Secure system configuration in XMC7000 and XMC5000 MCUs

About this document

Scope and purpose

This document explains the requirements to configure a secure system from the boot process to application software execution in the XMC7000/XMC5000 family.

Associated part family

XMC7000 family XMC7100/XMC7200 series.
XMC5000 family XMC5100/XMC5200/XMC5300 series.

Table of contents

About this document . . . 1
Table of contents . . . 2
1 Introduction . . . 4
2 What is a secure system? . . . 5
3 Basic definitions . . . 6
4 Lifecycle stages . . . 9
4.1 NORMAL_PROVISIONED . . . 9
4.2 SECURE . . . 9
4.3 SECURE_W_DEBUG . . . 9
4.4 RMA . . . 10
4.4.1 Requirements for transition to RMA lifecycle stage . . . 11
4.4.1.1 TransitiontoRMA API . . . 11
4.4.1.2 OpenRMA API . . . 11
4.5 CORRUPTED . . . 12
5 Boot sequence and chain of trust . . . 13
5.1 Boot sequence . . . 13
5.2 Chain of Trust (CoT) . . . 14
6 Code signing and validation . . . 15
6.1 Code signing . . . 15
6.2 Code validation . . . 16
7 Resource protection . . . 18
7.1 Boot protection . . . 18
7.2 Application protection . . . 18
7.3 Security-enhanced PPU configuration . . . 19
7.4 Debug and test access port restriction . . . 20
7.4.1 Minimum requirements for a system call initiated by a debugger . . . 24
8 Configure a secure system . . . 25
8.1 TOC2 . . . 26
8.1.1 Configuration . . . 30
8.2 User application block . . . 31
8.3 Secure boot RSA public key format . . . 33
9 Appendix A - Example of creating public and private keys . . . 36
9.1 Edge protect tools . . . 36
9.2 Python scripts . . . 37
9.2.1 Additional tools required . . . 37
9.2.2 Scripts . . . 37
9.2.3 Running the scripts . . . 42

Table of contents

9.3 Installing the public key . . . 43

10 Appendix B - Requirements for generating a digital signature . . . 48

11 Appendix C - Authentication of the main user application . . . 49

12 Appendix D - Read device Unique ID . . . 52

13 Appendix E - Transition to SECURE/SECURE_WITH_DEBUG lifecycle stage . . . 54

14 Appendix F - Transition to RMA lifecycle stage . . . 55

14.1 Generate certificate . . . 59

15 Appendix G - Configure application protection . . . 61

15.1 Configuration . . . 61

16 Appendix H - Normal access restriction . . . 65

16.1 Configuration . . . 65

References . . . 66

Revision history . . . 67

Disclaimer . . . 68

1 Introduction

This application note discusses how to make sure that the system executes code only from a trusted source and how to configure a secure embedded system using XMC7000/XMC5000 MCUs. You will learn about the boot process and how it pertains to a secure system.

This is an advanced application note and assumes that you are familiar with the basic XMC7000/XMC5000 architecture (see the device datasheet [1] or the architecture reference manual [2]).

There are many reasons why a system must be secure: manufacturers want to protect their IP to maintain their market, or to protect the end user from dangerous operations caused by a malicious attack from a third party.

There are three main ways products can be hacked:

- **Direct access to the debug port:** XMC7000/XMC5000 is based on the Arm® architecture; accessing or reprogramming firmware or examining the internal data is easy with the use of a common debugger. Hacking or reverse engineering a product is easy if the device is left unsecured
- **Direct connection to a communication port such as SPI, I²C, CAN, or a UART:** These ports are used for communication between ECUs, or between the ECU and other components in a car. It is also used for ECU software update or information acquisition. This connection may allow firmware to be read or updated with non-sanctioned access if the device is left unsecured
- **Wireless connection such as OTA:** This has become the standard method of hacking because it does not require physical contact. Perpetrators can access the car to perform unauthorized firmware update and control from anywhere if the device is left unsecured

***Note**:*

- *All security-related features offered by the XMC7000/XMC5000 MCUs are intended to provide protection only against logical attacks. These are attacks which use regular interfaces to communicate in an unexpected way with the device and can be performed even remotely. Typical examples are buffer overflow, authentication bypass, malicious software injection, misuse of debug functions or interfaces, etc*
- *Physical attacks are considered out of scope and hence, the XMC7000/XMC5000 MCUs do not contain any hardware countermeasures against non-invasive, semi-invasive, or invasive attacks. Countermeasures against side-channel attacks and fault injection attacks must be implemented by software*

2 What is a secure system?

The definition of a secure system may differ depending on the application. Some systems require that all access to the device is blocked, but others just need to verify that the firmware has not been corrupted. The XMC7000/ XMC5000 MCU allows you to define the security level required for the project. There is no perfect method because every project has different requirements. The following is a list of projects with different security goals:

- **Trusted firmware updates only with a hardware debugger:** This is usually not considered as a secure system, but if the hardware is installed such that third parties cannot get direct access, it may be secure
  - Flash write commands from firmware can be disabled so no internal hack can change or replace the application
  - The device can be put in a secure mode with the debug port open, which will force the firmware to be authenticated with a public key each time the device comes out of reset
- **No access to debug port; support firmware updates:** The debug port provides access to all memory and a method for the device to be reprogrammed. In most cases, a real secure system requires that the debug port is disabled. Then, the only way to update the firmware is for the application to provide a way to download new program data with some type of communication port such as UART, I^2C, CAN, or SPI. The designer decides how secure this communication port must be
- **Lock down firmware; no updates:** This means that the debug port is disabled and there is no provision for bootloading. This may be the most secure, but there is no way to perform bug fixes or add future enhancements
- **Trusted firmware updates; protect IP:** To fully protect the IP, the debug port must be disabled. Because the debug port is disabled, the user must provide a method to load new firmware with a bootloader. This is typically implemented with a serial port such as UART, SPI, or I^2C. Because the IP must be protected, the bootloader must encrypt the data transferred. The XMC7000/XMC5000 MCU includes a crypto block to help accelerate encryption and decryption. Security keys installed in the device at the factory can be used to authenticate the code and decrypt the data transferred during the bootload process

3 Basic definitions

This section describes some terms that will be used throughout this document.

- **Application flash (User):** This is the flash memory that is used to store your application code
- **Chain of Trust (CoT):** The root of trust begins with the code residing in the ROM, which cannot be altered. The Chain of Trust is established by validating the blocks of software before the execution starting from the root of trust located in the ROM
- **Cipher-based message authentication code (CMAC):** Message authentication code algorithm based on a block cipher. (e.g., AES)
- **CySAF:** Infineon secured application format
- **Debug access port (DAP):** Interface between an external debugger/programmer and XMC7000/XMC5000 MCU for programming and debugging. This allows the connection to one of the three access ports (AP), CM0_AP, CM7_AP or CM4_AP, and System_AP. The System_AP can only access SRAM, flash, and MMIOs, not the CPUs
- **DAP access restriction:** This determines the debug port access restrictions, and has three states corresponding to the protection state: Normal, Secure, and Dead. Each of these states may be configured by the user. See Security-enhanced PPU configuration for the storage location of access restriction in each protection state
- **Digital digest/signature:** The signature generated by the SHA-256 function that operates on a block of data
- **Electronic control unit (ECU):** Unit for controlling the system using electronic circuits; mainly mounted on automobiles. There are various ECUs depending on the application, such as body control, engine control, and brake control
- **eFuse:** One-time programmable (OTP) memory, that by default is '0' and can be changed only from '0' to '1'. The eFuse bits can be programmed individually, but cannot be erased
- **eFuse read protection unit (ERPU):** This is part of SWPU (software protection unit; see definition below). ERPUs are used to implement read access restrictions to eFuse
- **eFuse write protection unit (EWPU):** This is part of SWPU. EWPUs are used to implement write access restrictions to eFuse
- **Flash boot:** This is part of the boot system that performs two basic tasks:
  - Sets up the debug port based on the lifecycle stage
  - Validates the user application before executing it
- **Flash write protection unit (FWPU):** This is part of SWPU. FWPUs are used to implement write access restrictions to flash
- **Hash:** A crypto algorithm that generates a repeatable but unique digest for a given block of data. This function is non-reversible
- **IP**: Intellectual property. This can be both code and data stored in a device
- **IPC:** Inter-processor communication hardware used to facilitate communication between the two CPU cores
- **Lifecycle:** Security mode in which the device is operating. XMC7000/XMC5000 MCU has five stages NORMAL_PROVISIONED, SECURE, SECURE_W_DEBUG, RMA, and CORRUPTED. To the user, it has only three states of interest: NORMAL_PROVISIONED, SECURE, and SECURE_W_DEBUG
- **Main user application:** This is part of the user application that is not authenticated by flash boot. It is mainly executed by the CM7 or CM4 CPU. For CoT, it needs to be authenticated with the secure image
- **Memory protection unit (MPU):** MPU is used to isolate memory sections from different software components executed on the same CPU. MPU is bus-master-specific
- **MMIO:** Memory-mapped input/output; usually refers to registers that control the hardware I/O

