k pubkey.pem ./eat.cbor -p`

The example below shows the decoding result of an EAT token response:

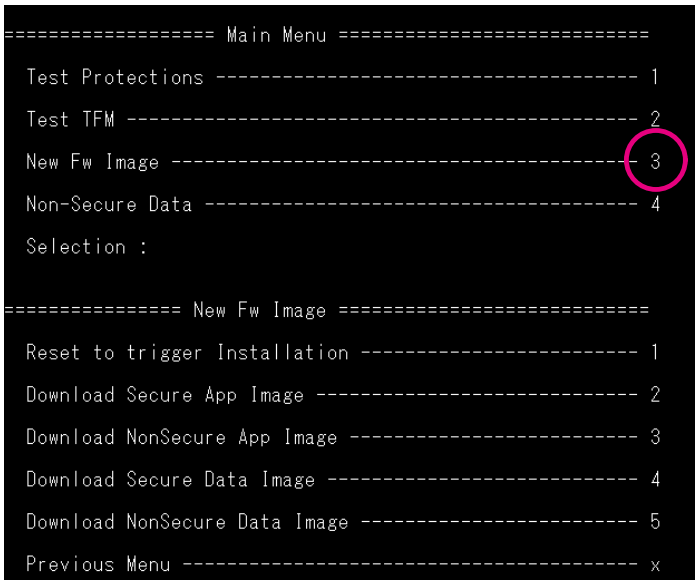
```
Signature OK
Token format OK
Token:
{
    "CHALLENGE":
"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000",
    "BOOT_SEED":
"5350bd5adf2246508f1ec3dd08e755ca78a2410825bdd0f52de9d12bf041d26b",
    "INSTANCE_ID":
"01fa58755f658627ce5460f29b75296713248cae7ad9e2984b90280efcbcb50248",
    "IMPLEMENTATION_ID":
"14802125f37bb43958d97435ddfcea8421c22d92ebac15a270a17c3130fd6bc3",
    "CLIENT_ID": -1,
    "SECURITY_LIFECYCLE": "SL_SECURED",
    "SW_COMPONENTS": [
        {
            "SW_COMPONENT_TYPE": "SPE",
            "SW_COMPONENT_VERSION": "1.0.0",
            "MEASUREMENT_VALUE":
"b0b145240ead6765f9ee80b1bb241d9ad3d8deaf3c812ab904dd358beddfadcf",
            "MEASUREMENT_DESCRIPTION": "SHA256",
            "SIGNER_ID":
"fc5701dc6135e1323847bdc40f04d2e5bee5833b23c29f93593d00018cfa9994"
        },
        {
            "SW_COMPONENT_TYPE": "NSPE",
            "SW_COMPONENT_VERSION": "1.0.0",
            "MEASUREMENT_VALUE":
"adbd515b6814de8a2e2bf3aed29abde50b10e201844e8d7e86be519a52f28fb2",
            "MEASUREMENT_DESCRIPTION": "SHA256",
            "SIGNER_ID":
"e18015993d6d2760b499274baef264b83af229e9a785f3d5bf00b9d32c1f0396"
        }
    ],
    "HARDWARE_ID": "04080202000000"
}
```

11.4 New firmware image

11.4.1 New firmware image in overwrite mode configuration (default configuration)

By pressing ‘3’, the user enters the new firmware image menu.

Figure 56. New firmware image menu



It is possible to download a new TFM secure application image, a new TFM nonsecure application image, or both.

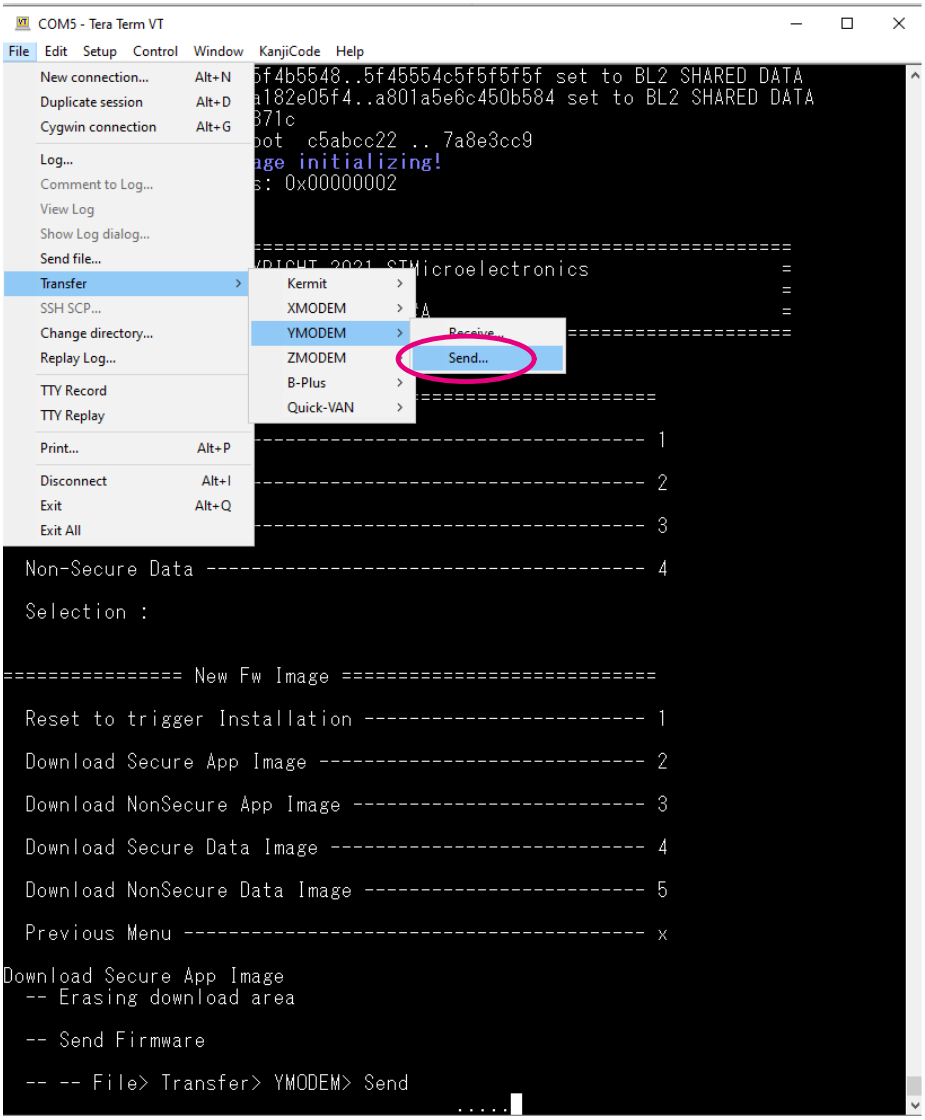
- Press ‘2’ to download a secure signed application image
  - either the encrypted secure signed image `TFM_Appli\Binary\tfm_s_app_enc_sign.bin`
  - or the clear secure signed image `TFM_Appli\Binary\tfm_s_app_sign.bin`
- Press ‘3’ to download a nonsecure signed application image
  - either the encrypted nonsecure signed image `TFM_Appli\Binary\tfm_ns_app_enc_sign.bin`
  - or the clear nonsecure signed image `TFM_Appli\Binary\tfm_ns_app_sign.bin`

It is also possible to download a new TFM secure data image, a new TFM nonsecure data image, or both.

- Press ‘4’ to download a secure signed data image
  - either the encrypted secure signed image `TFM_Appli\Binary\tfm_s_data_enc_sign.bin`
  - or the clear secure signed image `TFM_Appli\Binary\tfm_s_data_sign.bin`
- Press ‘5’ to download a nonsecure signed data image
  - either the encrypted nonsecure signed image `TFM_Appli\Binary\tfm_ns_data_enc_sign.bin`
  - or the clear nonsecure signed image `TFM_Appli\Binary\tfm_ns_data_sign.bin`

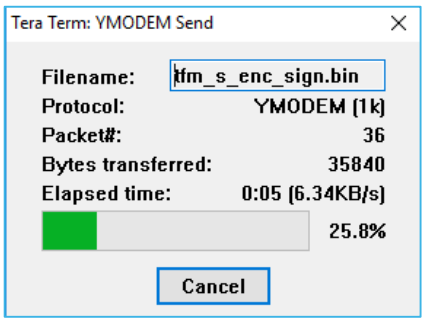
In all four cases, send the signed binary with Tera Term by using the menu [**File**]>[**Transfer**]>[**YMODEM**]>[**Send...**].

Figure 57. Firmware image transfer start



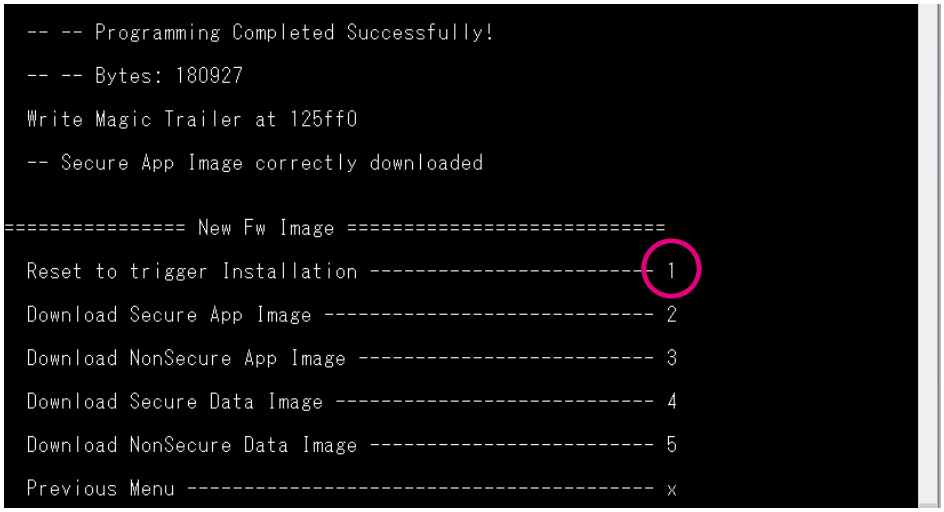
Once the file is selected, the Ymodem transfer starts. The transfer progress is reported as shown in Figure 58.

Figure 58. Firmware image transfer in progress



After the download, press ‘1’ to reset the board and trigger the installation (or press the board reset button), as shown in Figure 59.

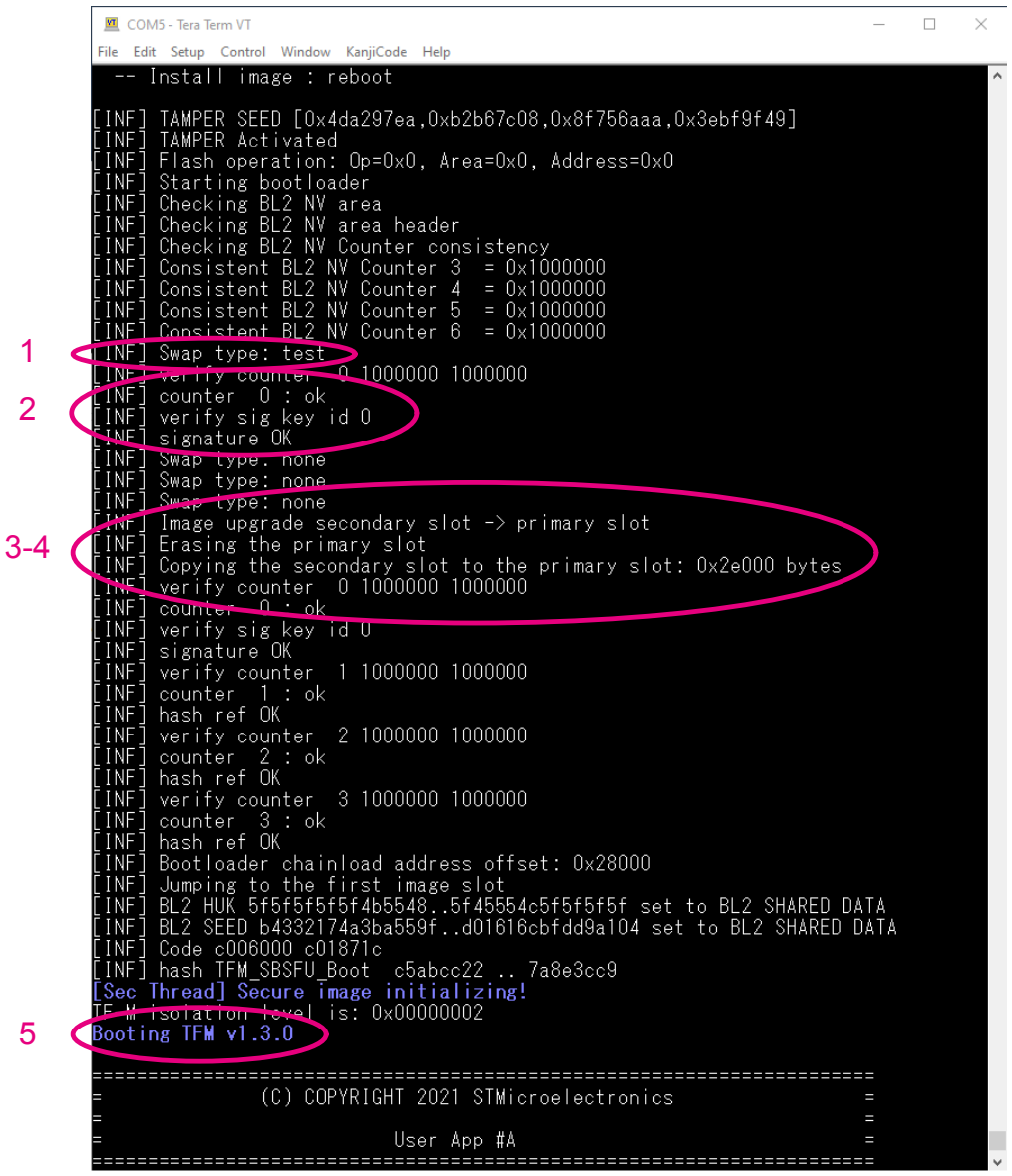
Figure 59. Reset to trigger installation



After reset, the downloaded firmware images (one or two) are

1. detected
2. verified (including version antirollback check)
3. decrypted (if needed)
4. installed
5. executed by TFM_SBSFU_Boot

Figure 60. Image installation (in overwrite mode)



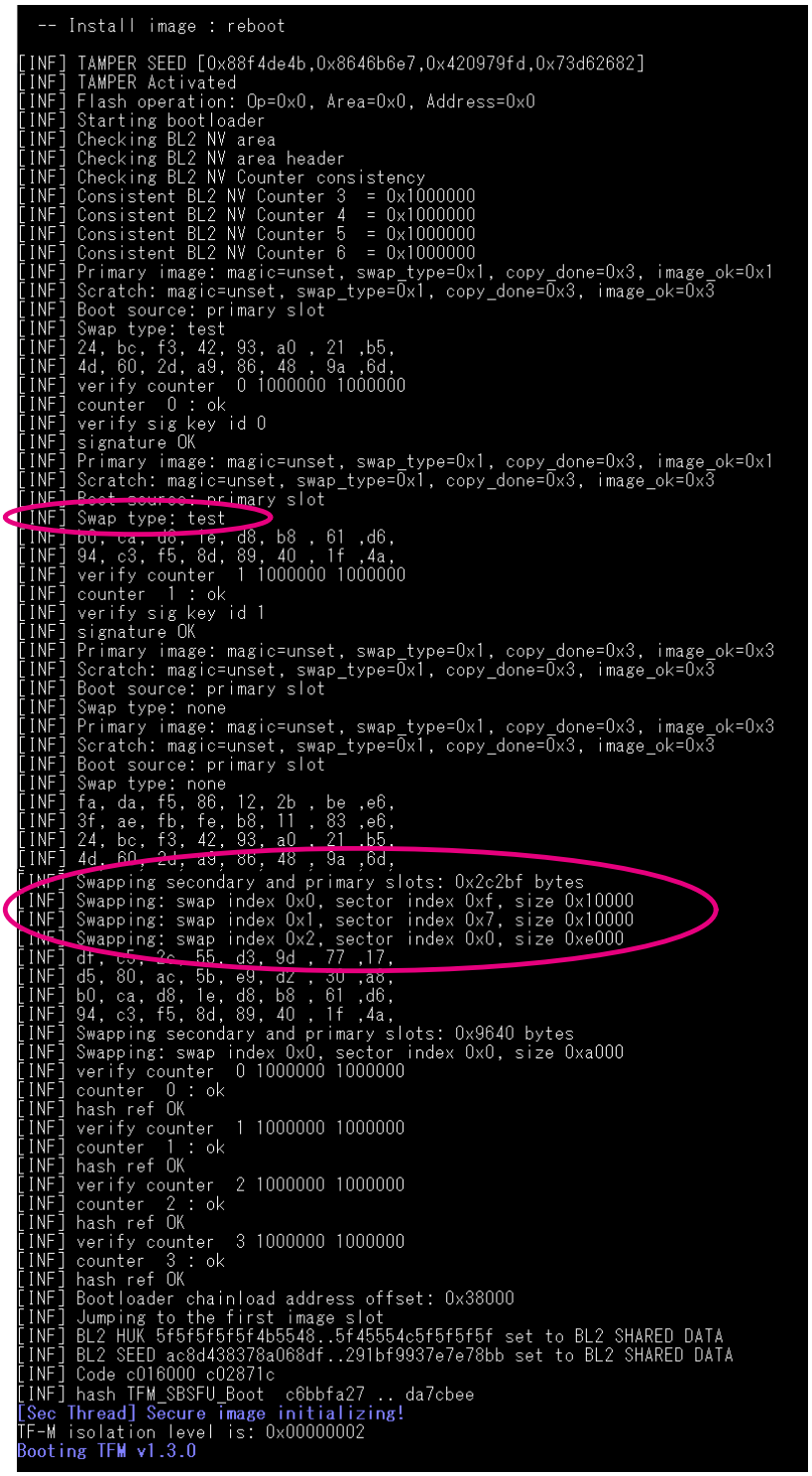
In the case of a one-image configuration (and not the two-image configuration by default), the procedure is similar with the exception that the new firmware image menu proposes to download a unique image instead of secure and nonsecure images.

In the case of primary slot only configuration (and not primary and secondary slots configuration by default), the new firmware image menu is not available. This is because the image under execution cannot download a new image at the same address. In this case, the local loader (refer to Section 11.6 Local loader) must be used to download a new image.

11.4.2 New firmware image in swap mode configuration

In the swap mode configuration, the procedure to install a new image is similar to the one described in the overwrite mode configuration (refer to Section 11.4.1 New firmware image in overwrite mode configuration (default configuration)) while the image installation logs differ.

Figure 61. Image installation (in swap mode)



In this the swap mode configuration, once installed, the new image must be validated at the first image boot; otherwise, it is reverted at the next boot. To validate the image after the installation, the user must press ‘3’ to enter the new firmware image menu in the user application, then press ‘6’ or ‘7’.

Figure 62. New firmware image menu (swap mode)

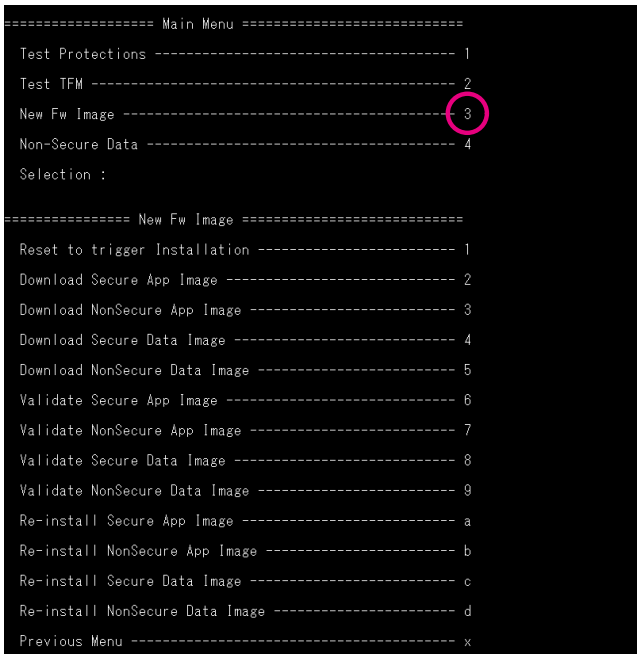
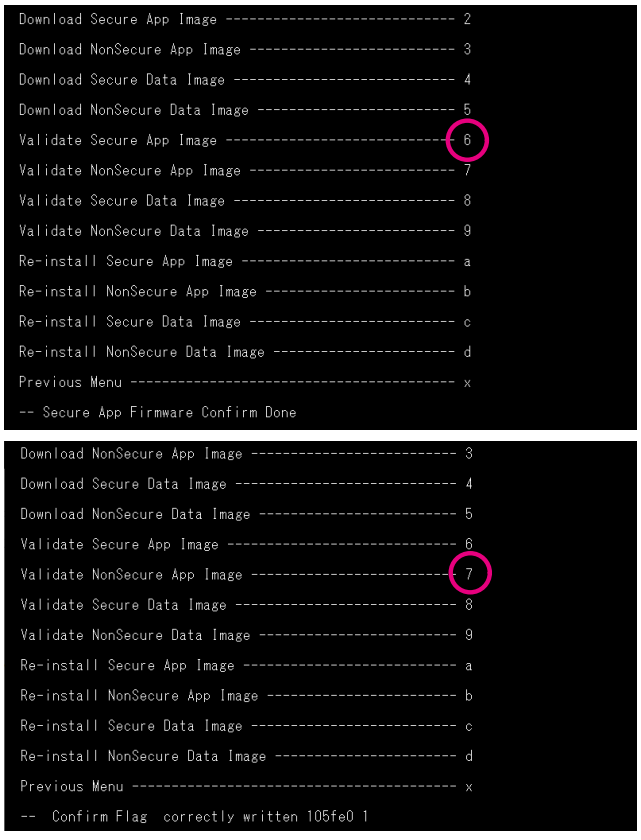
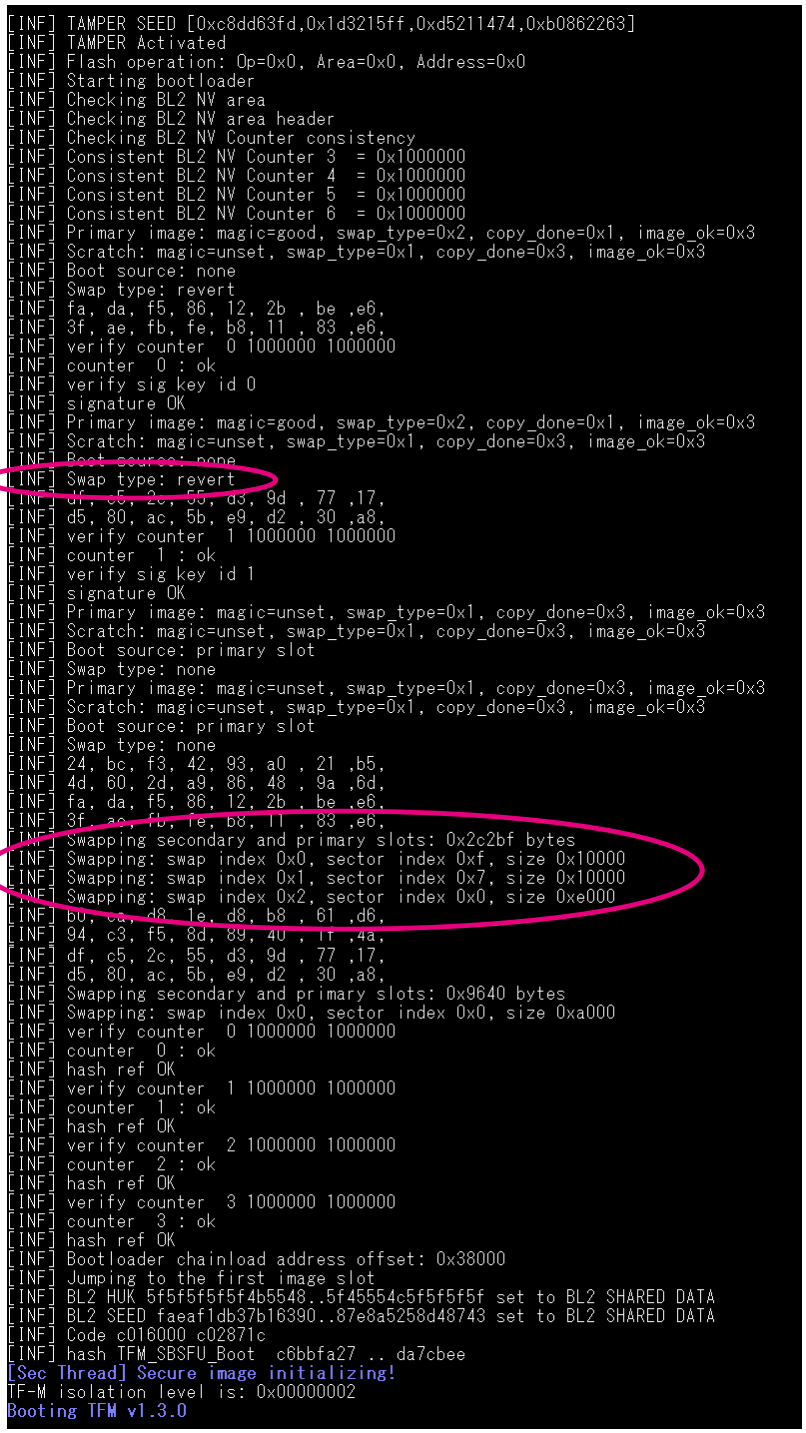


Figure 63. Validate secure or nonsecure image



If the new image is not validated, it is then reverted at the next boot.

Figure 64. Image reverted if not validated

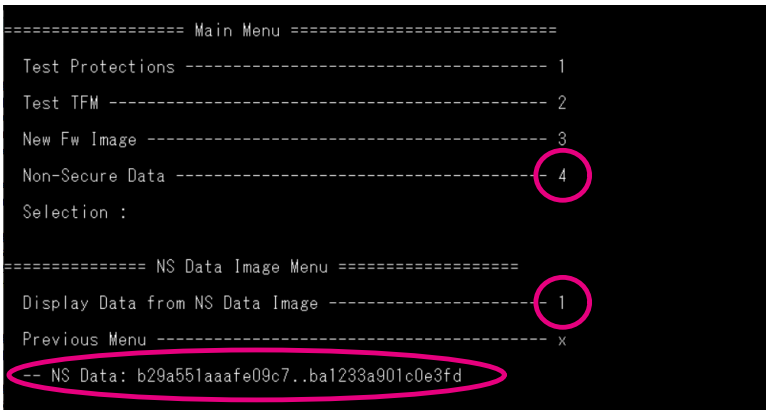


The new firmware image menu offers also the possibility to request the reinstallation of a reverted image. Press ‘a’ or ‘b’ in the new firmware image menu to request the reinstallation of a reverted secure or nonsecure image. Image reinstallation is then performed at the next reset.

11.5 Nonsecure data

By pressing ‘4’, the user enters the nonsecure data menu. When pressing ‘1’, the primary nonsecure data image content is read and displayed.

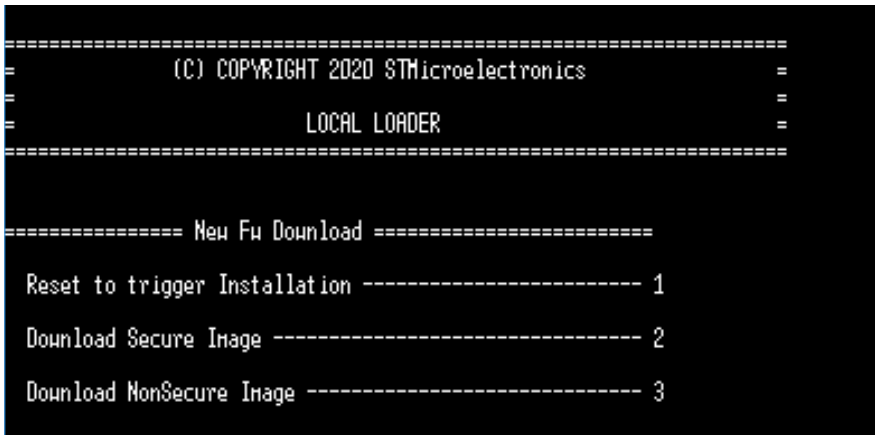
Figure 65. Nonsecure data menu



11.6 Local loader

By pressing the user button (blue) during board reset, the user enters the local loader menu. The local loader is not part of the TFM nonsecure application, but is an immutable standalone application, in the nonsecure area.

Figure 66. TFM local loader application welcome screen



This local loader allows the download of a new TFM secure image, a new TFM nonsecure image, or both, exactly in the same way as the new firmware image menu of the TFM application (refer to Section 11.4.1 New firmware image in overwrite mode configuration (default configuration)).

12 Integrator role description

STMicroelectronics delivers to customers (also called integrators or OEMs) a complete ecosystem around the STM32U5 microcontroller:

- STM32U5 device: delivered with virgin user flash memory and security not activated in option byte
- Set of reference boards (Nucleo board, Discovery kit, and Evaluation board)
- STM32CubeU5 MCU Package containing: STM32U5 HAL drivers, BSPs for the supported references boards, projects examples (including the certified PSA L3 TFM application example) for three IDEs (IAR Systems® IAR Embedded Workbench®, Keil® MDK-ARM, and STMicroelectronics STM32CubeIDE).
- Tools: STM32CubeProgrammer (STM32CubeProg) for option bytes and flash memory programming, STM32CubeMX for the configuration of STM32 microcontrollers and generation of initialization code, STM32CubeIDE free-of-charge IDE for building, downloading, and debugging applications.

Integrators start to develop their products from the STM32U5 ecosystem delivered by STMicroelectronics. They are responsible for personalizing the product data and for configuring the product security following the guidelines provided by STMicroelectronics:

- Develop product mechanics
- Develop own board (based on STM32U5 device)
- Develop own product software application (at least own nonsecure applications)
- Integrate product software application onto the board
- Prepare STM32U5 device:
  - STM32U5 user flash memory programming (TFM_SBSFU_Boot application binary, TFM_Appli secure, and nonsecure binaries, TFM_Loader binaries)
  - STM32U5 TFM product personalization (such as integrator personalized parameter)
  - STM32U5 device security configuration
- Product manufacture (hardware board and product mechanics)
- Product maintenance when deployed in the field (update the nonsecure application, update the updatable part of the secure application, or both)

The integrator has full access to the source code delivered in the STM32CubeU5 MCU Package and full access to the security features of the STM32U5 device integrated on the board. STMicroelectronics delivers this device in virgin state without any security feature activated.

The integrator is responsible for the security of the product starting from the PSA L3 certified STM32U5 platform. The integrator may want to reuse as much as possible the PSA L3 certified STM32U5 platform from STMicroelectronics in order to make the product certification easier and faster. Nevertheless, at least some parts must be customized or changed by the integrator as described in Section 12.1 Configuration, Section 12.2 Minimal customization, Section 12.3 Other customization, and Section 12.4 Production.

12.1 Configuration

The integrator must first select the configuration of the application by activating the different compiler switches described below.

Crypto scheme

In the TFM and SBSFU applications, by default, the crypto scheme features an RSA-2048 signature, and an AES-CTR-128 image encryption with the key RSA-OAEP encrypted. This crypto scheme provides a good trade-off between boot time performance and security level. It is possible to select another crypto scheme by means of the `CRYPTO_SCHEME` define in `TFM_SBSFU_Boot\Inc\mcuboot_config\mcuboot_config.h`.

```
#define CRYPTO_SCHEME_RSA2048    0x0 /* RSA-2048 signature,
                                        AES-CTR-128 encryption with key RSA-OAEP encrypted */
#define CRYPTO_SCHEME_RSA3072    0x1 /* RSA-3072 signature,
                                        AES-CTR-128 encryption with key RSA-OAEP encrypted */
#define CRYPTO_SCHEME_EC256      0x2 /* ECDSA-256 signature,
                                        AES-CTR-128 encryption with key ECIES-P256 encrypted */

#define CRYPTO_SCHEME            CRYPTO_SCHEME_RSA2048  /* Select one of available crypto schemes */
```

Hardware-accelerated cryptography

By default, the cryptography operations in the TFM and SBSFU applications are performed through the hardware cryptography peripherals of the device (PKA, SAES, HASH). The hardware-accelerated cryptography improves performance and is resistant against side channel attacks.

It is possible to disable the hardware acceleration in MCUboot by commenting the `BL2_HW_ACCEL_ENABLE` define in `TFM_SBSFU_Boot\Inc\config-boot.h`.

```
/* HW accelerators activation in BL2 */
#define BL2_HW_ACCEL_ENABLE
```

The hardware acceleration in the TFM crypto service also enables the usage of the hardware secret nonvolatile unique key of the STM32U5 device as HUK for AES-GCM based AEAD encryption in the PS secure service. It is possible to disable hardware acceleration in the TFM crypto service by commenting the `TFM_HW_ACCEL_ENABLE` define in `TFM_Appli\Secure\Inc\tfm_mbedcrypto_config.h`. In this case, the HUK for AES-GCM based AEAD encryption in the PS secure service relies on provisioned HUK in the integrator's personalized area.

```
/* HW accelerators activation in TFM */
#define TFM_HW_ACCEL_ENABLE
```

When the hardware-accelerated cryptography is disabled, the TFM and SBSFU examples can be run on the boards with microcontrollers without an encryption accelerator engine (AES, PKA, and OTFDEC).

Encryption

In the TFM and SBSFU applications, by default, the image encryption support is enabled, so that the firmware image can be provided either in clear format or in AES-CTR-128 encrypted format. It is possible to disable the image encryption support to reduce the flash memory footprint using the `MCUBOOT_ENC_IMAGES` define in `TFM_SBSFU_Boot\Inc\mcuboot_config\mcuboot_config.h`.

```
#define MCUBOOT_ENC_IMAGES   /* Defined: Image encryption enabled. */
```

Local loader

A Ymodem local loader example application is included by default in the TFM and SBSFU applications. It is possible to remove it using the `MCUBOOT_EXT_LOADER` define in `Linker\flash_layout.h`.

```
#define MCUBOOT_EXT_LOADER         /* Defined: Add external local loader application.
                                              To enter it, press user button at reset.
                                     Undefined: No external local loader application. */
```

Number of application images

The TFM application example is statically defined to manage two application images so that the images are smaller, and the secure image and the nonsecure image can be managed by two distinct entities. However, it is possible to manage a single application image so that the boot time is reduced by adapting the example.