3 Basic definitions

- **Non-secure (NS):** NS is a protection attribute used to distinguish between secure and non-secure accesses. In non-secure setting, secure access is also allowed. The NS attribute is allowed and restricted by SMPU, PPU, and SWPU
- **Over-the-air (OTA):** OTA refers to the transmission and reception of data via wireless communication
- **Protection context (PC):** PC can apply different protection attributes to bus master access without changing the setting of protection units. The PC attribute is allowed and restricted by SMPU, PPU, and SWPU. Although PC most often refers to a program counter, in this document, it refers to the protection context
- **Peripheral protection unit (PPU):** PPUs are used to restrict access to a peripheral or set of peripherals to only one or a specific set of bus masters
- **Protection state:** There are Normal, Secure, Dead, and Virgin states depending on the lifecycle stage. ROM boot deploys access restrictions according to the protection status
- **Protection units:** These are four types of protection units: memory protection unit (MPU), shared memory protection unit (SMPU), peripheral protection unit (PPU), and software protection units (SWPU). MPU, SMPU, and PPU are implemented by hardware; SWPU is implemented by software
- **Public-key cryptography (PKC):** Otherwise known as asymmetrical cryptography. Public-key cryptography is an encryption technique that uses a paired public and private key (or asymmetric key) algorithm for secure data communication. It is used to decode a message or block of data. The private key is used to decrypt the data and must be kept secured, while the public key is used to encrypt the data but can be disseminated widely.
  - **Public key:** The public key can be shared, but it should be authenticated or secured so it cannot be modified
  - **Private key:** The private key must be kept in a secure location so it cannot be viewed or stolen. It is used to decrypt a block of data that has been encrypted using an associated public key
- **RMA:** Returned material authorization
- **ROM:** Read-only memory that is programmed as part of the fabrication process and cannot be reprogrammed
- **RSA-nnnn:** An asymmetric encryption system that uses two keys. One key is private and should not be shared and the other is public and can be read without loss of security. The encryption/decryption is controlled by a key that is commonly 2048 bits, 3072 bits, or 4096 bits in length (RSA-2048, RSA-3072, or RSA-4096)
- **Secure image:** Software that is used to set up the security features such as HSM firmware of XMC7000/ XMC5000 MCU; it is mainly executed by CM0+. It can be modified by the programmer to implement a specific security policy
- **Security policy:** This is the set of rules that the designer imposes to determine what resources are protected from outside tampering or between the internal CPUs
- **Serial memory interface (SMIF):** An SPI (serial peripheral interface) communication interface to serial memory devices, including NOR flash, SRAM, and non-volatile SRAM
- **SFlash:** Supervisor flash memory. This memory partition in flash contains several areas that include system trim values, flash boot executable code, public key storage, etc. After the device transitions into a SECURE lifecycle stage, it can no longer be changed
- **SHA-256:** A cryptographic hash algorithm used to create a digest for a block of data or code. This hash algorithm produces a 256-bit unique digest of the data no matter the size of the data block
- **Shared memory protection unit (SMPU)**: SMPUs are used to allow access to a specific memory space (flash, SRAM, or registers) to only one or a specific set of bus masters
- **Software protection units (SWPU):** SWPUs are used to implement access restrictions to flash write, and eFuse read and write

3 Basic definitions

- **System calls:** Functions such as flash write functions that are executed by the Arm® Cortex® M0+ CPU (CM0+) from ROM
- **TOC1:** This is an area in the SFlash that is used to store pointers to the trim values, flash boot entry points, etc. It is used only by boot code in the ROM and is not editable by the designer
- **TOC2:** This is an area in the SFlash that is used to store pointers to two applications blocks: secure image and main user application. It also contains some boot parameters that are settable by the system designer

4 Lifecycle stages

The device lifecycle is a key aspect of the XMC7000/XMC5000 MCU security. Lifecycle stages follow a strict irreversible progression dictated by blowing eFuses (changing a fuse's value from '0' to '1'). This system is used to protect the internal device data and code at the level required by the customer. Figure 1 shows the XMC7000/ XMC5000 MCU supported lifecycle stages.

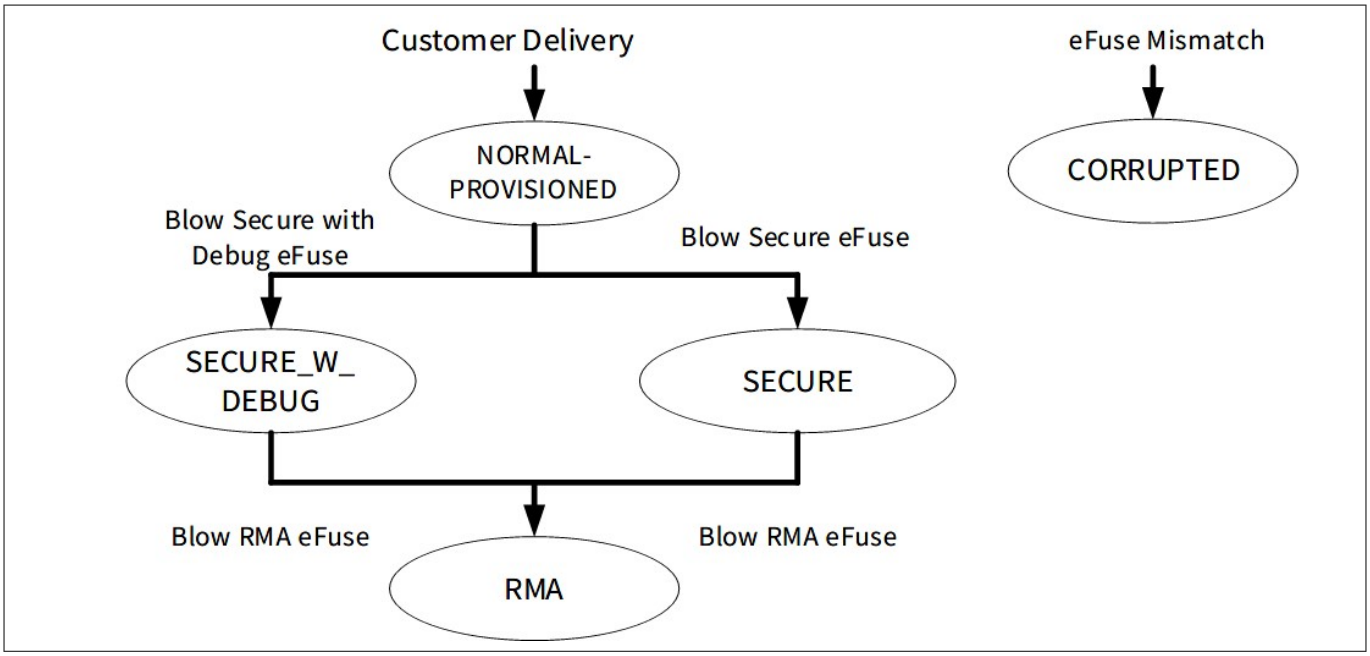


Figure 1 Device lifecycle stages

4.1 NORMAL_PROVISIONED

This is the lifecycle stage of a device after trimming and testing is complete in the factory. All configuration and trimming information are complete. Valid flash boot code has been programmed in the SFlash. To allow the OEM to check the data integrity of trims, flash boot, and other objects from the factory, a hash (SHA-256 truncated to 128 bits) of these objects is stored in eFuse. This hash is referred to as FACTORY_HASH. Customers receive parts in this stage.