The number of images is one by default in the SBSFU application (one image for both NS and S binaries, with one single signature). It is possible to separate the NS and S binaries into two images with two distinct signatures using the `MCUBOOT_IMAGE_NUMBER` define in `Linker\flash_layout.h`.

```
#define MCUBOOT_IMAGE_NUMBER 1    /* 1: S and NS application binaries are assembled in one single image.
                                     2: Two separated images for S and NS application binaries. */
```

The number of images is two by default in the TFM_Appli application (one nonsecure image and one secure image, each with its own signature).

```
#define MCUBOOT_APP_IMAGE_NUMBER 2 /* 1: S and NS application binaries are assembled in one single imag
e.
                                      2: Two separated images for S and NS application binaries. */
```

Number of data images

The TFM application example can be provisioned with data images. The installation and update mechanisms are like for the application images. No image, one data image (secure or nonsecure), or two data images (secure and nonsecure) can be defined in `Linker\flash_layout.h`. By default, one secure data image and one nonsecure data image are managed.

```
#define MCUBOOT_S_DATA_IMAGE_NUMBER 1   /* 1: S data image for S application.
                                           0: No S data image. */

#define MCUBOOT_NS_DATA_IMAGE_NUMBER 1  /* 1: NS data image for NS application.
                                           0: No NS data image. */
```

Provisioned HUK

The TFM_SBSFU_Boot application places a provisioned HUK in the shared data area so that TFM_Appli can execute software cryptography for protected storage. By default, no provisioned HUK (SW key) is included in the secure shared data because PS relies on a hardware secret nonvolatile unique key. Enabling this option is specifically mandatory on a target where the hardware-accelerated cryptography is not supported.

By commenting this switch, no provisioned HUK is included in the secure shared data.

```
//#define BL2_USE_HUK_HW /* Use hardware device HUK, else use software HUK provisioned in Perso area, f
or TFM PS */
```

PSA_USE_SE_ST

By default, STSAFE is not enabled in TFM_Appli. It is possible to define it by adding the `PSA_USE_SE_ST` compile switch in the TFM_Appli secure project (to which STSAFE is connected) and in the TFM_Appli nonsecure project (to which STSAFE is addressed through the PSA APIs).

When this option is activated in the TFM_Appli secure project, STSAFE registers within TFM as a PSA crypto driver. It is integrated as a provider of services and is accessible through the PSA APIs with dedicated vendor keys.

When this option is activated in TFM_Appli nonsecure project, STSAFE can be tested from the user application test TFM menu.

STSAFE comes with a personalization profile and assists the device in the establishment of a secure connection with the infrastructure.

Token authenticity is based on STSAFE: a hardware ID claim is computed from the unique STSAFE ID (instead of STM32 ID) then embedded in the EAT messages. These messages are signed with the private key stored in STSAFE (instead of using the EAT personalized private key). The private key never leaves the secure enclave whereas the public key, the X.509 public key certificate are retrieved on PSA requests.

STSAFE also comes with nonvolatile data areas with access conditions defined in [AN5435].

USE_PAIRING

By default, STSAFE is not configured with a defined host and most of the commands are anyway operational.

To protect the entire device and avoid the replacement of a component by another, it is needed to associate STSAFE with STM32 for ever.

Enable this switch in `TFM_Appli\Secure\Inc\stsafea_interface_conf.h` to apply the pairing and set up a secure channel.

Communication becomes secure with the generation of two 128-bit computed pairing keys called the host MAC key (for commands authentication) and the host cipher key (for data encryption) over the link.

Pairing keys are programmed once only in the STSAFE, for instance during its first execution. Then, keys are dynamically retrieved at every secure runtime.

No keys storage is necessary.

```
/* Uncomment to associate and pair definitively STM32 and STSAFE */
//#define USE_PAIRING

#ifdef USE_PAIRING
/* Set to 1 if Host Keys are retrieved from a generation algorithm during the runtime.
   Set to 0 otherwise to use static keys values */
#define USE_COMPUTED_HOST_KEYS       1U

/* Set to 1 if STSAFE has never been provisioned */
/* Set to 0 otherwise to avoid self-verification of provisioning state */
#define USE_SELF_PROVISIONING        1U
#endif /* USE_PAIRING */
```

Slot mode

The TFM application example is statically defined to use the primary and secondary slots. This configuration allows over-the-air firmware image update, because the download of an image in the secondary slot can be performed by the firmware image executing in the primary slot. However, it is possible to adapt the example to use only the primary slot, so that the slot area size can be maximized. In that case, over-the-air firmware update is not possible.

The primary slot only configuration is used by default for each image in the SBSFU application. In this mode, the local loader downloads the encrypted image directly in the primary slot, and the image is decrypted in place during the installation process. It is possible to use primary and secondary slot mode, to have the image decrypted during the installation from the secondary slot to the primary slot. This is configured using the `MCUBOOT_PRIMARY_ONLY` define in `Linker\flash_layout.h`.

```
#define MCUBOOT_PRIMARY_ONLY       /* Defined: No secondary (download) slot(s),
                                            only primary slot(s) for each image.
                                      Undefined: Primary and secondary slot(s) for each image. */
```

The devices with 512 Kbytes of flash memory only support the primary slot configuration.

Image upgrade strategy

MCUboot supports either swap- or overwrite-based image upgrades, for primary and secondary slots configuration.

In the overwrite-based image upgrade mode, the image in the secondary slot overwrites the image in the primary slot. There is no possibility to revert the image in this mode.

In the swap-based image upgrade mode, the images in the primary and secondary slots are swapped. After the swap, the user application must confirm the new image in the primary slot. If this is not done, the images are swapped back at the next boot. By default, the overwrite mode is used. It is possible to select the image upgrade strategy using the `MCUBOOT_IMAGE_NUMBER` define in `Linker\flash_layout.h`.

```
#define MCUBOOT_OVERWRITE_ONLY     /* Defined: the FW installation uses ovewrite method.
                                      UnDefined: The FW installation uses swap mode. */
```

Application RoT

By default, application RoT is enabled in the TFM secure application, with an example of OEM secure service. It is possible to disable it using the `TFM_PARTITION_APP_ROT` define in `Linker\flash_layout.h`.

```
#define TFM_PARTITION_APP_ROT       /* comment to remove APP_ROT partition */
```

Antitamper

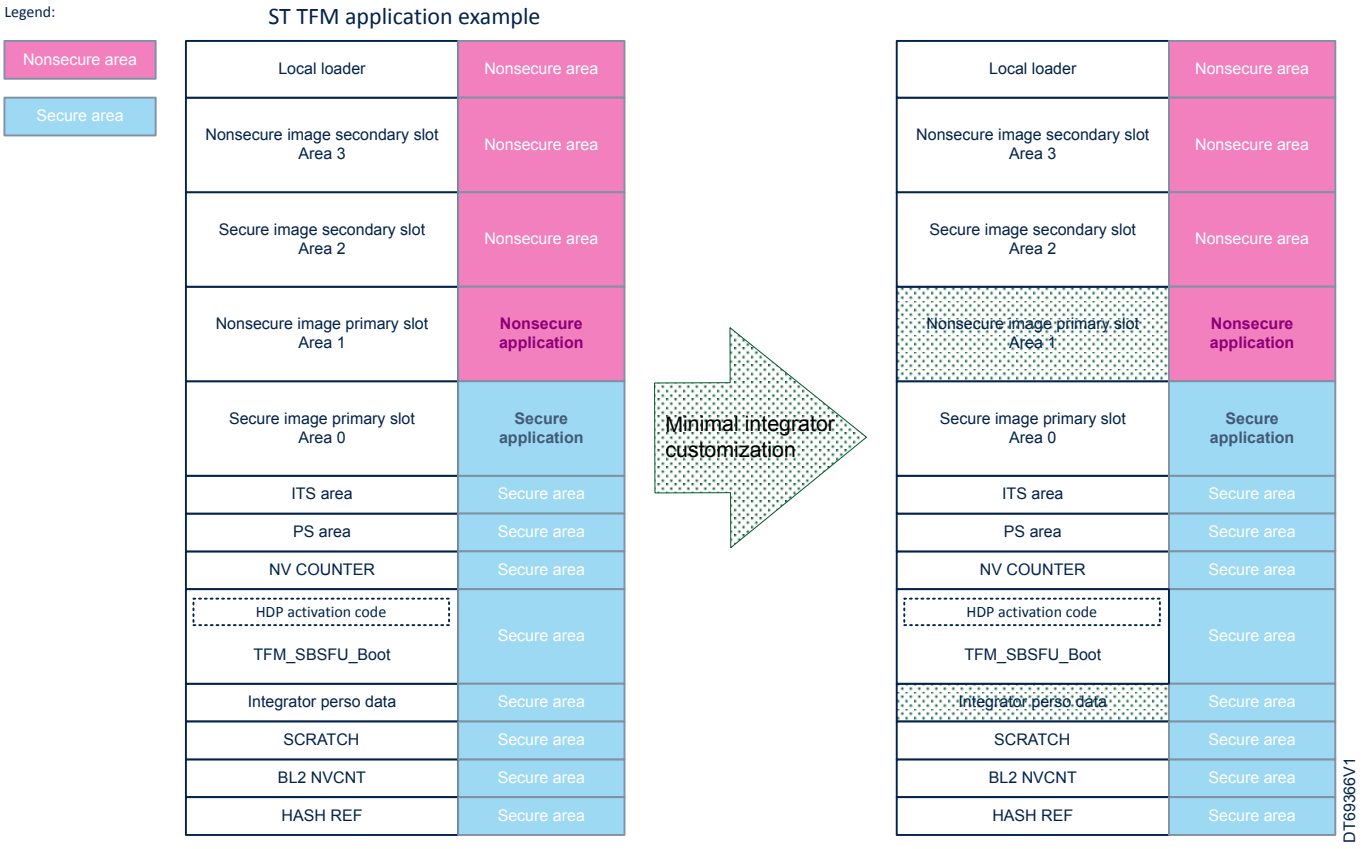
In the TFM and SBSFU applications, by default the antitamper protection is enabled for both internal tamper events (Backup domain voltage threshold monitoring and Cryptographic IPs fault: SAES or AES or PKA or TRNG) and external tamper events (using TAMP_IN8 (PE4 pin) and TAMP_OUT8 (PE5 pin)). It is possible to change this configuration using the `TFM_TAMPER_ENABLE` define in `TFM_SBSFU_Boot\Inc\boot_hal_cfg.h`.

```
#define NO_TAMPER            (0)              /*!< No tamper activated */
#define INTERNAL_TAMPER_ONLY (1)              /*!< Only Internal tamper activated */
#define ALL_TAMPER           (2)              /*!< Internal and External tamper activated */
#define TFM_TAMPER_ENABLE ALL_TAMPER          /*!< TAMPER configuration flag  */
```

12.2 Minimal customization

At this stage, the integrator must at least customize the following:

Figure 67. Integrator minimal customizations



- Replace the nonsecure TFM application example delivered in the STM32CubeU5 MCU Package by its own product nonsecure application. Integrators can keep the nonsecure project structure but must integrate their own source code in the project `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Appli\NonSecure`.
- Personalize some TFM immutable data specific to the integrator or product specific. Integrators must build their own binaries containing their own data as listed below:
  - RSA-2048 or RSA-3072 or EC-256 public key for secure image authentication
  - RSA-2048 or RSA-3072 or EC-256 public key for nonsecure image authentication, in the case of two firmware images configuration
  - RSA-2048 or EC-256 private key for AES-CTR key decryption
  - EAT private key (unique to each device) if not preferably provisioned in and retrieved from secure data image
  - HUK (unique to each device) for software-only cryptography

  These data can be personalized in the file `TFM/TFM_SBSFU_Boot/Src/keys.c` or in the TFM_SBSFU_Boot binary itself.
- Personalize the keys used to prepare the images:
  - RSA-2048 or RSA-3072 or EC-256 private key for secure image authentication
  - RSA-2048 or RSA-3072 or EC-256 private key for nonsecure image authentication in case of two-image configuration
  - RSA-2048 or EC-256 public key for AES-CTR key decryption

  These data can be personalized in the default keys files (`.pem`) in `TFM/TFM_SBSFU_Boot/Src`.

Table 6. Integrator personalized data in source code

Personalized data		Variable and source file	In integrator personalized area binary
RSA-2048 crypto scheme	RSA-2048 private key for secure image signature generation	`TFM\TFM_SBSFU_Boot\Src\root-rsa-2048.pem`	No (used by the TFM_Appli postbuild)
	RSA-2048 private key for nonsecure image signature generation	`TFM\TFM_SBSFU_Boot\Src\root-rsa-2048_1.pem`	
	RSA-2048 public key for secure image signature verification	`rsa2048_pub_key` in `TFM\TFM_SBSFU_Boot\Src\keys.c`	Yes
	RSA-2048 public key for nonsecure image signature verification	`rsa2048_pub_key_1` in `TFM\TFM_SBSFU_Boot\Src\keys.c`	
	RSA-2048 private key for AES-CTR key decryption	`enc_rsa_priv_key` in `TFM\TFM_SBSFU_Boot\Src\keys.c`	
	RSA-2048 public key for AES-CTR key encryption	`TFM\TFM_SBSFU_Boot\Src\enc-rsa2048-pub.pem`	No (used by the TFM_Appli postbuild)
RSA-3072 crypto scheme	RSA-3072 private key for secure image signature generation	`TFM\TFM_SBSFU_Boot\Src\root-rsa-3072.pem`	
	RSA-3072 private key for nonsecure image signature generation	`TFM\TFM_SBSFU_Boot\Src\root-rsa-3072_1.pem`	
	RSA-3072 public key for secure image signature verification	`rsa3072_pub_key` in `TFM\TFM_SBSFU_Boot\Src\keys.c`	Yes
	RSA-3072 public key for nonsecure image signature verification	`rsa3072_pub_key_1` in `TFM\TFM_SBSFU_Boot\Src\keys.c`	
	RSA-2048 private key for AES-CTR key decryption	`enc_rsa_priv_key` in `TFM\TFM_SBSFU_Boot\Src\keys.c`	
	RSA-2048 public key for AES-CTR key encryption	`TFM\TFM_SBSFU_Boot\Src\enc-rsa2048-pub.pem`	No (used by the TFM_Appli postbuild)
EC-256 crypto scheme	EC-256 private key for secure image signature generation	`TFM\TFM_SBSFU_Boot\Src\root-ec-p256.pem`	
	EC-256 private key for nonsecure image signature generation	`TFM\TFM_SBSFU_Boot\Src\root-ec-p256_1.pem`	
	EC-256 public key for secure image signature verification	`ecdsa_pub_key` in `TFM\TFM_SBSFU_Boot\Src\keys.c`	Yes
	EC-256 public key for nonsecure image signature verification	`ecdsa_pub_key_1` in `TFM\TFM_SBSFU_Boot\Src\keys.c`	
	EC-256 private key for AES-CTR key decryption	`enc_ec256_priv_key` in `TFM\TFM_SBSFU_Boot\Src\keys.c`	

Personalized data		Variable and source file	In integrator personalized area binary
EC-256 crypto scheme	EC-256 public key for AES-CTR key encryption	`TFM\TFM_SBSFU_Boot\Src\enc-ec256-pub.pem`	No (used by the TFM_Appli postbuild)
HUK for AES-GCM based AEAD encryption for the protected storage service (for full software cryptography configuration only)		`huk_value` in `TFM\TFM_SBSFU_Boot\Src\keys.c`	Yes
EAT private key for the initial attestation service		`initial_attestation_priv_key` in `TFM\TFM_SBSFU_Boot\Src\keys.c`	

Key generator procedure

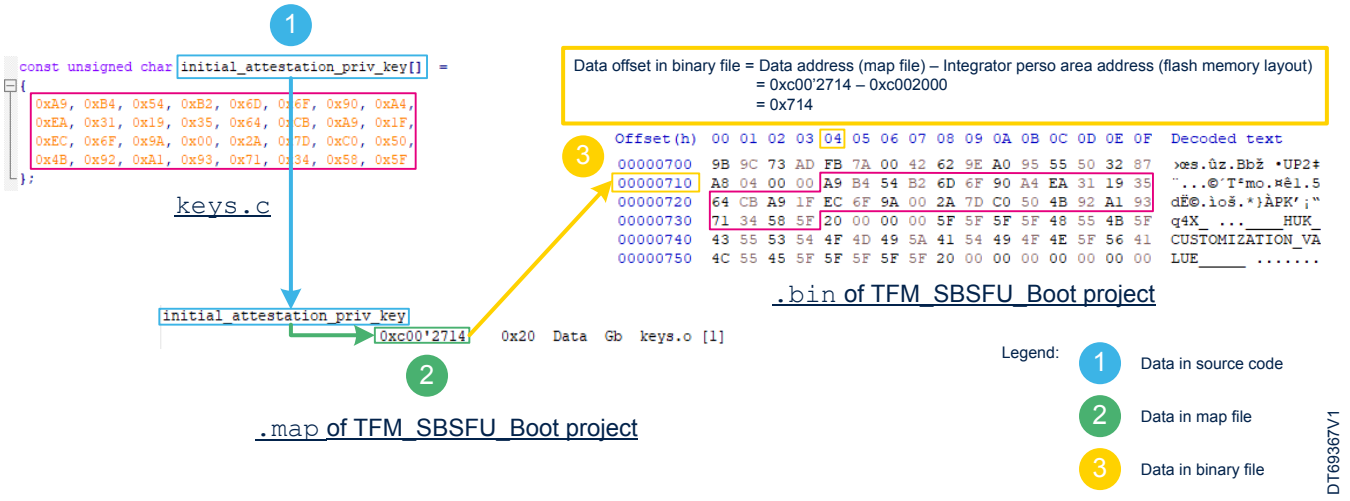
To personalize the keys (mandatory for production to replace the default keys), a script generating random keys is provided in the STM32CubeU5 MCU Package:

- EWARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\EWARM\keygen.bat`
- MDK-ARM: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\MDK-ARM\keygen.bat`
- STM32CubeIDE: `Projects\B-U585I-IOT02A\Applications\TFM\TFM_SBSFU_Boot\STM32CubeIDE\keygen.sh`

When using these automatic scripts, integrators must check that no error is reported during the script execution. The random-generated keys are placed into the file `keys.c` and all the `.pem` files mentioned previously, replacing the default ones. These scripts also generate an attestation private key in the file `s_data.bin`. The binary is then processed to produce a secure data image (refer to Step 1.6 in Section 10.1 Application compilation process).