4.2 SECURE

This stage is the lifecycle stage of a secure device. Before transitioning to this stage, the SECURE_HASH must have been blown in the eFuse and a valid application code must have been programmed in the code flash. In this stage, the protection state is set to SECURE and SECURE access restrictions are deployed. A SECURE device will boot only when authentication of its flash boot and application code succeeds. The SECURE_HASH is calculated and written to the eFuse by the SROM firmware when transitioning to the SECURE or SECURE_W_DEBUG lifecycle stage from the NORMAL_PROVISIONED lifecycle stage.

***Note**: Lifecycle stage is irreversible. Therefore, it is recommended that the programming process is evaluated and the application program is authenticated before transitioning to SECURE lifecycle stage.*

4.3 SECURE_W_DEBUG

This stage is the same as the SECURE lifecycle stage, except that the device allows debugging. Before transitioning to this stage, the SECURE_HASH must have been blown in the eFuse and a valid application code

must have been programmed in the code flash. In this stage, the protection state is set to SECURE, but NORMAL access restrictions are deployed to enable debugging. When there is an authentication failure during ROM boot or flash boot, the protection state is set to “SECURE” but NORMAL access restrictions are deployed, SWD/JTAG pins are enabled, and is not transitioned to “DEAD” state. Transition from SECURE_W_DEBUG to SECURE is not allowed. SECURE_W_DEBUG parts are used only by the developers and software testers. Devices are not shipped in this lifecycle stage.

4.4 RMA

This stage allows one to perform failure analysis (FA). The customer transitions the part to the RMA lifecycle stage when the customer wants to perform failure analysis on the part. The customer erases all the sensitive data before invoking the `TransitiontoRMA` API system call that transitions the part to RMA.

When invoking the system call to transition to RMA, the customer must create a certificate that authorizes to transition the part with a specific Unique ID to the RMA lifecycle stage. The certificate will be signed by the customer using the same private key that is used for signing the user application image. The verification of the signature uses the same algorithm used by flash boot in authenticating the user application. The same public key (injected by the OEM) stored in the SFlash is used for the verification.

When a part is reset in the RMA lifecycle stage, the boot will set the access restrictions such that the DAP has access only to System AP and IPC MMIO registers for making system calls, and adequate RAM for communication. It will then wait for the system call (`OpenRMA`) from the DAP along with the certificate of authorization. The boot process will not initiate any firmware until it successfully executes the `OpenRMA` API. After the command is successful, the lifecycle stage stored in the eFuse cannot be changed from RMA. Every time the part is reset, it must execute the `OpenRMA` API successfully before the part can be used. To execute this API, the customer must create the certificate signed using their private key. See Appendix F - Transition to RMA lifecycle stage for transition to the RMA lifecycle stage.

***Note**: The device with a DEAD protection state or CORRUPTED lifecycle stage cannot transition to the RMA lifecycle stage.*

***Note**: Transition to RMA is possible only from the SECURE or SECURE_W_DEBUG lifecycle stage. Therefore, the device must transition to SECURE or SECURE_W_DEBUG and then to RMA if it is in the NORMAL_PROVISIONED lifecycle stage.*

***Note**: An RSA key pair (private key and public key) is required to run the TransitiontoRMA and OpenRMA APIs. The private key is used to create the certificate and the public key is used to authenticate the certificate. The public key must be written to the device in advance. However, if the device was transitioned to the RMA lifecycle stage from the SECURE_WITH_DEBUG lifecycle stage, then OpenRMA is skipped, and the device does not wait for OpenRMA execution. Hence full access to the device will be unlocked without the certificate and the user application will be executed*

***Note**: ECC errors might occur during the execution of the TransitiontoRMA and OpenRMA APIs. Therefore, the user software should be careful in handling failures during the TransitiontoRMA API execution and after the OpenRMA API execution. See Requirements for transition to RMA lifecycle stage for more details.*

If the device transitions to a DEAD protection state due to an unintended authentication failure, such as hardware failure or sensitive data erase, and cannot transition to the RMA lifecycle stage, you may be able to solve it by preparing a second application software to only allow transition to the RMA lifecycle stage. The start address is defined in TOC2. If the first application software fails or erases all of the first application, Flash Boot can run the second application. However, if the authentication of the second application software is also corrupted, the device cannot transition to the lifecycle stage. The second application software could be

exclusively used for RMA management. It is recommended that you locate the second application software in a different sector from the first application software, so that only the first application can be erased.

However, if the device meets the bootloader enabling conditions, the second application cannot be activated when all of the first application has been erased. See the TOC2 for bootloader enabling conditions.

If all of the first application is erased, it can transition to DEAD state. There are two options:

- If the second application is not implemented:
  - Prepare and reprogram the signed dummy code and digital signature, instead of the first application software, after erasing. This dummy code can be exclusively used for RMA management
- If the second application is implemented:
  - Prepare and reprogram the dummy application header, instead of the first application software, after erasing
  - Disable the bootloader (TOC2_FLAGS.FB_BOOTLOADER_CTL=0x2). It is available only in the NORMAL_PROVISIONED lifecycle stage

4.4.1 Requirements for transition to RMA lifecycle stage

This section describes the conditions for the TransitiontoRMA and OpenRMA system call APIs execution used to transition to the RMA lifecycle stage.

4.4.1.1 TransitiontoRMA API

ECC errors might occur during the `TransitiontoRMA` API execution. Therefore, the user software should not configure the fault structure for Crypto and SRAM0 ECC errors before triggering the API. Otherwise, the software will ignore the ECC faults reported during the execution. The following failures must be masked:

- (Fault number in XMC7100/XMC7200 series):
  - CPUSS_RAMC0_C_ECC (Fault number = 58)
  - CPUSS_RAMC0_NC_ECC (Fault number = 59)
  - CPUSS_CRYPTO_C_ECC (Fault number = 64)
  - CPUSS_CRYPTO_NC_ECC (Fault number = 65)

See the device-specific datasheet [1] for the fault number.

In addition, the device in the RMA lifecycle stage requires the OpenRMA API on every reset. In the case of a device-specific failure, such as a hardware failure, the fault report triggers a reset. After the OpenRMA execution, the device cannot open RMA. There are two options before triggering the `TransitiontoRMA` API:

- Mask the device-specific failure that triggers a reset in advance. Also, mask the application software that runs after OpenRMA
- Reprogram the dummy code that performs only RMA management

Certificate and digital signature must be written to the SRAM to run the TransitiontoRMA API. When using the TransitiontoRMA API to move a device to the RMA lifecycle stage, the parameters, such as the certificate and digital signature, must be placed from [System RAM0 start address + 4KB]. See Appendix F - Transition to RMA lifecycle stage for details of the certificate and digital signature.

4.4.1.2 OpenRMA API

ECC errors might also occur during the OpenRMA API execution. Therefore, the software that runs after the OpenRMA API should not configure the fault structure for Crypto and SRAM0 ECC errors. Otherwise, the software ignores the reported ECC faults. See TransitiontoRMA API for the failures that must be masked.

The protection state of the RMA lifecycle stage indicates VIRGIN. Therefore, software can know if the device is in the RMA lifecycle stage by using the CPUSS_PROTECTION register. See the Registers TRM [2] for more details.