The exact location of each data in the binary is toolchain dependent. It can be identified through the map file of the TFM_SBSFU_Boot application.

Figure 68. Integrator personalized data in TFM_SBSFU_Boot binary (initial_attestation_priv_key example)



12.3 Other customization

The integrator can also change or configure the source code of the three TFM projects delivered in the STM32CubeU5 MCU Package via compiler switches to do additional customizations such as:

- integrate additional secure services into the secure application.
- remove some cryptographic algorithms.
- adapt the internal user flash memory mapping.
- change IDE compiler options.
  Tune the scratch area size according to the expected number of upgrades in the product life.
- adapt the SRAM memory layout.

For example:

- Cryptographic algorithms can be disabled through compile switches (refer to Performance).
- The internal user flash memory mapping can be modified in files `flash_layout.h` and `region_defs.h` (refer to Section 8.3 Memory layout). For example, increase `FLASH_S_PARTITION_SIZE`, `FLASH_NS_PARTITION_SIZE` or both. The different flash memory areas can also be tuned to lower the size according to the effective needed size. Indeed, by default, the flash memory areas are sized to be compatible with all possible software configurations and supported IDEs (refer to Memory footprint).
  The software programming addresses as well as the protection configurations are automatically updated during the TFM_SBSFU_Boot postbuild according to the flash memory layout changes in the scripts `regression.bat` (or `.sh`), `TFM_UPDATE.bat` (or `.sh`), and `hardening.bat` (or `.sh`). However, after having changed the internal user flash memory mapping, it is required that the integrator verifies the security protections.
- To use SRAM3 in the TFM_Appli nonsecure application instead of SRAM1, replace `NS_DATA_START` and `NS_DATA_SIZE` with `NS_DATA_START_2` and `NS_DATA_SIZE_2` in the TFM_Appli nonsecure linker file.
  To use both SRAM3 and SRAM1, add a new data section with `NS_DATA_START_2` and `NS_DATA_SIZE_2` in the TFM_Appli nonsecure linker file.

12.4 Production

At the end of the development phase, the integrator must enable the production mode (refer to Section 10.1 Application compilation process).

It is the integrator's responsibility to put in place a secure personalization process in product manufacturing in order to keep the confidentiality of product security assets (TFM immutable data specific to the integrator or specific to a product) until they are provisioned in the STM32U5 device and until STM32U5 device security is fully activated. Once the STM32U5 microcontroller security is fully activated, the confidentiality of the product security assets is ensured by the STM32U5 microcontroller security protections. However, if the customer cannot rely on a trusted manufacturing, then the secure firmware installation service (refer to [AN4992]) embedded inside STM32U5 can be used.

It is mandatory to change the keys and not use the default keys provided as examples (refer to Key generator procedure in Section 12.2 Minimal customization).

Appendix A Development hardware boards

B-U585I-IOT02A Discovery kit

This board supports the examples for the STM32U5 microcontrollers with 2 Mbytes of flash memory.

The antitamper protection is enabled with active tamper pins usage by default. Connect TAMP_IN8 (PE4 on CN3 pin 11) and TAMP_OUT8 (PE5 on CN3 pin 14) on the B-U585I-IOT02A board as indicated in Figure 69 and Figure 70. If this is not done, the antitamper protection prevents the application to run. If the tamper pins are opened, then the application is reset and blocked.

The BOOT0 pin must also be configured to boot from the flash memory in the case of the development mode: the switch SW1 must be configured on the position 0 indicated in Figure 69.

Figure 69. B-U585I-IOT02A board setup

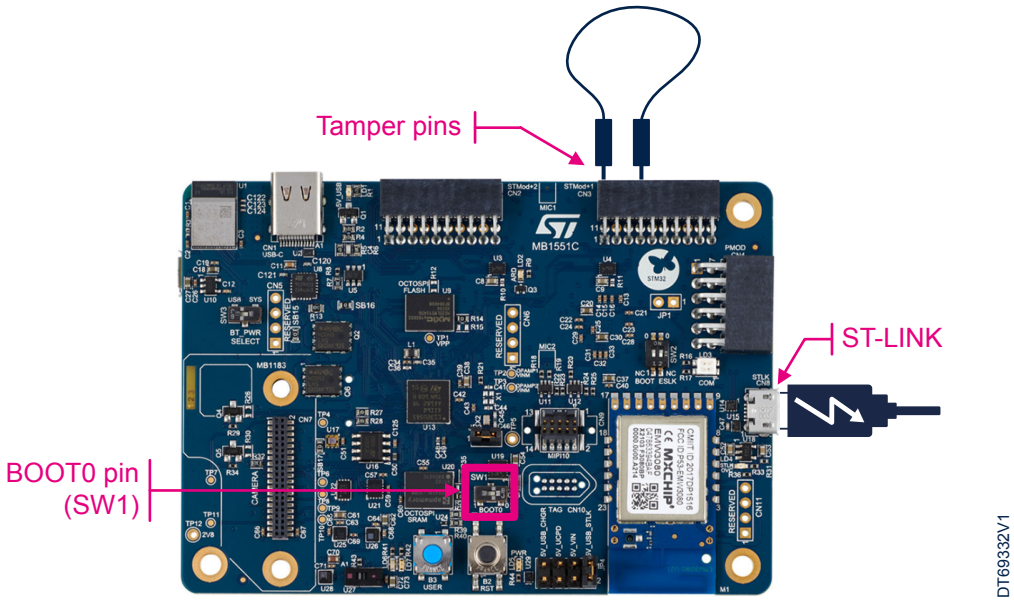


Figure 70. B-U585I-IOT02A board setup (detail)

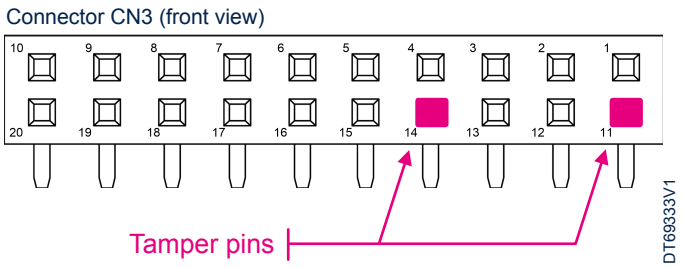


Figure 71 and Figure 72 show the reset button and the JP3 jumper (IDD) used to disable the ST-LINK interface.

Figure 71. Reset button on the B-U585I-IOT02A

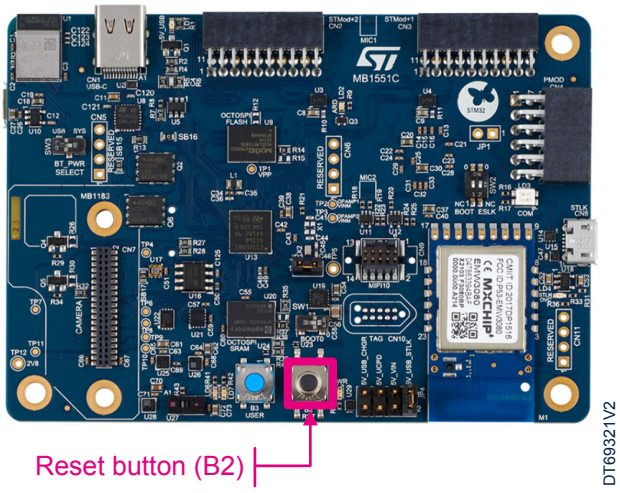
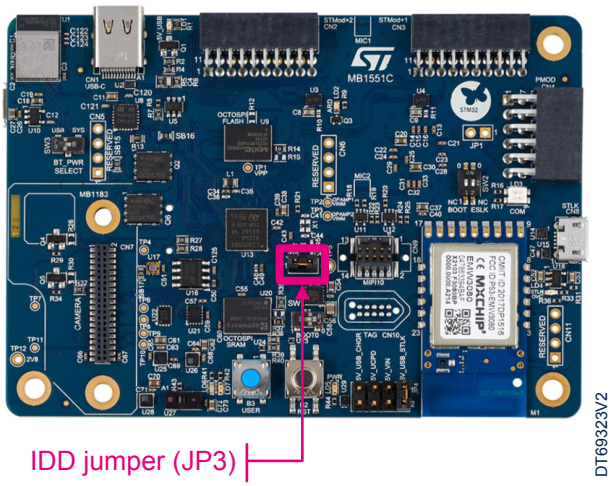


Figure 72. Jumper JP3 (IDD) on the B-U585I-IOT02A board



STM32U5A9J-DK Discovery kit

This board supports the examples for STM32U5 microcontrollers with 4 Mbytes of flash memory.

The antitamper protection is enabled with active tamper pins usage by default. Connect TAMP_IN8 (PE4 on CN10 pin 7) and TAMP_OUT8 (PE5 on CN10 pin 5) on the STM32U5A9J-DK board as indicated in Figure 73 and Figure 74. If this is not done, the antitamper protection prevents the application to run. If the tamper pins are opened, then the application is reset and blocked.

The BOOT0 pin must also be configured to boot from the flash memory in the case of the development mode: the switch SW1 must be configured on the position 0 indicated in Figure 73.

Figure 73. STM32U5A9J-DK board setup

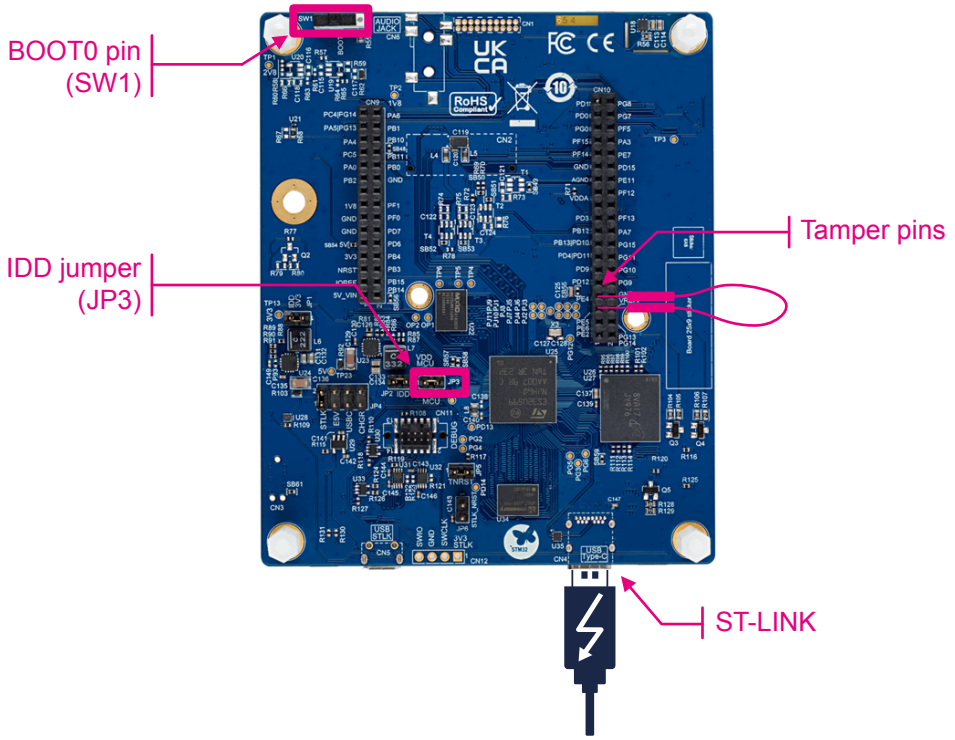
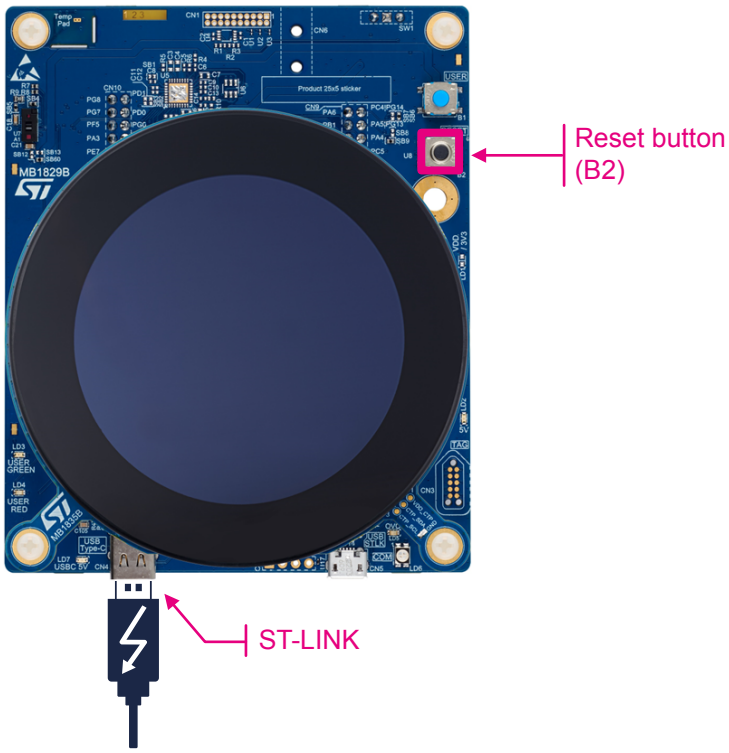


Figure 74. STM32U5A9J-DK board setup (detail)

Figure 75 and Figure 73 show the reset button and the JP3 jumper (IDD) used to disable the ST-LINK interface.

Figure 75. Reset button on the STM32U5A9J-DK

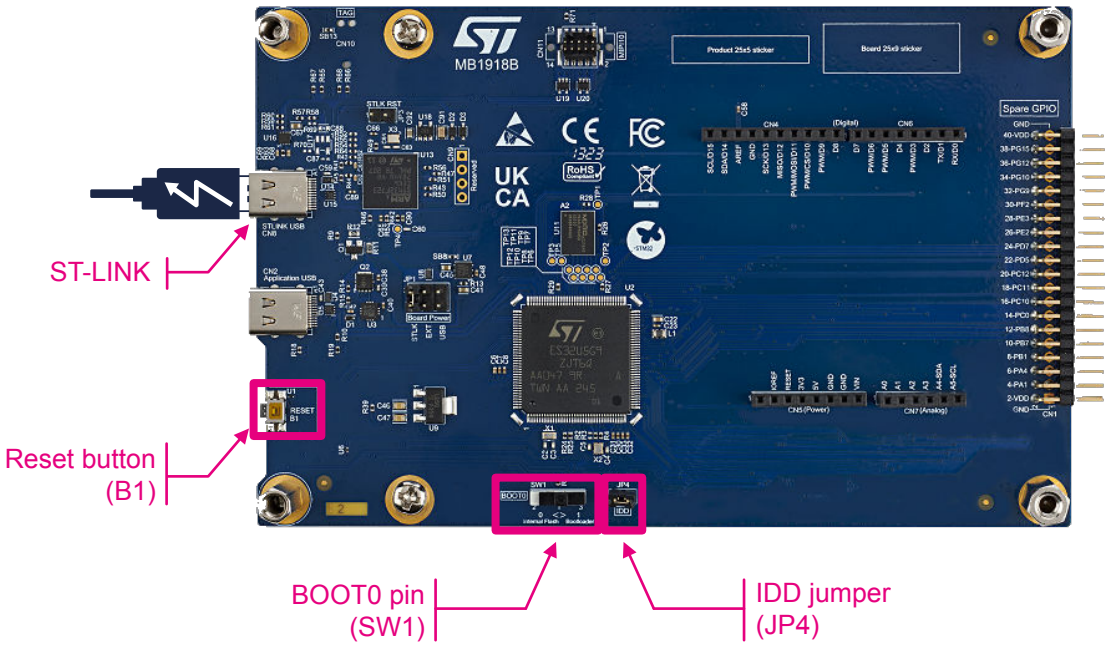


STM32U5G9J-DK2 Discovery kit

This board supports the examples for STM32U5 microcontrollers with 4 Mbytes of flash memory.

The BOOT0 pin must be configured to boot from the flash memory in the case of the development mode: the switch SW1 must be configured on the position 0 indicated in Figure 76. This figure also shows the reset button, and the JP4 jumper (IDD) used to disable the ST-LINK interface.

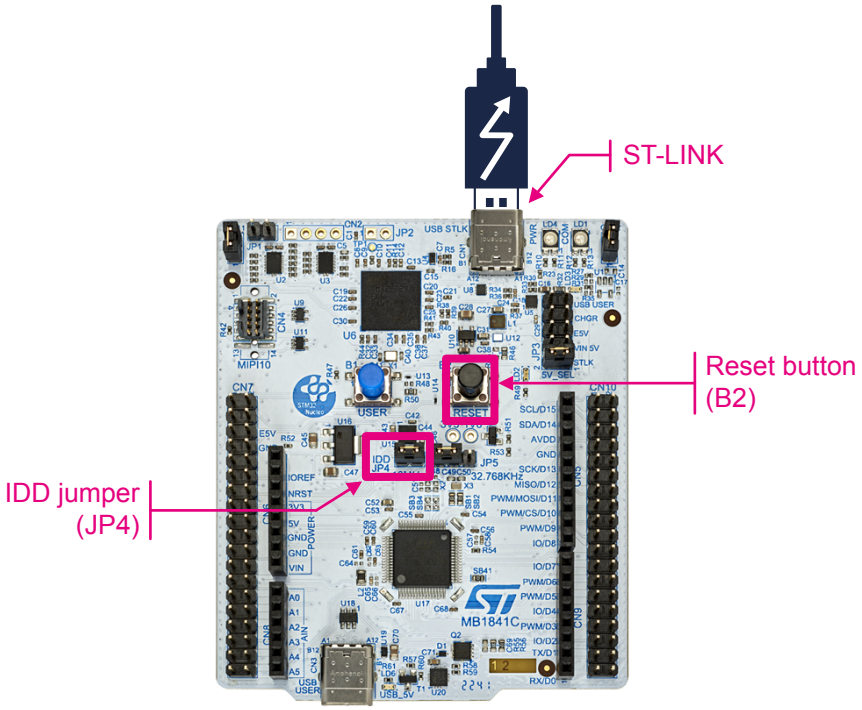
Figure 76. STM32U5G9J-DK2 board setup



NUCLEO-U545RE-Q Nucleo-64 board

This board supports the examples for the STM32U5 microcontrollers with 512 Kbytes of flash memory. Figure 77 shows the reset button and the JP4 jumper (IDD) used to disable the ST-LINK interface.

Figure 77. Reset button and JP4 jumper (IDD) on the NUCLEO-U545RE-Q



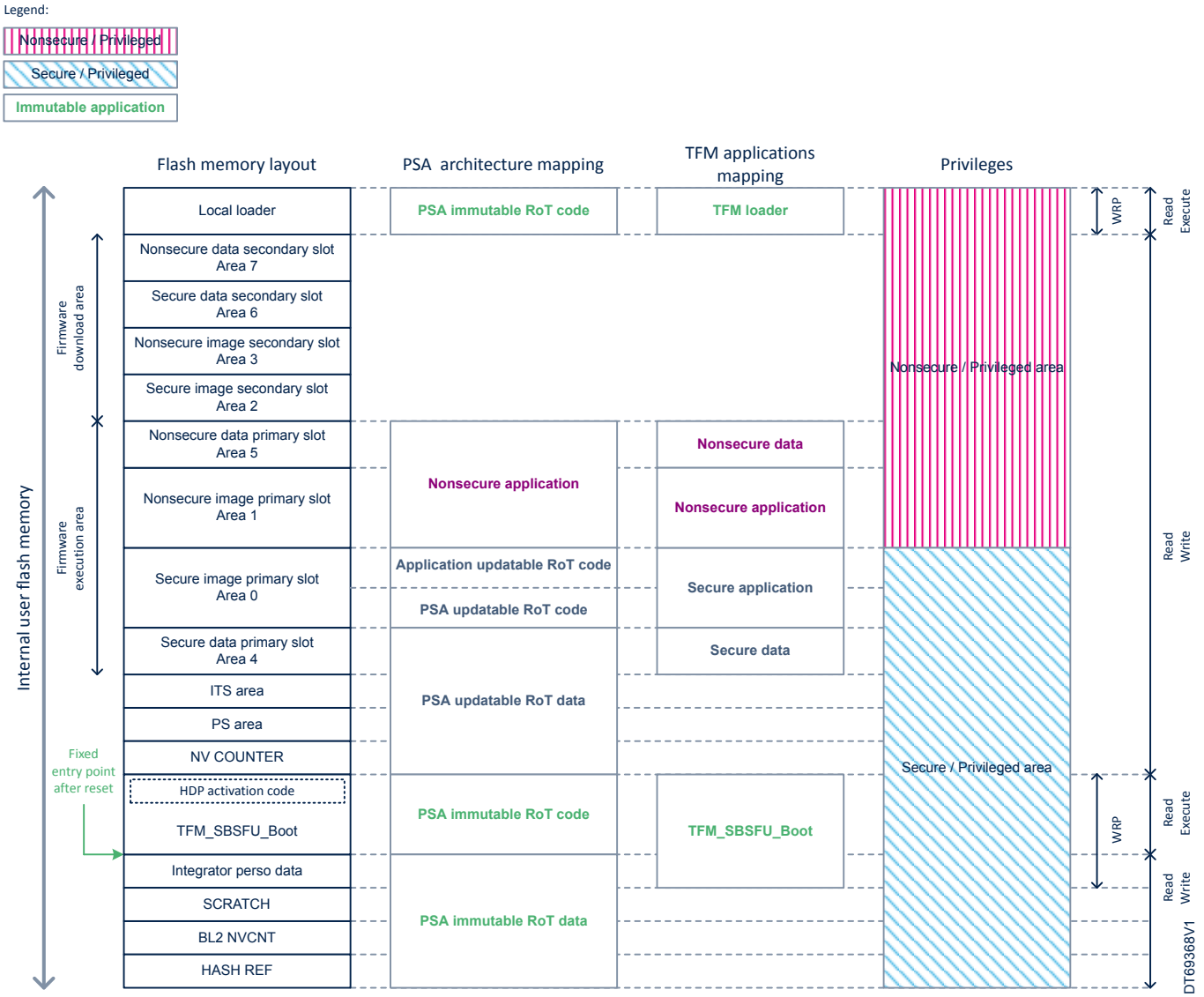
Appendix B Memory protections

B.1 Flash memory protections

During the different applications execution, the flash memory protections are achieved by combining SAU, MPU, and GTZC configurations.

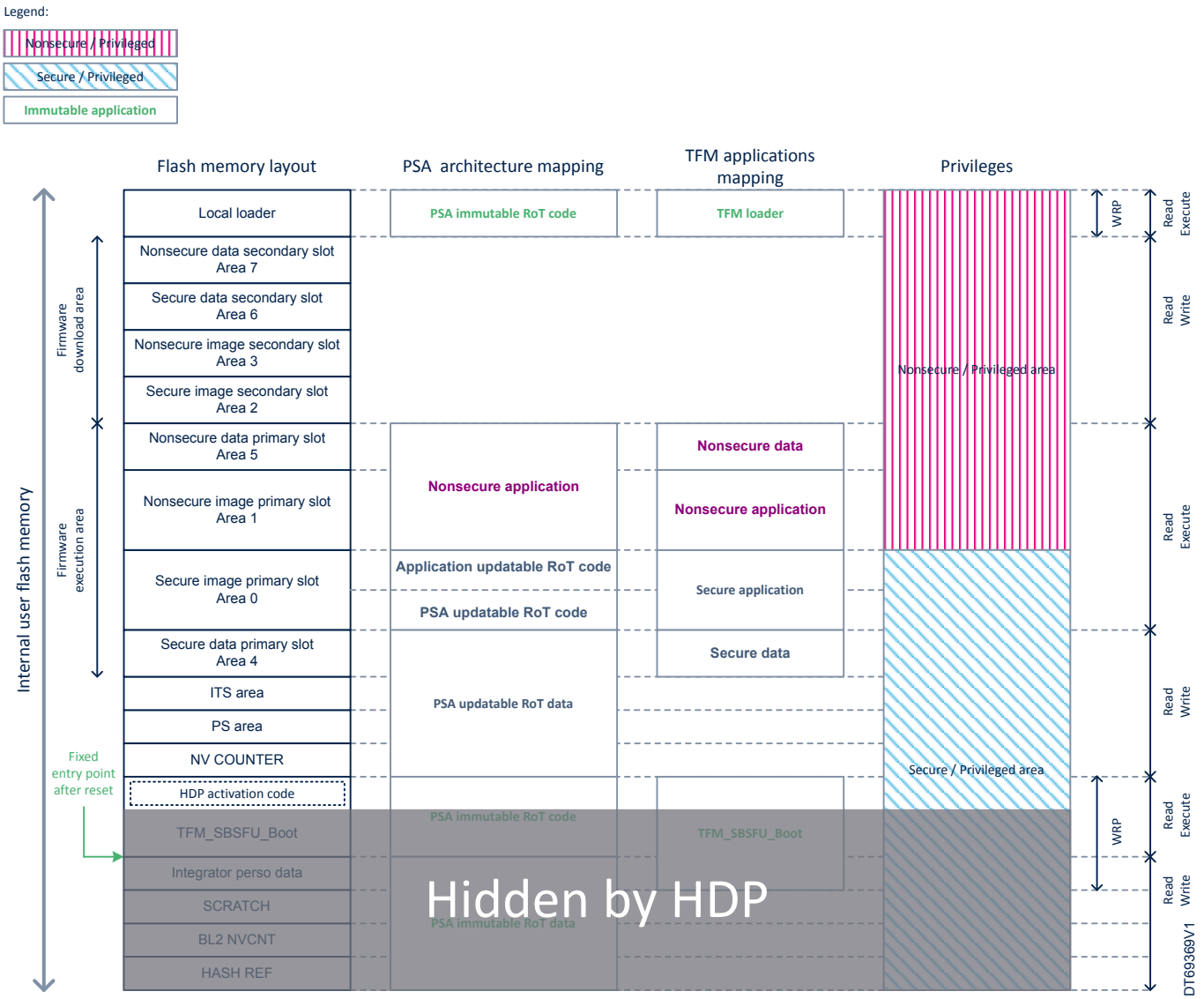
During the TFM_SBSFU_Boot execution, the TFM_SBSFU_Boot code area is the only flash memory area allowed to be executed, with the immutable local loader.

Figure 78. Flash memory protection overview during TFM_SBSFU_Boot application execution



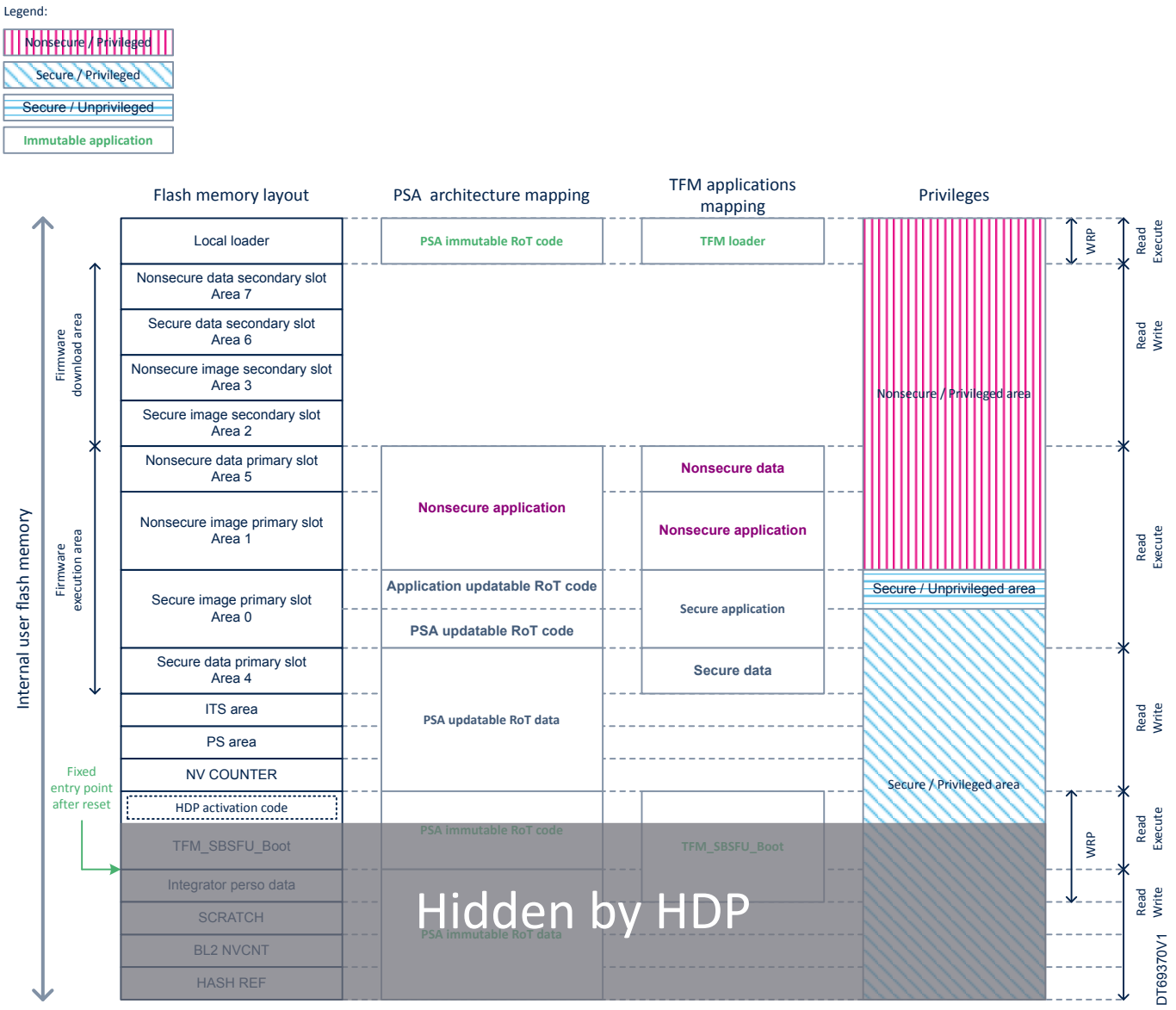
When leaving the TFM_SBSFU_Boot application for jumping to the secure application, all flash memory areas dedicated to the TFM_SBSFU_Boot execution are hidden, and the execution is allowed in the secure and nonsecure primary slot areas.

Figure 79. Flash memory protection overview when leaving TFM_SBSFU_Boot application to TFM application



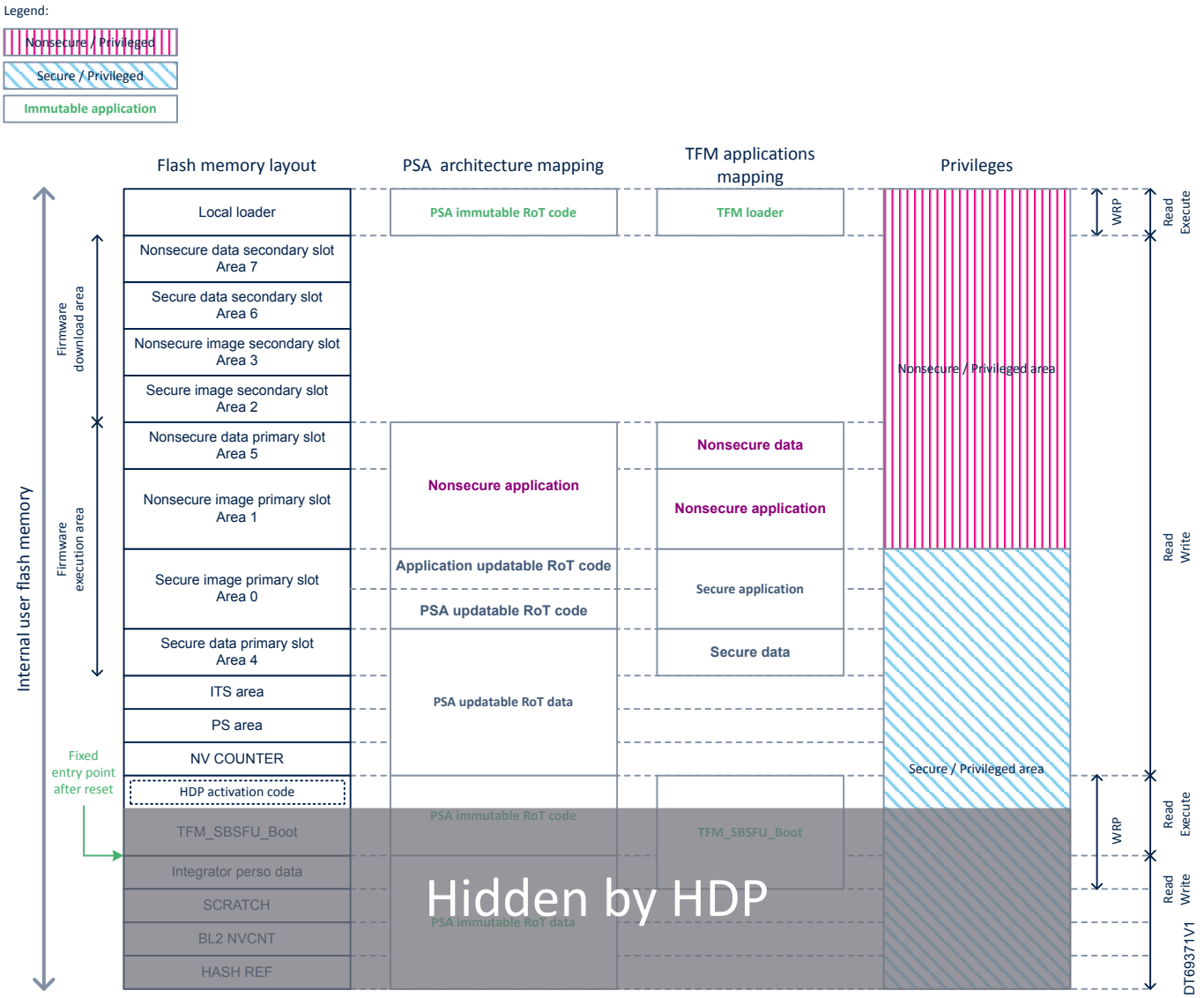
Once the secure application has performed the SPM initialization, a secure unprivileged area is created inside the secure image primary slot for the application RoT.