The certificate and digital signature must be written to the SRAM to run the OpenRMA API. After the device transitions to the RMA lifecycle stage, the Sys-DAP can only access IPC MMIO and 1/16 of the System RAM0 by using the Sys-DAP MPU. When using the OpenRMA API, the parameters, such as the certificate and digital signature, must be placed as follows:

- Devices with SRAM 0 size larger than 64 KB: Place the parameters from [System RAM0 start address + 4KB] to [System RAM0 start address + 1/16 of System RAM0 size]
- Devices with SRMA0 of 64 KB or less: Place the parameters within 600 bytes from [System RAM0 start address + 2KB]. The certificate and signature address are 24 bytes, and the digital signature is 512 bytes (RSA-4K)

See Appendix F - Transition to RMA lifecycle stage for details of the certificate and digital signature.

See the device-specific datasheet [1] for the fault number and System RAM0 size.

4.5 CORRUPTED

The device is in this lifecycle stage if a read error is detected when reading the eFuse bits that determine the lifecycle stage. The device will enter the DEAD protection state and only the IPC MMIOs can be read via the SYS-AP. No other accesses are allowed.

5 Boot sequence and chain of trust

The Chain of Trust (CoT) is inherently a part of the boot sequence. It begins at the root of trust (RoT), which is the initial boot code stored in the ROM, and which cannot be changed.

5.1 Boot sequence

The boot sequence and the validation sequence are, for the most part, one and the same. Figure 2 shows how the CM0+ operation starts from reset. After reset, CM0+ starts executing from ROM boot. ROM boot validates the SFlash. After validation of the SFlash is complete, execution jumps to the flash boot and configures the DAP as required by the protection state. Notice the color coding that depicts the memory type where the data and code resides.

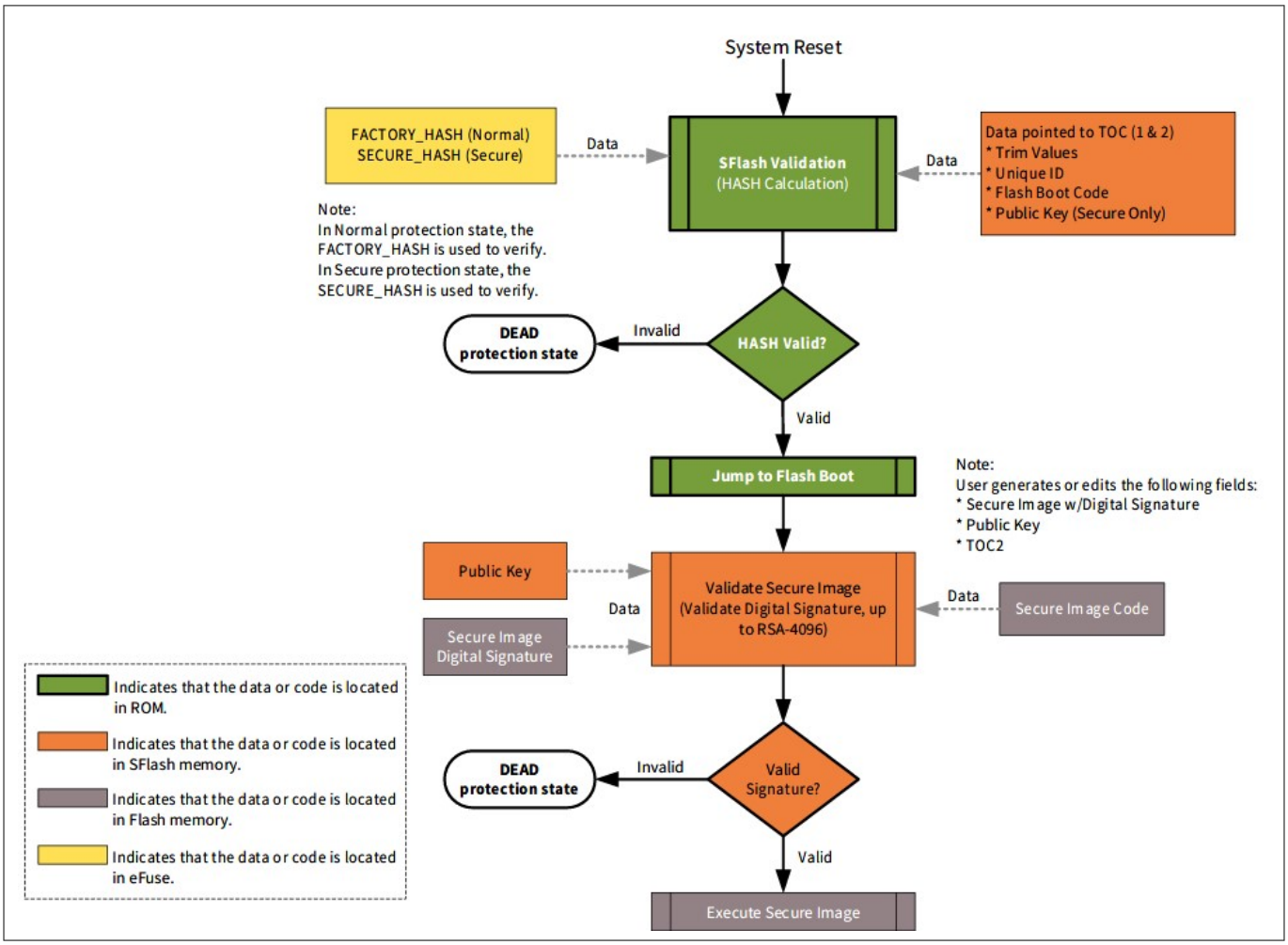


Figure 2 XMC7000/XMC5000 MCU boot sequence with CoT

Flash boot then validates the first application listed in TOC2 and jumps to its entry point if validated. In the secure system defined in this application note, the first user application is the secure image. After the secure image configures the hardware to secure the system, it will validate the main user application, if required.

If the SFlash or secure image is found to be invalid or corrupted, the device will enter a DEAD protection state and stays in the DEAD protection state until the device is reset.

***Note**: If the device enters the DEAD protection state, it cannot transition to the RMA lifecycle stage; failure analysis cannot be performed in such cases.*

5.2 Chain of Trust (CoT)

The basis of the chain of trust relies on the memory that cannot be changed, such as the ROM. The rest of the chain is dependent on this fact. The ROM code cannot be changed and is used to validate the next block of execution; in this case it is the flash boot.

Flash boot code, trim constants, and TOC1 are in the SFlash that cannot be reprogrammed. Most of it is preprogrammed at the factory and the calculated hash value for this section is stored in the eFuse and referred to as the “FACTORY_HASH” value. This ensures that the flash boot code, trim values, and TOC1 have not been tampered with after the MCU was provided.

After the lifecycle stage transition from Normal to Secure, the SFlash blocks are validated with another hash value referred to as the “SECURE_HASH”. This value is also stored in the eFuse and cannot be changed after it is programmed. These items in the SFlash include TOC2 and the public key.

The FACTORY_HASH value that is used to validate the SFlash in the Normal lifecycle stage is stored in the eFuse and cannot be changed. A different location is used to validate the SFlash with extra items that were programmed, and SECURE_HASH. This location is written into a separate section of the eFuse.

The entire SFlash block is validated with the SECURE_HASH each time the device wakes from reset in the Secure lifecycle stage. If an error is found while validating the SFlash, the device will no longer complete the boot sequence, and will enter the Dead state.

When verification is successful, the entire SFlash is now trusted because its validation is based on memory (eFuse) that cannot be modified without detection during the SFlash validation in the ROM.

The public key, which is locked into the SFlash, is secure and cannot be changed without being detected as well. It is used by flash boot to validate the next step in the boot process. The flash boot validates the code in the secure image block, which includes a digital signature at the end of the code block. The flash boot uses the SHA-256 hash function to calculate the digital signature of the secure image block. The digital signature attached to the secure image block is encrypted using a private key that is associated with the public key in the SFlash, using up to RSA 4096-bit encryption. The calculated and the stored encrypted digital signatures are then checked to see whether they match. If they match, the secure image block has been validated. The same process can be used by the secure image to validate the user application block. See Figure 4 and Figure 5 for the signing and validation flow of the secure image code.