Figure 80. Flash memory protection overview during application execution



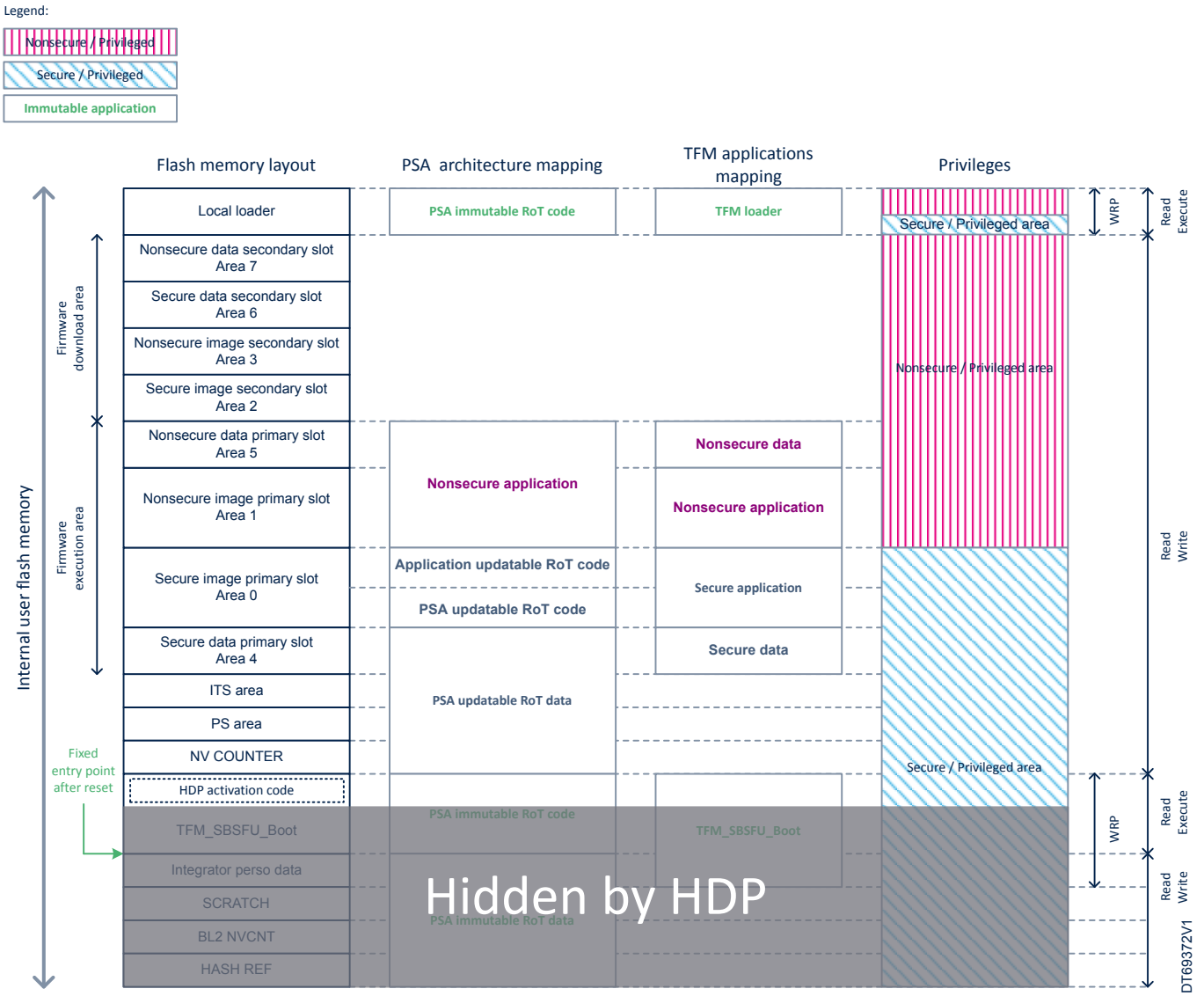
When leaving the TFM_SBSFU_Boot application for jumping to the nonsecure local loader application (primary and secondary slot configuration), all flash memory areas dedicated to the TFM_SBSFU_Boot execution are hidden, and the SAU/MPU configurations are locked.

Figure 81. Flash memory protection overview when leaving TFM_SBSFU_Boot application to nonsecure local loader application



When leaving the TFM_SBSFU_Boot application for jumping to the secure and nonsecure local loader application (primary only slot configuration), all flash memory areas dedicated to TFM_SBSFU_Boot execution are hidden, the SAU/MPU configurations are locked, and the flash memory area corresponding to the secure part of the local loader is configured as secure.

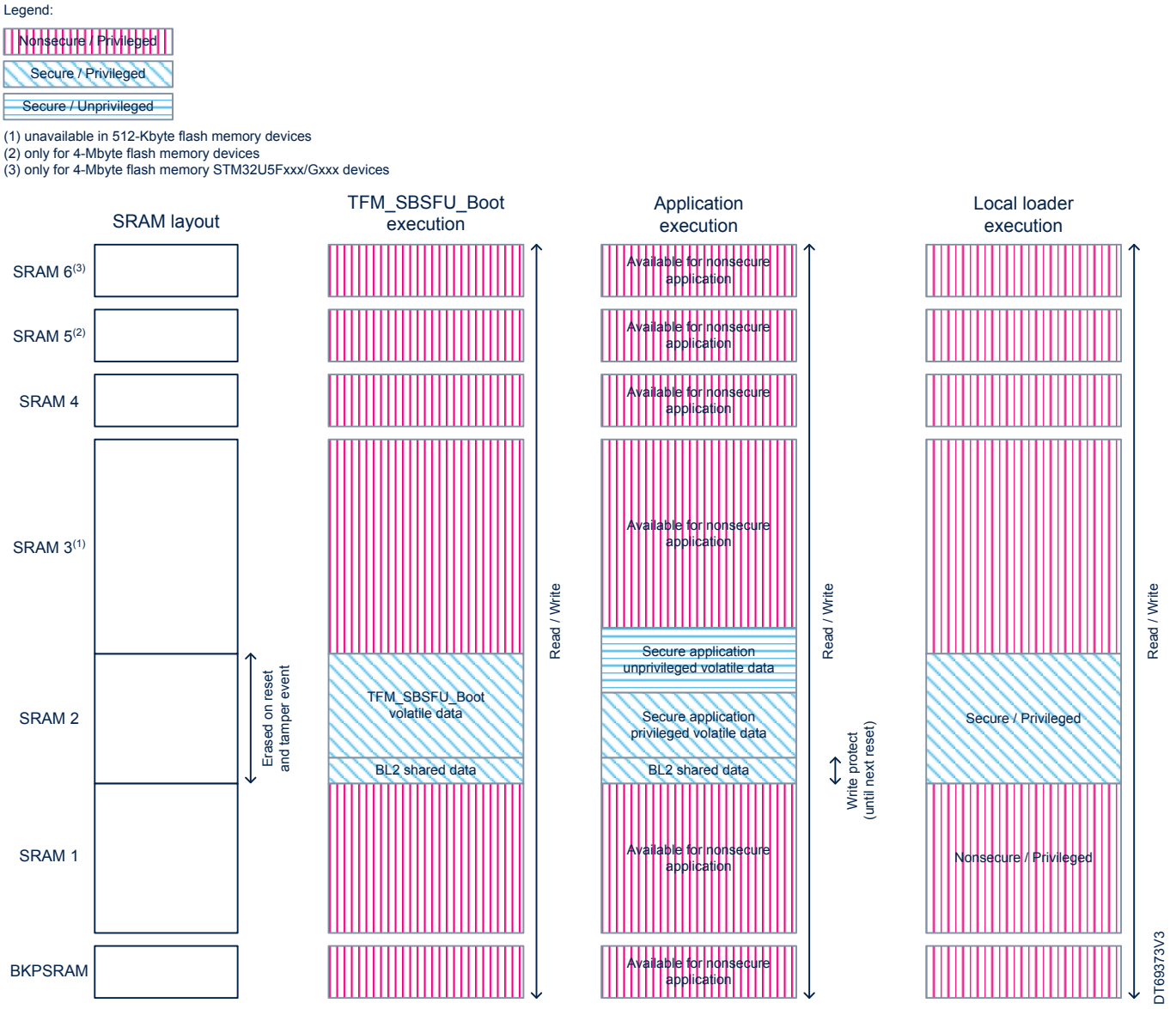
Figure 82. Flash memory protection overview when leaving TFM_SBSFU_Boot application to secure and nonsecure local loader application



B.2 SRAM protections

The SRAM memory protection is achieved by combining the SAU, MPU, and GTZC configurations. GTZC and MPU are used to fix privileged and unprivileged accesses. The GTZC protection prevents illegal accesses from the peripheral bus controller (such as DMA). The MPU prevents the SRAM from being executed. During the application execution, the BL2 shared data area is write protected.

Figure 83. SRAM protections overview



Appendix C Memory footprint

As described in Section 8.2, the TFM-based application examples consist of four main software components, which can be configured by the integrators according to their needs:

- TFM_SBSFU_Boot: Secure Boot and Secure Firmware Update application
- TFM_Loader: application loader application based on Ymodem protocol over USART
- TFM_Appli_Secure: secure application providing secure services to the nonsecure user application (at runtime)
- TFM_Appli_NonSecure: nonsecure user application

The size of these four main software components depends on the configuration selected by the integrator:

- Hardware configuration:
  - STM32U5 hardware-accelerated cryptography capability
- Development or production mode: logs and automatic security activation added in development mode
- Number of firmware images:
  - Single firmware image combining the secure application and the nonsecure application
  - Two firmware images: secure application image and nonsecure application image
- Number of firmware slots:
  - Primary and secondary slots: the installed firmware image can download a new firmware image (download while running the user application)
  - Primary slot only: the installed firmware image is overwritten and only a standalone loader application can download a new firmware image
- Image upgrade strategy (for primary and secondary slots configuration):
  - Overwrite mode: new firmware image in secondary slot is overwriting image in primary slot
  - Swap mode: actual firmware image in primary slot and new firmware image in secondary slot are swapped
- SBSFU crypto scheme configuration
  - Asymmetric crypto scheme based on RSA or on ECC
  - Firmware encryption support
- Standalone local loader capability
- Type and amount of secure services needed by the nonsecure application
  - Initial attestation secure service
  - Protected storage service
  - Internal trusted storage service
  - Cryptographic services

Moreover, the size of these four main software components depends on the IDE compiler used (such as EWARM, MDK-ARM, or STM32CubeIDE). In the next sections, all memory footprint indications are given with optimizations to reduce the code size.

Therefore, the flash memory layout (refer to Section 8.3 Memory layout) delivered by default in the TFM application example can be optimized by integrators according to their needs in order to maximize the size of the nonsecure application.

To change the flash memory mapping, the integrator must modify the files `flash_layout.h` and `region_defs.h` in the folder `TFM/Linker`. All flash memory areas (except TFM_SBSFU_Boot area) must be aligned on an address offset multiple of 8 Kbytes.

If the integrator changes the flash memory mapping, the security protections must be verified.

The next sections describe how the different configurations impact the size of the four main software components and especially how it impacts the area size that can be used for the nonsecure application.

C.1 TFM_SBSFU_Boot memory footprint

The TFM_SBSFU_Boot application consists of the sections in the flash memory described in Section 8.3 Memory layout. The sizes of these flash memory sections can be impacted by the configuration described in Table 7.

Table 7. SBSFU configuration option

Items	Configuration possibility	Size impact(1)
HASH REF data	Maximum number of images	32 bytes / image Example: 8 Kbytes for 32 images.
BL2 NVCNT data	Maximum number of version updates during product life.	16 bytes / version update Example: 8 Kbytes for 500 version updates.
SCRATCH area	Required in swap mode configuration only. Increasing the SCRATCH area size reduces the number of SCRATCH area erase cycles during the swap process. *Note: the image slot size does not need to be a multiple of scratch size; only the effective image size is copied in the SCRATCH area.*	num_upgrades = number_of_erase_cycles / (image_size / scratch_size) Example: 64 Kbytes for 10 k programming cycles, 640 Kbytes firmware image size, and 1000 firmware image updates
Integrator perso data	SBSFU crypto scheme: • RSA keys or ECC keys, and presence of additional keys for image encryption if enabled. Number of firmware images managed by SBSFU: • 1 key per firmware image. TFM secure services information: • Keys/information for initial attestation secure service • HUK for protected storage services (for full software cryptography configuration)	EC-256, no image encryption, 1 firmware image only, and no TFM secure services: • EWARM: 0.2 Kbyte • MDK-ARM: 0.2 Kbyte • STM32CubeIDE: 0.2 Kbyte RSA-3072, image encryption, 2 firmware images, and TFM secure services: • EWARM: 1.8 Kbyte • MDK-ARM: 1.9 Kbyte • STM32CubeIDE: 1.9 Kbyte
SBSFU code	Minimal code size	• EWARM: 32 Kbytes • MDK-ARM: 32 Kbytes • STM32CubeIDE: 48 Kbytes
	SBSFU or TFM_SBSFU mode: • Specific TFM operations needed in TFM_SBSFU mode	TFM operations: • EWARM: + 1.1 Kbyte • MDK-ARM: + 1.1 Kbyte • STM32CubeIDE: + 1.1 Kbyte
	Development or production mode: • Logs and automatic security activation added in development mode	Development mode: • EWARM: + 7.3 Kbytes • MDK-ARM: + 6.5 Kbytes • STM32CubeIDE: + 10 Kbytes
	Local loader compatibility: • Specific processing to interact with a nonsecure standalone loader application	With local loader compatibility: • EWARM: + 0.7 Kbyte • MDK-ARM: + 0.7 Kbyte • STM32CubeIDE: + 0.7 Kbyte
	Number of firmware slots: • Additional code needed to manage the 2 firmware slots	2 firmware slots: • EWARM: + 1.8 Kbyte • MDK-ARM: + 1.5 Kbyte • STM32CubeIDE: + 1.8 Kbyte
	Number of application images: • Additional code needed to manage the 2 firmware images	2 firmware images: • EWARM: + 0.4 Kbyte • MDK-ARM: + 0.4 Kbyte • STM32CubeIDE: + 0.4 Kbyte

Items	Configuration possibility	Size impact(1)
SBSFU code	Number of data images: • Additional code needed to manage the 2 data images	2 data images: • EWARM: + 0.4 Kbyte • MDK-ARM: + 0.3 Kbyte • STM32CubeIDE: + 0.4 Kbyte
SBSFU code	Image upgrade strategy: • Additional code needed to manage swap mode, compared to overwrite mode	Swap mode: • EWARM: + 2.1 Kbytes • MDK-ARM: + 2.5 Kbytes • STM32CubeIDE: + 2.1 Kbytes
	Hardware crypto acceleration: • mbed-crypto code bigger than crypto HAL drivers	Without hardware crypto acceleration: • EWARM: + 0.9 Kbyte • MDK-ARM: + 1 Kbyte • STM32CubeIDE: + 0.9 Kbyte
	Crypto scheme: • Different code sizes according to crypto algorithms (RSA or ECC) and firmware encryption activation	EC-256 firmware encryption: • EWARM: + 7 Kbytes • MDK-ARM: + 6.9 Kbytes • STM32CubeIDE: + 7.4 Kbytes RSA-2048 firmware encryption: • EWARM: + 8 Kbytes • MDK-ARM: + 8.2 Kbytes • STM32CubeIDE: + 8.3 Kbytes
	Antitamper: • Additional code for antitamper (internal and external) protection	Antitamper: • EWARM: + 1.8 Kbyte • MDK-ARM: + 1.7 Kbyte • STM32CubeIDE: + 1.7 Kbyte
	IDE: • Different binary sizes according to the IDE used and IDE compiler options used	Code and IDE compiler dependent
HDP activation code	IDE: • Different binary sizes according to the IDE used and IDE compiler options used	None