Figure 3 shows the CoT from the perspective of data and code validation.

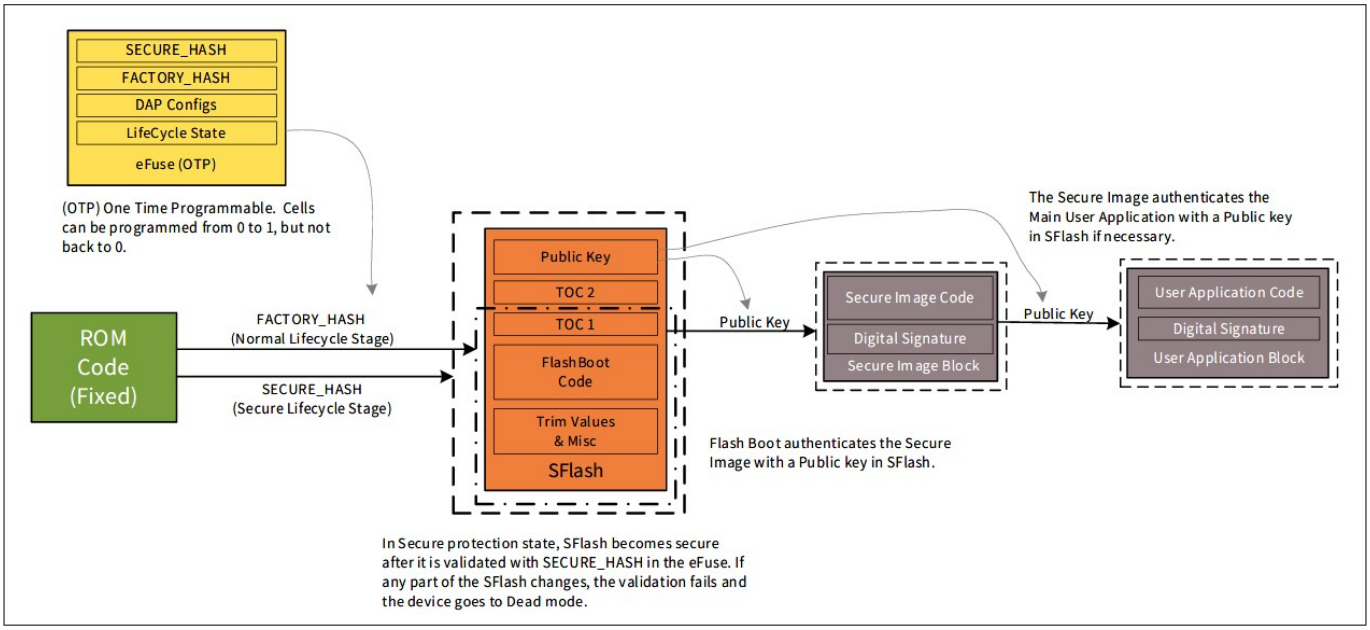


Figure 3 Basic CoT

6 Code signing and validation

This section describes the process of signing a block of code so that it can be validated during boot time.

The encryption method used is public key cryptography (PKC) that uses a private and a public key. Care must be taken to keep the private key at a secure location, so that it never gets into the public domain. If the private key is exposed, it will endanger your system's security. Companies must create a method in which very limited access to the private key is allowed.

The public key, on the other hand, can be viewed by anyone. The only requirement is that the public key must be validated or locked in a way that it can't be changed, so that any modification to the public key can be detected.

In XMC7000/XMC5000 MCU, the public key is stored in the SFlash and validated with the SECURE_HASH as defined in the CoT section. For details of generating and using the private and public keys, see Appendix A - Example of creating public and private keys.

6.1 Code signing

To validate the code such as the user applications during boot time, a digital signature must be created and bundled with the code during build time. The code itself is not stored in the flash in an encrypted format but the digital signature is encrypted. The digital signature is generated with the SHA-256 hash function (a), then encrypted using a private key with up to RSA-4096-bit encryption (b). The reason the digital signature is encrypted is to ensure that a third party, without access to the private key, cannot create a valid code/signature bundle. See Figure 4.

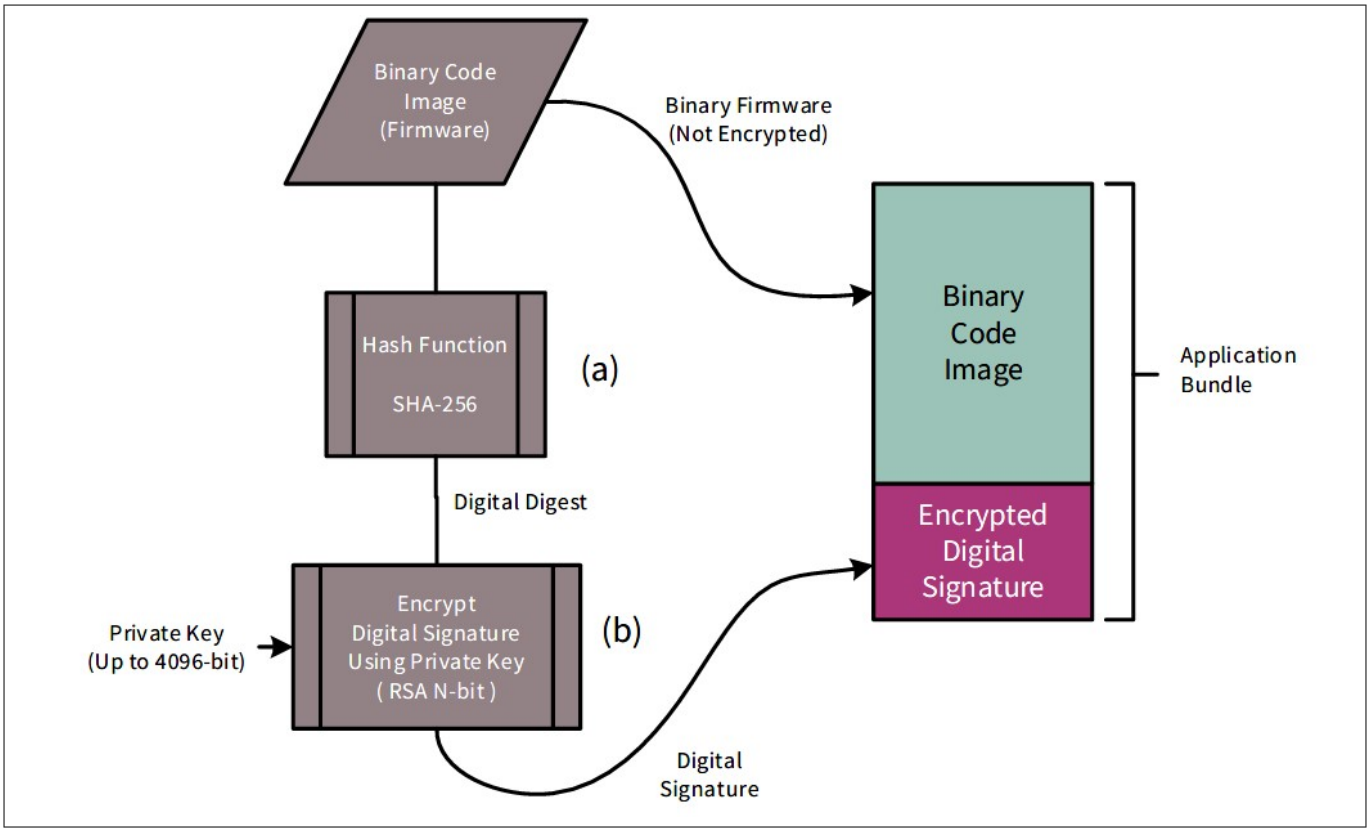


Figure 4 Generation of encrypted digital signature

***Note**: The Flash boot of XMC7000/XMC5000 MCU only supports RSA-2048.*

6.2 Code validation

A secure system must be able to detect code that was not created by the original manufacturer or a trusted source. If non-trusted code is detected, execution must take a known path to a safe state. This also validates that the firmware was not corrupted intentionally or accidentally.

Validation requires three pieces: data, signature, and crypto key. The data and the signature come as a pair; the key is stored in a location that cannot be changed.

- **Data:** This includes both executable code and constants that make up the firmware in an embedded system. This data usually resides in flash memory that usually can be modified at one time or another. Depending on the system, you may modify or update it. Therefore, you must be able to determine whether this data is from a known source and has not been corrupted either by accident or by a malicious event
- **Digital signature:** A digital signature is the hash of a block of data. The hash algorithm used in this case is SHA-256. The digital signature alone can be used to validate that the data is intact. By encrypting the digital signature with a crypto key, you can determine whether the data is from a trusted source, as well as intact
- **Crypto key:** This can be either a public or private key. In the system described in this application note, the public key is stored on the device and the private key is secured by the developer. The public key must be secured in one of two ways:
  1. *A method to validate the source of the key.* This can be accomplished with some type of communication with a known source or server. This is not practical for devices that cannot easily communicate with a known server when required
  2. *Have the key data itself validated by using a key that is stored in memory that cannot be modified.* This is the most likely option for an embedded system. In XMC7000/XMC5000 MCU, a hash is calculated from the areas containing the public key, flash boot code, and trim values. This hash is then stored in one-time programmable eFuse and referred to as “SECURE_HASH”

A user application binary code block (application bundle) includes an encrypted digital signature that was created during build time. The secure image application also uses this format. To validate the block of code, a hash function (SHA-256) is applied to the binary code image, which creates a calculated digital signature, or digital digest. Next, the encrypted digital signature is decrypted using the stored public key, to reveal the decrypted digital signature. The calculated digital digest and the decrypted digital digest (signature) are then checked for an exact match. If they are an exact match, the code is validated. See Figure 5.

6 Code signing and validation

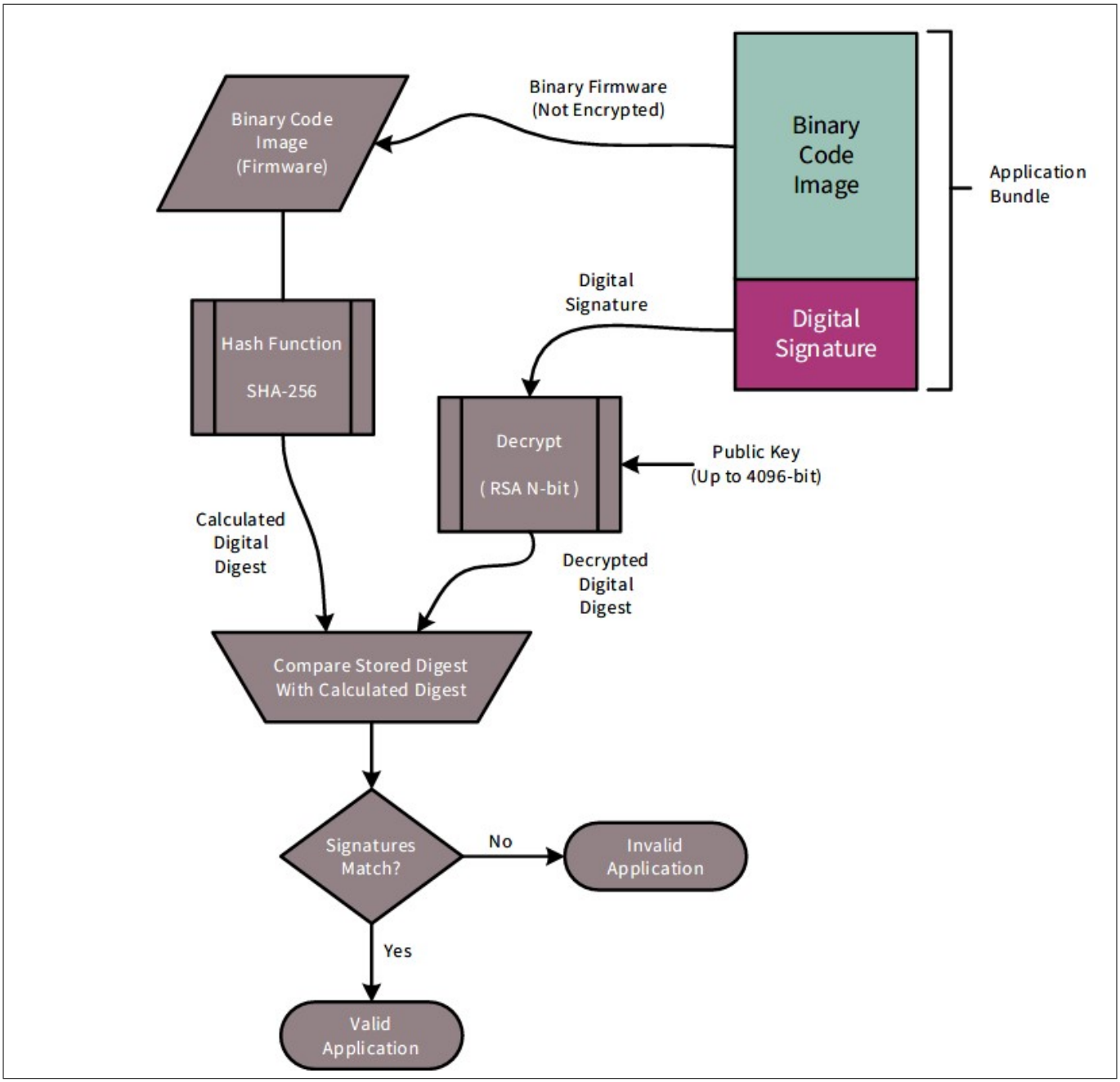


Figure 5 Code validation

***Note**: Due to the improper initialization of the Crypto memory buffer, Crypto ECC errors may be set after boot with application authentication. Therefore, user software should clear or ignore Crypto ECC errors, which are generated during boot with application authentication.*

7 Resource protection

Resource protection means that during runtime, only the bus master or task that should have access to a memory space or register space can access it. This can be a combination of read, write, or execute.

The XMC7000/XMC5000 MCU has blocks called “protection units” to add this functionality. These protection units can be configured to create multiple protection zones that include the flash, SRAM, peripherals, and I/Os. These zones can then be restricted by CPU, tasks, or both. There are four main types of protection units: MPU, SMPU, PPU, and SWPU.

- Each CPU has its own MPU. The MPU is different from the protection structure of the other protection units. MPU protection structures cater only to protection attributes pertaining to a single master. MPU protection structures do not have a protection context parameter associated with them. Protection attributes for an MPU are user/privilege, read/write/execute. MPUs are specific for each bus master and provide resource protection from its various threads or tasks
- SMPUs protect the memory regions that are used by multiple masters. These SMPUs have all the attributes of the MPU, the protection context, and non-secure attribute. The secure image uses SMPUs to restrict access to secure sections of the memory from the non-secure application. Registers used for flash write operations are restricted so that only the SROM code may access those operations. This eliminates any accidental writing or erasure of the flash memory. Also, the SRAM and registers used for Crypto operations are protected to keep operations secure
- PPUs are designed specifically to protect peripheral registers. A PPU is similar to the SMPU. PPUs are fixed-function because they are hardwired to protect a specific peripheral region. Therefore, this type of PPU cannot configure address and size parameters in the protection structure, but you can set protection attributes
- SWPUs are used to implement access restrictions to flash (program/erase) and eFuse (Read/Write). The SWPU is broken into two parts and stored in the SFlash. The first part is configured by boot process, and cannot change. The second part can be used by the application for additional access restrictions specific to the application. It can be updated in the NORMAL_PROVISIONED lifecycle stage by writing to specific row in the SFlash. Also, it can also be updated using the SROM API. The ROM/flash boot reads and configures the two parts of the SWPU from the SFlash. The SWPU consists of the FWPU, ERPU, and EWPU

7.1 Boot protection

Some protection units are configured during the boot process and cannot be reconfigured. These protection units are vital to providing a secure system and providing a reliable access to system call functions. See the “BootROM” chapter in the architecture TRM [2] for details of boot protection.