1. *Figures given with EWARM version 9.20.1, MDK-ARM version 5.37.0.0, and STM32CubeIDE version 1.10.0.*

Table 8 describes three examples:

- Minimal configuration example
- SBSFU_Boot example delivered in the STM32CubeU5 MCU Package
- TFM_SBSFU_Boot example delivered in the STM32CubeU5 MCU Package

Table 8. SBSFU footprint examples

Items	Minimal configuration	SBSFU example(1)	Full TFM example(1)
HASH REF data	32 SHA256 max	32 SHA256 max	32 SHA256 max
BL2 NVCNT data	500 firmware updates max	500 firmware updates max	500 firmware updates max
SCRATCH area	Not needed (overwrite mode)	Not needed (overwrite mode)	Not needed (overwrite mode)
Integrator perso data	RSA-2048 crypto scheme 1 firmware image No TFM secure services	RSA-2048 crypto scheme **1 firmware image** **No TFM secure services**	RSA-2048 crypto scheme **2 firmware images** **TFM secure services**
SBSFU code	SBSFU mode only Production mode No local loader compatibility 1 firmware slot only 1 firmware application image No data image Overwrite mode Hadware-accelerated cryptography RSA-2048 crypto scheme No firmware encryption No antitamper	**SBSFU mode only** Development mode Local loader compatibility **1 firmware slot only** **1 firmware application image** **No data image** Overwrite mode Hardware-accelerated cryptography RSA-2048 crypto scheme Firmware encryption Internal and external antitamper	**TFM_SBSFU mode** Development mode Local loader compatibility **2 firmware slots** **2 firmware application images** **2 data images** Overwrite mode Hardware-accelerated cryptography RSA-2048 crypto scheme Firmware encryption Internal and external antitamper
IDEs	EWARM version 9.20.1, MDK-ARM version 5.37.0.0, and STM32CubeIDE version 1.10.0		
Total size(2)	HASH REF data: 8 Kbytes BL2 NVCNT data: 8 Kbytes SCRATCH area: 0 Kbyte Integrator perso data: 8 Kbytes SBSFU code: • EWARM: 32 Kbytes • MDK-ARM: 32 Kbytes • STM32CubeIDE: 40 Kbytes HDP activation code: 8 Kbytes Total: • EWARM: 64 Kbytes • MDK-ARM: 64 Kbytes • STM32CubeIDE: 72 Kbytes	HASH REF data: 8 Kbytes BL2 NVCNT data: **4 Kbytes** SCRATCH area: 0 Kbyte Integrator perso data: 8 Kbytes SBSFU code(3): • EWARM: **56 Kbytes** • MDK-ARM: **56 Kbytes** • STM32CubeIDE: **64 Kbytes** HDP activation code: 8 Kbytes Total: • EWARM: **88 Kbytes** • MDK-ARM: **88 Kbytes** • STM32CubeIDE: **96 Kbytes**	HASH REF data: 8 Kbytes BL2 NVCNT data: **8 Kbytes** SCRATCH area: 0 Kbyte Integrator perso data: 8 Kbytes SBSFU code(3): • EWARM: **56 Kbytes** • MDK-ARM: **56 Kbytes** • STM32CubeIDE: **64 Kbytes** HDP activation code: 8 Kbytes Total: • EWARM: **88 Kbytes** • MDK-ARM: **88 Kbytes** • STM32CubeIDE: **96 Kbytes**

1. *Differences between SBSFU and TFM examples are highlighted in bold.*
2. *The size is aligned according to the 8-Kbyte flash memory sector alignment constraints.*
3. *The default size in the example delivered is larger to fit any configuration change.*

C.2 TFM_Appli_Secure memory footprint

The secure application provides secure services that can be used by the nonsecure application at runtime:

- Put in place the security architecture with the isolation of the different domains and with the secure APIs mechanisms
- Provides the secure services needed by the nonsecure user application

The secure application binary is encapsulated in a firmware image, which contains some metadata (refer to image format in Section 8.3 Memory layout) that is used in the context of the "Secure Boot" or "Secure Firmware Update" functions.

The size of the secure application image can be impacted by the configuration described in Table 9.

Table 9. Secure application configuration options

Items	Configuration possibility	Size impact(1)
Number of firmware images	Single firmware image combining the secure application and the nonsecure application: common image metadata (Header + TLV; refer to Figure 17) for the secure application binary and for the nonsecure application binary.	None
	2 firmware images: dedicated image metadata for secure application image and dedicated image metadata nonsecure application image.	+ 2 Kbytes
Image upgrade strategy	Overwrite mode: The slot area is fully available for the firmware image. Swap mode: The last 3 Kbytes of the slot area are reserved for the swap process.	None (for overwrite mode) + 3 Kbytes reserved in the slot area, for each firmware image (for swap mode)
Secure services	No secure services needed at user application runtime: if the integrator does not need any secure services for the user application.	The secure application can be completely removed.
	“Specific” secure services with one level of isolation (secure domain and nonsecure domain) needed at user application runtime: in case the integrator needs some specific secure services for its user application with a 1 level of isolation, then the integrator must implement the specific secure services using the secure application template provided in the SBSFU example. The size of the secure application depends directly on the complexity of the specific secure services implementation with an overhead related to the code to put in place the security infrastructure (1 level of isolation and secure functions export).	~1 Kbyte for the security infrastructure + size of the specific secure services
	PSA L2 type of security infrastructure needed at user application runtime: • Based on open-source TFM reference implementation (TFM-core) • 2 levels of isolation (secure/nonsecure and privileged/unprivileged) • Secure APIs communication mechanisms	None
	Initial attestation service needed at user application runtime: • Based on open-source TFM reference implementation *Note: requires ECDSA cryptography service* • Can be fully deactivated if not needed	• EWARM: + 9.4 Kbytes • MDK-ARM: + 0.5 Kbyte • STM32CubeIDE: + 5.6 Kbytes

Items	Configuration possibility	Size impact[1]
Secure services	Protected storage service needed at user application runtime: • Based on open-source TFM reference implementation • 2 NV data buffers needed (2 flash memory sectors of 8 Kbytes each at minimum) • 1 NV COUNTER buffer needed if the PS area is located in external flash memory *Note: requires AES-GCM cryptography service* • Can be fully deactivated if not needed	Code: • EWARM: + 3.8 Kbytes • MDK-ARM: + 3.5 Kbytes • STM32CubeIDE: + 4 Kbytes Minimum NV data: + 16 Kbytes
	Internal trusted storage needed at user application runtime: • Based on open-source TFM reference implementation • 2 NV data buffers needed (2 flash memory sectors of 8 Kbytes each at minimum) • Can be fully deactivated if not needed	Code: • EWARM: + 8 Kbytes • MDK-ARM: + 8 Kbytes • STM32CubeIDE: + 0.27 Kbyte Minimum NV data: + 16 Kbytes
	Cryptography services needed at user application runtime: • Based on open-source TFM reference implementation • Each algorithm can be deactivated independently (compiler switch at each cryptographic algorithm level) *Note: few algorithms are needed if the initial attestation secure service or if protected storage services are activated.*	When using all crypto algorithms activated by default in the open-source TFM reference implementation, up to: • EWARM: 92 Kbytes • MDK-ARM: 96.5 Kbytes • STM32CubeIDE: 101 Kbytes
	Hardware crypto acceleration: mbed-crypto code bigger than crypto HAL drivers	When using full software mbed-crypto implementation: • EWARM: + 10 Kbytes • MDK-ARM: + 2.46 Kbytes • STM32CubeIDE: + 9.6 Kbytes
	IDE: different binary size according to IDE used and according to IDE compiler options used	Code and IDE dependent
	STSAFE support: • Based on TFM, mbed-crypto, and STSAFE middleware • Not activated by default	PSA crypto driver, communication channel, STSAFE middleware: • EWARM: + 25 Kbytes • MDK-ARM: + 9.4 Kbytes • STM32CubeIDE: + 18.7 Kbytes

1. *Figures given with EWARM version 9.20.1, MDK-ARM version 5.37.0.0, and STM32CubeIDE version 1.10.0.*

Table 10 describes three examples:

- Empty secure application template
- Limited TFM crypto services only
- Full TFM secure services

Table 10. Secure application footprint example

Configuration	Empty secure application template	Limited TFM crypto services only	Full TFM secure services
Security infrastructure	Very basic infrastructure with 1 level of isolation	TFM security infrastructure with 2 levels of isolation.	TFM security infrastructure with 2 levels of isolation.
TFM initial attestation service	No	No	Yes
TFM protected storage service	No	No	Yes (16 Kbytes for NV data)
TFM internal trusted storage service	No	No	Yes (16 Kbytes for NV data)
TFM cryptography services	No	SHA256	All cryptographic algorithms activated by default in the open-source TFM reference implementation: AES all modes, RSA, ECC, HASH.
		AES-GCM	
		ECDSA P256	
Crypto implementation	NA	Hardware crypto used	Hardware crypto used
IDE	EWARM version 9.20.1		
Total size(1)	8 Kbytes	56 Kbytes	136 Kbytes

1. *Size is aligned according to the 8-Kbyte flash memory sector alignment constraints.*

C.3 TFM_Loader memory footprint

The TFM_Loader application delivered as an example in the STM32CubeU5 MCU Package permits the download in the device of new firmware versions using the UART interface with the Ymodem protocol. The TFM_Loader application is optional; it can be fully removed if not needed. Integrators can configure it according to their product specifications and can customize it to support other hardware interfaces or to support other protocols.

The size of the TFM_Loader application can be impacted by the configuration described in Table 11.

Table 11. Firmware loader application configuration options

Items	Configuration possibility	Size impact(1)
Minimal code size	None.	Nonsecure part: 12.2 Kbytes(2)
Number of firmware slots	Primary and secondary slots: the nonsecure loader application writes both the nonsecure application image and the secure application image in the secondary slots located in the nonsecure domain.	None
	Primary slot only: the loader application must integrate a specific secure part to be able to write the secure application in the secure application primary slot located in the secure domain.	Secure part: • EWARM: + 4.3 Kbytes • MDK-ARM: + 4.3 Kbytes • STM32CubeIDE: + 4.9 Kbytes
IDE	Different binary sizes according to the IDE and compiler options used.	Code and IDE dependent
Interface/protocol change	Integrator implementation to enable other loader interfaces or protocols than UART interface with Ymodem protocol.	Undefined

1. *Figures given with EWARM version 9.20.1, MDK-ARM version 5.37.0.0, and STM32CubeIDE version 1.10.0.*

2. *The minimal code size is obtained using MDK-ARM version 5.37.0.0.*

Table 12 describes two examples:

- Single image slot
- Two images slots

Table 12. Firmware loader application footprint example

Configuration	Single image slot	Two images slots
Number of firmware slots	1 (primary slot only)	2
IDEs	EWARM version 9.20.1, MDK-ARM version 5.37.0.0, and STM32CubeIDE version 1.10.0	
Interface/protocol change	UART interface Ymodem protocol	UART interface Ymodem protocol
Total size(1)	Secure • EWARM: 8 Kbytes • MDK-ARM: 8 Kbytes • STM32CubeIDE: 8 Kbytes Nonsecure: • EWARM: 16 Kbytes • MDK-ARM: 16 Kbytes • STM32CubeIDE: 16 Kbytes	Secure • EWARM: 0 Kbyte • MDK-ARM: 0 Kbyte • STM32CubeIDE: 0 Kbyte Nonsecure: • EWARM: 16 Kbytes • MDK-ARM: 16 Kbytes • STM32CubeIDE: 24 Kbytes

1. *Size is aligned according to the 8-Kbyte flash memory sector alignment constraints.*

C.4 TFM_Appli_NonSecure memory footprint

In the case of the internal flash memory usage, the size available for the nonsecure application area depends on the configurations described in Table 13.

Table 13. Nonsecure application configuration options

Items	Configuration possibility	Size impact
Number of firmware images	Single firmware image combining the secure application and the nonsecure application: common image metadata (Header + TLV; refer to Figure 17) for the secure application binary and for the nonsecure application binary	None
	2 firmware images: dedicated image metadata for secure application image and dedicated image metadata nonsecure application image.	+ 2 Kbytes
Image upgrade strategy	Overwrite mode: the slot area is fully available for firmware image.	None
	Swap mode: the last 3 Kbytes of slot area are reserved for the swap process.	+ 3 Kbytes reserved in slot area, for each firmware image
Number of firmware slots	Primary and secondary slots: need space for secondary slots, which are only used to download a new firmware version. Enable the over-the-air download UC from the user application.	None
	Primary slot only: primary slot containing the "active" nonsecure application can be increased as there is no secondary slot.	Nonsecure application size can be doubled
Size of SBSFU application	Refer to TFM_SBSFU_Boot memory footprint.	Refer to TFM_SBSFU_Boot memory footprint
Size of secure application	Refer to TFM_Appli_Secure memory footprint.	Refer to TFM_Appli_Secure memory footprint

Items	Configuration possibility	Size impact
Size of firmware loader application	Refer to TFM_Loader memory footprint.	Refer to TFM_Loader memory footprint
IDE	Different binary size according to the IDE used and its compiler options.	Code and IDE dependent

Table 14 describes the two examples provided in the STM32CubeU5 MCU Package:

- SBSFU example
- Full TFM example

Table 14. Nonsecure application footprint example

Configuration	SBSFU example	Full TFM example
Number of firmware slots	1	2
Secure application	1 level of isolation Basic toggle GPIO	PSA L2 security infrastructure Full TFM secure services (with all cryptographic algorithms activated by default in the open-source TFM reference implementation)
Local loader	Yes (UART/Ymodem protocol)	Yes (UART/Ymodem protocol)
Crypto implementation	Hardware accelerated	Hardware accelerated
IDE	EWARM version 9.20.1	
Max size	Up to 1.9 Mbyte	Up to 728 Kbytes

Appendix D Performance

D.1 TFM_SBSFU_Boot application performance

The TFM_SBSFU_Boot application implements the "Secure Boot" function and the "Secure Firmware Update" function.

The "Secure Boot" function:

- controls the security static protections and sets up the runtime protections.
- configures runtime protections.
- verifies the installed images (integrity check, authenticity check, version control).
- computes specific TFM values.
- launches the execution of the verified images.

The "Secure Boot" function is using cryptographic algorithms, which can be implemented in full software (mbed-crypto software implementation) or can be accelerated with STM32U5 hardware cryptographic accelerators. Table 15 lists the cryptographic algorithms used depending on the configured crypto scheme (refer to Section 5.4 Cryptography operations), and clarifies the ones that can be hardware-accelerated.

Table 15. TFM_SBSFU_Boot cryptographic algorithms

Crypto scheme	Functionality	Algorithm	Implementation
RSA-2048	Image signature verification	RSA-2048	Hardware accelerated
	Image integrity check	SHA256	Hardware accelerated
	Image decryption	AES-CTR-128	Hardware accelerated
	AES-CTR key decryption	RSA-OAEP	Hardware accelerated
RSA-3072	Image signature verification	RSA-3072	Hardware accelerated
	Image integrity check	SHA256	Hardware accelerated
	Image decryption	AES-CTR-128	Hardware accelerated
	AES-CTR key decryption	RSA-OAEP	Hardware accelerated
EC-256	Image signature verification	ECDSA-P256	Hardware accelerated
	Image integrity check	SHA256	Hardware accelerated
	Image decryption	AES-CTR-128	Hardware accelerated
	AES-CTR key decryption	ECIES-P256	Hardware accelerated

The cryptographic algorithms can be configured to use the mbed-crypto software implementation fully instead of the hardware-accelerated version (refer to Hardware-accelerated cryptography in Section 12.1 Configuration).

The “Secure Boot” function execution timing depends directly on the cryptographic algorithms implementation but also on other system parameters such as:

- Hardware configuration
  - STM32U5 hardware-accelerated cryptography capability
  - Core frequency clock (160 MHz maximum)
- Number of firmware images:
  - Single firmware application image combining the secure application and the nonsecure application
  - Two firmware application images: secure application image and nonsecure application image
- Number of data images:
  - None
  - One data image (secure or nonsecure)
  - Two data images (secure and nonsecure)

- Number of firmware slots:
  - Primary and secondary slots: new firmware image can be downloaded by the nonsecure application
  - Primary slot only: active image overwritten, new firmware image can only be downloaded by a standalone loader application
- Firmware images size
- Size of flash memory area used for firmware image version storage
- SBSFU crypto scheme configuration
  - Asymmetric crypto scheme based on RSA or on ECC
  - Firmware encryption support
- SBSFU configuration:
  - TFM support
  - Hash reference support
  - FIH that introduces random delays

Moreover, the “Secure Boot” function execution timing depends on the IDE compiler used (such as EWARM, MDK-ARM, or STM32CubeIDE). In the next sections, performance measurement values are provided based on the three IDEs (EWARM, MDK-ARM, and STM32CubeIDE).