7.2 Application protection

You can configure access protection to the flash and eFuse during the boot process using the SWPU. This is called application protection units. Application protection units can be configured in the SFlash with the device in the NORMAL_PROVISIONED lifecycle stage. The application protection SWPU has up to 16 entries of the FWPU and up to 4 entries of the ERPU and EWPU. See the “Protection unit” chapter in the architecture TRM [2] for details of the SWPU.

Table 1 shows the default value of application protection.

Table 1 SWPU default value of application protection

SWPU Layout	Description	Default Value
PU_OBJECT_SIZE (4 bytes)	SWPU object size	0x00000030
N_FWPU (4 bytes) Max 16 entries	Number of FWPUs	0x00000000

(table continues...)

Table 1 (continued) SWPU default value of application protection

SWPU Layout	Description	Default Value
N_ERPU	Number of ERPUs	0x00000001
ERPU0_SL_OFFSET (4 bytes)	Protection offset address setting	0x00000068
ERPU0_SL_SIZE (4 bytes)	Region size and ERPU0 enable	0x80000018
ERPU0_SL_ATT (4 bytes)	Slave attribute	0x00FF0007
ERPU0_SL_ATT (4 bytes)	Master attribute	0x00FF0007
N_EWPU	Number of EWPUs	0x00000001
EWPU0_SL_OFFSET (4 bytes)	Protection offset address setting	0x00000068
EWPU0_SL_SIZE (4 bytes)	Region size and EWPU0 enable	0x80000018
EWPU0_SL_ATT (4 bytes)	Slave attribute	0x00FF0007
EWPU0_SL_ATT (4 bytes)	Master attribute	0x00FF0007

See Appendix G - Configure application protection to learn how to configure application protection.

7.3 Security-enhanced PPU configuration

When the magic number is set to the security marker (TOC2_SECURITY_UPDATES_MARKER), the boot process configures the following PPUs to enhance the security and safety. Table 2 shows the PPUs configured by the security marker.

Table 2 PPUs configured by the security marker

Name of PPU	Start address	Size (bytes)	Access for PC > 0 (slave attribute)	Access for PC > 0 (master attribute)
Programmable PPU 11	0x40201000	32	PC1: Full access PC1: Full access	PC1: Full access PC1: Full access
Programmable PPU 12	0x402013c8	4	PC1: Full access PC1: Full access	PC1: Full access PC1: Full access
Programmable PPU 13	0x40201300	256	PC1: Full access PC1: Full access	PC1: Full access PC1: Full access
PERI_MS_PPU_FX_PERI_GR2_GROUP (PPU index=4)	0x40004050	4	PC1: Read-only PC1: Read-only	PC1: Read-only PC1: Read-only

Programmable PPUs 11 and 13 help separate the HSM software and application software by combination with PERI_MS_PPU_FX_CPUSS_CM0. For example, Programmable PPU 11 and 13 are allowed access for application software, and PERI_MS_PPU_FX_CPUSS_CM0 is allowed access for the HSM software. As a result, the CPUSS_AP_CTL register is exclusively controlled by the HSM software while CPUSS_CM0_CLOCK_CTL and RAM0_PWR_CTL, RAM1_PWR_CTL can be controlled by the application software.

Programmable PPU 12 is used to protect the CPUSS_ECC_CTL register. This register provides the ECC error insertion functionality. ECC error injection is a valuable tool to test the ECC logic of memories. In XMC7000/ XMC5000 MCU, like in any other microcontroller where memories are shared by several CPUs, the ECC error injection capability can be misused by one CPU or the debugger to manipulate data in memory regions used by

other CPUs and potentially compromise its security. To mitigate such risks, it is recommended to disable the ECC error injection capability or to limit its usage, at least for the memories which are used by the HSM.

It is assumed that the ECC logic will be tested only during the startup of the device and the ECC error injection functionality is not required during the regular device operation once the startup is completed. To reduce the security risk, it is recommended to disable the ECC error injection logic after the ECC test completion by blocking access to the ECC error injection control registers using an appropriate PPU configuration. To be more precise, ensure that the ECC error injection control registers cannot be written. If write access to the ECC error injection control registers are required after startup, you must ensure that only the trusted software can write to these registers.

Figure 6 shows an example of use for programmable PPUs 11, 12, 13, PERI_MS_PPU_FX_CPUSS_CM0. Programmable PPUs have a higher priority than fixed PPUs. For the same PPU group, the lower number has a higher priority.

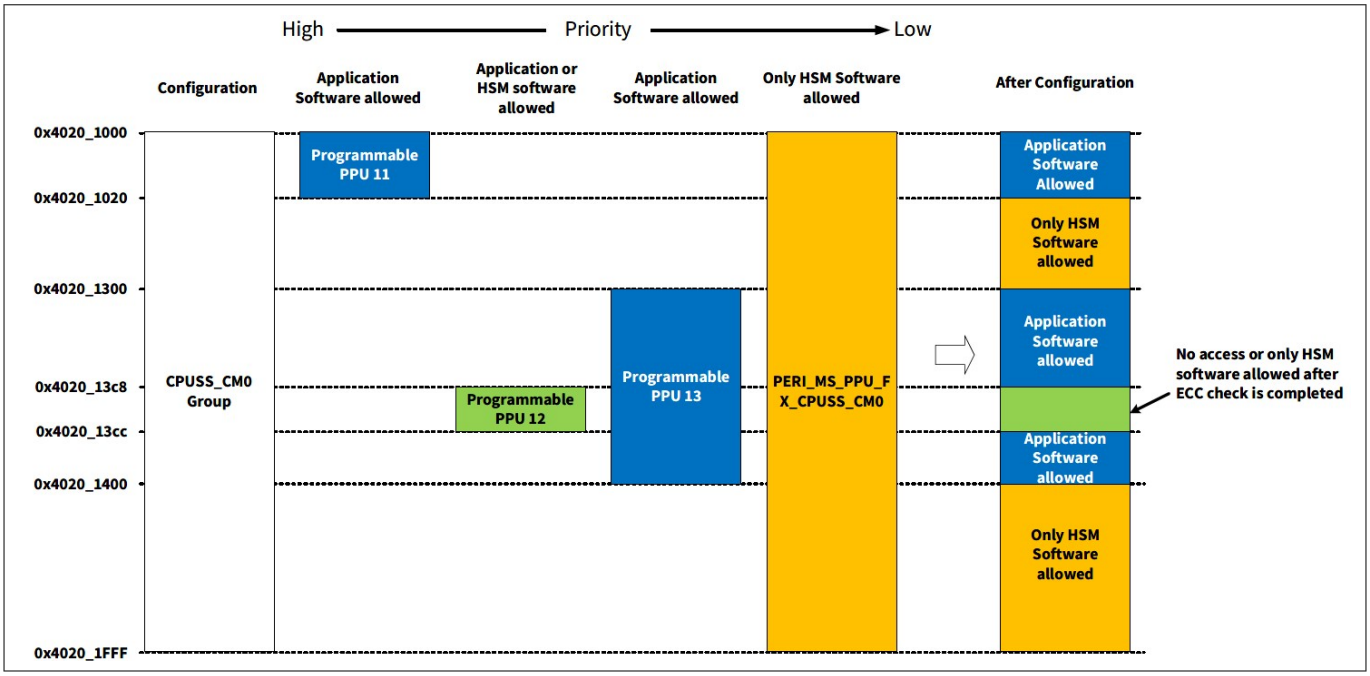


Figure 6 Example of use for PPUs

***Note**: For secure systems, it is recommended that these PPUs are configured by the HSM software.*

Accidental writing to the PERI_GR2_SL_CTL register can stop clock signals to the core MCU function blocks. PERI_MS_PPU_FX_PERI_GR2_GROUP protects PERI_GR2_SL_CTL from accidental write access.