The “Secure Boot” operations consist in:

- SBSFU application initialization:
  - System initialization (CPU at 160 MHz, instruction cache not activated)
  - Peripherals and security protections initialization (cache activated)
  - Flash memory driver initialization
  - Crypto initialization
  - Firmware image version counters consistency and integrity check
- Image integrity check:
  - A hash (SHA256) is computed on the firmware image
- Image version check:
  - The version of the firmware image is compared with the version in the flash memory area reserved for firmware image version storage.
- Image authentication:
  - The signature of the firmware image is authenticated according to an asymmetric cryptographic algorithm
- Image version update:
  - Update the firmware image version with version of the authenticated firmware image
- TFM value computation:
  - A hash (SHA256) of the SBSFU code is computed
- SBSFU application deinitialization
  - Activate runtime protection (HDP) and clean SRAM used by SBSFU application

As illustrated in Figure 84, the “Secure Boot” execution timing is the time between the reset and the launch of the verified image.

Figure 84. Secure Boot execution timing

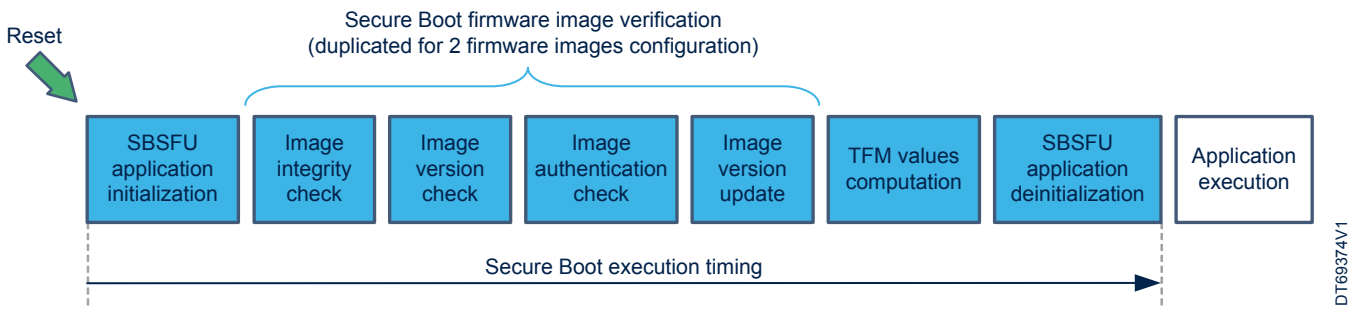


Table 16 provides the different “Secure Boot” operations timings for the following reference configurations:

- Hardware configuration: hardware-accelerated cryptography capability, 160 MHz, instruction cache activated
- Number of firmware images: two firmware images
- Number of firmware slots: primary and secondary slots
- Firmware application image sizes: 51 Kbytes (nonsecure) and 184 Kbytes (secure)
- Firmware data image sizes: 1 Kbyte (nonsecure) and 1 Kbyte (secure)
- Size of the flash memory area used for the storage of the firmware image version: 8 Kbytes
- SBSFU crypto scheme configuration: RSA-2048, firmware encryption support
- SBSFU configuration: TFM support

Table 16 also lists the factors that influence the “Secure Boot” operations timing, and gives some performance indications for various configurations.

Table 16. “Secure Boot” operations timing indications

Operation name	Timing influencing factor	Timing for reference configuration(1)			Fluctuation versus reference configuration(1)
		EWARM (version 9.20.1)	MDK-ARM (version 5.37.0.0)	STM32CubeIDE (version 1.10.0)	
SBSFU application initialization	• Size of RAM used by SBSFU application • Size of flash memory area used for firmware image version storage • Number of firmware images	1.9 ms	1.6 ms	6.4 ms	-
Image integrity check	• Image size	8.2 ms (184 Kbytes) 2.3 ms (51 Kbytes)	8.7 ms (183 Kbytes) 2.4 ms (34 Kbytes)	7.6 ms (183 Kbytes) 2.1 ms (41 Kbytes)	% image size
Image version check	• Size of flash memory area used for firmware image version storage	0.6 ms	0.6 ms	0.6 ms	-
Image authentication check	• Crypto scheme configuration • STM32U5 hardware-accelerated cryptography capability	5 ms (RSA-2048 hardware) per image(2)	5 ms (RSA-2048 hardware) per image(2)	5 ms (RSA-2048 hardware) per image(2)	RSA-3072 hardware: + 7 ms ECDSA-256 hardware: + 14 ms RSA-2048 software: • EWARM: 23 ms • MDK-ARM: 18 ms • STM32CubeIDE: 13 ms RSA-3072 software: • EWARM: 57 ms • MDK-ARM: 52 ms • STM32CubeIDE: 32 ms ECDSA-256 software: • EWARM: 290 ms • MDK-ARM: 253 ms • STM32CubeIDE: 278 ms
Image version update	• Size of flash memory area used for firmware image version storage	0.5 ms	0.5 ms	0.5 ms	-
TFM value computation	• Only applicable when using "TFM" secure application configuration • Size of SBSFU application	2.2 ms	2.4 ms	2.2 ms	% SBSFU application code size

Operation name	Timing influencing factor	Timing for reference configuration(1)			Fluctuation versus reference configuration(1)
		EWARM (version 9.20.1)	MDK-ARM (version 5.37.0.0)	STM32CubeIDE (version 1.10.0)	
SBSFU application deinitialization	• Size of RAM used by SBSFU application	1.4 ms	1.4 ms	1 ms	% RAM size used by SBSFU application

1. *The crypto operation timing slightly fluctuates according to the key value.*
2. *Applicable at the first boot or after an image download (the hash reference makes this operation on subsequent calls unnecessary).*

Table 17 gives some “Secure Boot” execution timing values for a set of configurations.

Table 17. “Secure Boot” execution timing value indications

Configuration description(1)	Secure Boot execution timing(2) (3)
Configuration -1- • 160 MHz, instruction cache activated on internal memories • **2 application images (184 Kbytes, 51 Kbytes)** • No data image • 2 slots • RSA-2048 **with hardware-accelerated cryptography capability** • **TFM configuration**	• EWARM: 19 ms • MDK-ARM: 19 ms • STM32CubeIDE: 21 ms
Configuration -2- • 160 MHz, instruction cache activated on internal memories • **1 firmware image (76 Kbytes)** • No data image • 2 slots • RSA-2048 **with software cryptography** • **SBSFU configuration (no TFM secure services)**	• EWARM: 8.5 ms • MDK-ARM: 8.2 ms • STM32CubeIDE: 10.7 ms

1. *Configuration differences are highlighted in bold.*
2. *The crypto operation timing slightly fluctuates according to the key value.*
3. *Figures given with EWARM version 9.20.1, MDK-ARM version 5.37.0.0, and STM32CubeIDE version 1.10.0.*

D.2 TFM cryptographic performance

The TF-M framework embeds a large set of cryptographic algorithms, which can be implemented in full software (mbed-crypto software implementation) or can be accelerated with STM32U5 hardware cryptographic accelerators. Several cryptographic algorithms are embedded in the source code files but not all of them are activated. Table 18 lists the cryptographic algorithms activated by default and states the ones that can be hardware accelerated.

Table 18. TFM runtime cryptographic algorithms activated by default

Functionality	Algorithm	Key size	Mode	Implementation
Hash algorithms	SHA1	-	-	Hardware accelerated
	SHA224 / SHA256	-	-	Hardware accelerated
	SHA384 / SHA512	-	-	mbed-crypto software
Symmetric algorithms	AES	128 256	CBC	Hardware accelerated
			CTR	Hardware accelerated
			GCM (aead)	Hardware accelerated
			CCM (aead)	Hardware accelerated
			CFB	Hardware accelerated

Functionality	Algorithm	Key size	Mode	Implementation
Asymmetric algorithms	RSA (PKCS#1 v1.5)	1024 2048 3072	-	Hardware accelerated
	RSA (PKCS#1 v2.1)		-	Hardware accelerated
	ECDH or ECDSA	192 224 256 384 512 521	Curves: secp192r1, secp224r1, secp256r1, secp384r1, secp521r1, secp192k1, secp224k1, secp256k1, bp256r1, bp384r1, bp512r1	Hardware accelerated
			Curves: 25519, 448	mbed-crypto software
Key generation and derivation	RSA key gen	1024 2048 3072	-	Hardware accelerated
	EC key gen	192 224 256 384 512 521	Curves: secp192r1, secp224r1, secp256r1, secp384r1, secp521r1, secp192k1, secp224k1, secp256k1, bp256r1, bp384r1, bp512r1	Hardware accelerated
			Curves: 25519, 448	mbed-crypto software

The TFM runtime cryptographic algorithms can be disabled through compile switches in `Projects\B-U585I-IOT02A\Applications\TFM\TFM_Appli\Secure\Inc\tfm_mbedcrypto_config.h` (such as `MBEDTLS_SHA1_C`, `MBEDTLS_GCM_C`, `MBEDTLS_ECDSA_C` and others). The cryptographic algorithms can be configured to fully use mbed-crypto software implementation instead of the hardware-accelerated version (refer to Hardware-accelerated cryptography in Section 12.1 Configuration).

For the hardware-accelerated versions of cryptographic algorithms, the ciphering operations are performed using the SAES peripheral, which is protected against side-channel and timing attacks. It is possible to use the AES peripheral instead of the SAES peripheral to achieve better performance, by disabling `HW_CRYPTO_DPA_AES` and `HW_CRYPTO_DPA_GCM` compile switches in `Projects\B-U585-IOT02A\Applications\TFM\TFM_Appli\Secure\Inc\tfm_mbedcrypto_config.h`.

Note: Some cryptographic algorithms may not be secure enough for some type of operations (for instance SHA1 may only be accepted for checksum and data integrity). The integrator must use the right cryptographic algorithms according to the product security requirements.

Table 19 lists the cryptographic algorithms embedded in the source code that are not activated.

Table 19. Cryptographic algorithms present but not activated

Functionality	Algorithm	Status
Hash algorithms	ripemd160	Not activated
	md5	Not activated
	md4	Not activated
	md2	Not activated
Symmetric algorithms	des	Not activated
	t-des	Not activated
	blowfish	Not activated
	camellia	Not activated
	arc4	Not activated
	chacha20	Not activated
	aria	Not activated
Cipher block modes and aead	arc4 stream	Not activated
	chacha20-poly1305 (aead)	Not activated

As an indication, some performance measurements are provided in Table 20 for some TFM cryptographic services with and without hardware accelerators usage.

The timings are measured at TFM nonsecure side, when calling PSA APIs with ICACHE enabled on the internal flash memory. They are measured in number of cycles and microseconds, assuming an STM32U5 system clock at 160 MHz.

Table 20. Performance for cryptographic TFM runtime services

PSA service (called from the TF-M nonsecure application)	Hardware accelerated			mbed-crypto software		
	EWARM (version 9.20.1)	MDK-ARM (version 5.37.0.0)	STM32CubeIDE (version 1.10.0)	EWARM (version 9.20.1)	MDK-ARM (version 5.37.0.0)	STM32CubeIDE (version 1.10.0)
Initial attestation (including ECDSA signature)						
psa_initial_attest_get_token[1] (544 bytes)	6 766 209 cycles 42 288 µs	6 666 116 cycles 41 663 µs	6 728 874 cycles 42 055 µs	43 711 844 cycles 273 199 µs	43 711 844 cycles 273 199 µs	41 374 101 cycles 258 588 µs
psa_initial_attest_get_token[2] (544 bytes)	3 364 397 cycles 21 027 µs	3 298 707 cycles 20 616 µs	3 344 394 cycles 20 902 µs	22 427 211 cycles 140 170 µs	22 530 932 cycles 140 818 µs	21 290 114 cycles 133 063 µs
AES-CBC - 128-bit key						
psa_cipher_update (1 392 bytes)	SAES: 256 290 cycles 1 601 µs AES: 113 009 cycles 706 µs	SAES: 245 906 cycles 1 536 µs AES: 94 275 cycles 589 µs	SAES: 254 221 cycles 1 588 µs AES: 124 774 cycles 779 µs	210 548 cycles 1 315 µs	210 548 cycles 1 315 µs	215 759 cycles 1 348 µs
SHA256						
psa_hash_update (1 400 bytes)	79 762 cycles 498 µs	74 258 cycles 464 µs	94 675 cycles 591 µs	78 175 cycles 488 µs	78 175 cycles 488 µs	85 138 cycles 532 µs

1. *With the MCU: the performance measurements provided for this TFM runtime service are applicable for the second and subsequent calls to this PSA API. The first call to this service lasts longer, as the EAT public key must be computed first, from the provisioned EAT private key.*
2. *With STSAFE: the performance is measured on a nonsecure communication channel (no data encryption, no MCU Command-MAC, and no STSAFE Response-MAC).*

Appendix E Troubleshooting

Table 21 provides some troubleshooting guidelines for some common problems.

Table 21. Troubleshooting

Problem	Possible solution
`Regression.bat` script failure (`DEV_CONNECT_ERR`)	The device may be in freeze state due to intrusion detection. Recover from intrusion detection with the procedure described in Section 10.5.3 ST-LINK disable (IDD jumper off and on).
No logs on the terminal after the device programming	Same as above.
No logs on the terminal when ST-LINK USB is connected to a board programmed with TFM	Same as above.
The boot is frozen with the following log on the terminal (in development mode): `Boot with TAMPER Event Active`	Plug the tamper cable between the tamper pins on the B-U585I-IOT02A or the STM32U5A9J-DK board, then reboot. Otherwise, disable the external tamper protection (by modifying the `TFM_TAMPER_ENABLE` flag value in `boot_hal_cfg.h`), then build and program again.
Error during the postbuild step when building	Check the postbuild logs in the file `output.txt` to get information on the error reason.

Revision history

Table 22. **Document revision history**

Date	Revision	Changes
25-Jun-2021	1	Initial release.
9-Mar-2022	2	TFM application update: • Added STSAFE for the EAT signature and device authentication (personalization profile) in *Section 8.1 TFM application description*, *Section 8.2.6 STSAFE*, *Section 8.4 Folder structure*, *Section 11.3 Test TFM*, and *Section 12.1 Configuration* • Added TRNG-based random delay for the FIH in *Section 7 Protection measures and security strategy* and *Section 8.1 TFM application description* • Added data images, which can be provisioned during manufacturing and downloaded like firmware images in *Section 8.1 TFM application description*, *Figure 7* to *Figure 10*, *Figure 25*, and *Section 12.1 Configuration* • Added hash references for boot time improvement in *Section 8.1 TFM application description* • Added OEM2 default provisioned password for RDP regression in *Figure 35* • Updated for TF-M v1.3.0 in *Figure 1*, *Section 6 Secure services at runtime*, and *Section 6.5 Firmware update service* • Updated the image trailer size computation details in *Section 8.3.1 Flash memory layout* and *Figure 17* • Updated the SRAM layout during application execution in *Figure 18* and *Figure 19* • Updated the BL2 shared data content in *Section 8.3.2 SRAM layout* and *Section 12.1 Configuration* • Replaced the secure storage service (SST) with the protected storage service (PS) across the whole document The document applicability is extended to microcontrollers in the STM32U5 series with 4-Mbyte flash memory: • Updated the memory layout in *Section 8.3 Memory layout* • Updated the memory footprints and performance figures in *Appendix B Memory footprint* and *Appendix C Performance* No support of MDK-ARM and STM32CubeIDE in this revision.
5-May-2023	3	Extended the document scope to the microcontrollers with 512 Kbytes of flash memory in the STM32U5 series: • Updated *Section 8.1 TFM application description* for the applicability to the 512-Kbyte devices • Updated *Figure 7*, *Figure 8*, *Figure 9*, and *Figure 10* in *Section 8.3.1 Flash memory layout* • Updated *Figure 18* and *Figure 19* in *Section 8.3.2 SRAM layout* • Updated *Figure 82* in *Appendix B.2 SRAM protections* Extended the compatible IDEs to Keil® MDK-ARM and STMicroelectronics STM32CubeIDE: project files updated across the whole document, footprint figures updated in *Appendix C Memory footprint*, and performance figures updated in *Appendix D Performance*. Added STM32U5A9J-DK and NUCLEO-U545RE-Q to the set of available development boards: • Added *Appendix A Development hardware boards* • Updated *Section 9.1 Hardware setup* and *Section 10.5.3 ST-LINK disable*

Date	Revision	Changes
27-Oct-2023	4	Added STM32U5G9J-DK2 to the set of available development boards: • Updated Section 1.1 Applicable products and default examples and Appendix A Development hardware boards • Updated Section 9.1 Hardware setup • Updated the compatibility with 4-Mbyte STM32U5Fxxx/Gxxx devices in Figure 18. STM32U5 user SRAM mapping (1 of 2), Figure 19. STM32U5 user SRAM mapping (2 of 2), and Figure 83. SRAM protections overview Updated the software setup to STM32CubeU5 v1.3.0 in Section 9.2.1 STM32CubeU5 MCU Package. Updated the description of the option bytes WRP2A and WRP2B in the *Step 2.2* of Section 10.2 STM32U5 device initialization.

Contents

List of tables

List of figures