When the security marker is not set in XMC7000/XMC5000 devices, Programmable PPU 11, 12, and 13 are not configured and PERI_MS_PPU_FX_PERI_GR2_GROUP is default value.

See TOC2 for the security marker location.

7.4 Debug and test access port restriction

XMC7000/XMC5000 MCU can configure debug access ports and MPU for test access port during the boot process, depending on the lifecycle stage and the protection state. Table 3 shows the storage location of each access restriction and the conditions to deploy it.

7 Resource protection

Table 3 Debug and test access port restriction

Access restriction	Configuration	Default Value	Location	Lifecycle stage to deploy	Protection state
Normal access restriction	Configured by programming to the SFlash	0x00000080 See Table 4.	SFlash	NORMAL_PROVISIONED or SECURE_W_DEBUG	Normal protection state in NORMAL_PROVISIONED, or Secure protection state in SECURE_W_DEBUG
Normal dead access restriction		0x00000000 See Table 4.			Normal, Secure, or Dead protection state. See the following Note
Secure access restriction	Configure by running `Transitionto Secure` API with SECURE	0x00000000 See Table 4.	eFuse	SECURE	Secure protection state
Secure dead access restriction		0x00000000 See Table 4.			Dead protection state

***Note**: Normal and Normal Dead access restrictions can be updated, but Secure and Secure Dead access restriction cannot be updated.*

***Note**: In SECURE_WITH_DEBUG, NORMAL_PROVISIONED, flash boot keeps the existing protection mode in the DEAD branch during flash boot. See the "Flash Boot Flow" in the architecture reference manual [2] for details.*

Table 4 lists the access restriction configuration. The configuration parameters are common to all access restrictions.

Table 4 Access restriction parameters

Field Name	Bit	Description
AP_CTL_M0_DISABLE	[1:0]	00 – Enable M0-DAP 01 – Disable M0-DAP 1x – Permanently disable M0-DAP See Table 5.
AP_CTL_M4_DISABLE	[3:2]	00 – Enable M4-DAP 01 – Disable M4-DAP 1x – Permanently disable M4-DAP See Table 5.
AP_CTL_SYS_DISABLE	[5:4]	00 – Enable SYS-DAP 01 – Disable SYS-DAP 1x – Permanently disable SYS-DAP See Table 5.

(table continues...)

Table 4 (continued) Access restriction parameters

Field Name	Bit	Description
SYS_AP_MPU_ENABLE	[6]	Indicates that the boot process programs and locks the MPU on the system access port according to the settings in the main/work flash, RAM0, SFlash, and MMIO fields. Access must be disabled to memory regions that are not covered by these six fields (for example, TCM, ROM). 0 – Disable: Does not configure the MPU by the boot process. The application software can configure the MPU. 1 – Enable: Configures the MPU by the boot process according to the settings. The MPU is protected by the PPU; the application software cannot reconfigure the MPU. ***Note**: When this bit is set to "Enable", the SRAM except SRAM0 cannot be accessed via Sys_DAP.*
DIRECT_EXECUTE_DISABLE	[7]	Disables the DirectExecute system call functionality (implemented in software). This filed is fixed "1" in NAR.
FLASH_ENABLE	[10:8]	Indicates the portion of the main flash that is accessible through the system access port. Only a portion of the flash starting at the bottom of the area is exposed. Encoding is as follows: "0": Entire region "1": 7/8th "2": 3/4th "3": 1/2 "4": 1/4th "5": 1/8th "6": 1/16th "7": Nothing
RAM0_ENABLE	[13:11]	Indicates the portion of SRAM 0 that is accessible through the system access port. Only a portion of the SRAM starting at the bottom of the area is exposed. Encoding is the same as FLASH_ENABLE.

(table continues...)

Table 4 (continued) Access restriction parameters

Field Name	Bit	Description
WORK_FLASH_ENABLE	[15:14]	Indicates the portion of the work flash that is accessible through the system access port. Only a portion of the work flash starting at the bottom of the area is exposed. Encoding is as follows: “0”: Entire region “1”: 1/2 “2”: 1/4th “3”: Nothing
SFLASH_ENABLE	[17:16]	Indicates the portion of the supervisory flash that is accessible through the system access port. Only a portion of the supervisory flash starting at the bottom of the area is exposed. Encoding is as follows: “0”: entire region “1”: 1/2 “2”: 1/4th “3”: Nothing
MMIO_ENABLE	[19:18]	Indicates the portion of the MMIO region that is accessible through the system access port. Encoding is as follows: “0”: All MMIO registers “1”: Only IPC MMIO registers accessible (for system calls) “2”, “3”: No MMIO access

Table 5 shows Debug port access restrictions setting.

Table 5 Debug port access restriction setting

Element	Description
Enable M0/M7/M4/SYS-DAP	Corresponding DAP is enabled.
Disable M0/M7/M4/SYS-DAP	Corresponding DAP is temporarily disabled. DAP can be re-enabled by application software.
Permanently Disable M0/M7/M4/SYS-DAP	Corresponding DAP is permanently disabled.

***Note**: Normal and Normal Dead access restriction can be updated, but they cannot be set less restrictive. For example, Disable DAP setting cannot be changed to Enable DAP setting.*

7.4.1 Minimum requirements for a system call initiated by a debugger

You should consider the following before enabling accesses that are required to perform the system call for the transition to the RMA API.

Considering a secured system, it is recommended to temporarily disable all DAPs as part of the initial access restriction configurations. It is also important to keep in mind that an enabled CM7/CM4_DAP allows the debugger to power up the CM7/CM4 CPU and start code execution from an arbitrary memory address. This is possible when ROM/Flash boot configures SWD/JTAG pins. In the worst-case scenario, this can happen even before the CM0+ application starts.

If the CM0+ application (for example, the HSM software) has specific assets that must be protected from the main application (for example, keys stored in the HSM portion of the flash), it is recommended not to enable CPU access ports through the access restrictions. Otherwise, CM7/CM4 CPU could be used to access the HSM memory before CM0+ could enable protection.

To enable the Sys_DAP, perform the following steps in the user application. To perform these steps, all DAPs should be set to "disabled" (temporarily disabled) and SYS_AP_MPU_ENABLE should be set to "0" (disabled).

1. Configure all necessary DAP pins
2. Configure the Sys_DAP MPU structures to give access to the required resources (for example, IPC MMIOs, SRAM, and so on)
3. Configure the Sys_DAP MPU structure so that it does not give any access to the SRAM area, which is additionally used by the TransitiontoRMA API. This is the 2 KB of SRAM starting from (SRAM0 + 2 KB)
4. Configure PPU protection for the Sys_DAP MPU
5. Make sure that the PPU/SMPU settings allow the required IPC MMIO and SRAM accesses
6. Interrupt initialization in user software: Enable IRQ0 and IRQ1 to allow the handling of system call interrupts. Define interrupt vectors for both interrupts
7. Enable all necessary DAPs

***Note**: If a configuration item fails, the system call cannot be applied.*

***Note**: Perform steps 1 to 5 before enabling the DAP.*

***Note**: You must decide on the allocation of these steps to individual CPUs based on your security requirements.*

8 Configure a secure system

This section will take you through each pieces of a secure system and describe how to generate them.

Configuring a fully secure system with a CoT is more complicated than generating a simple application. In addition to the user code, a secure system must contain several other pieces normally not required in a simple (non-secure) system. The following are the memory sections that must be programmed when creating a secure system:

- SECURE_HASH (eFuse)
- DAP configuration (eFuse)
- Lifecycle stage (eFuse)
- Public key (SFlash)
- TOC2 (SFlash)
- Secure image block (user flash)
- User application block (user flash)

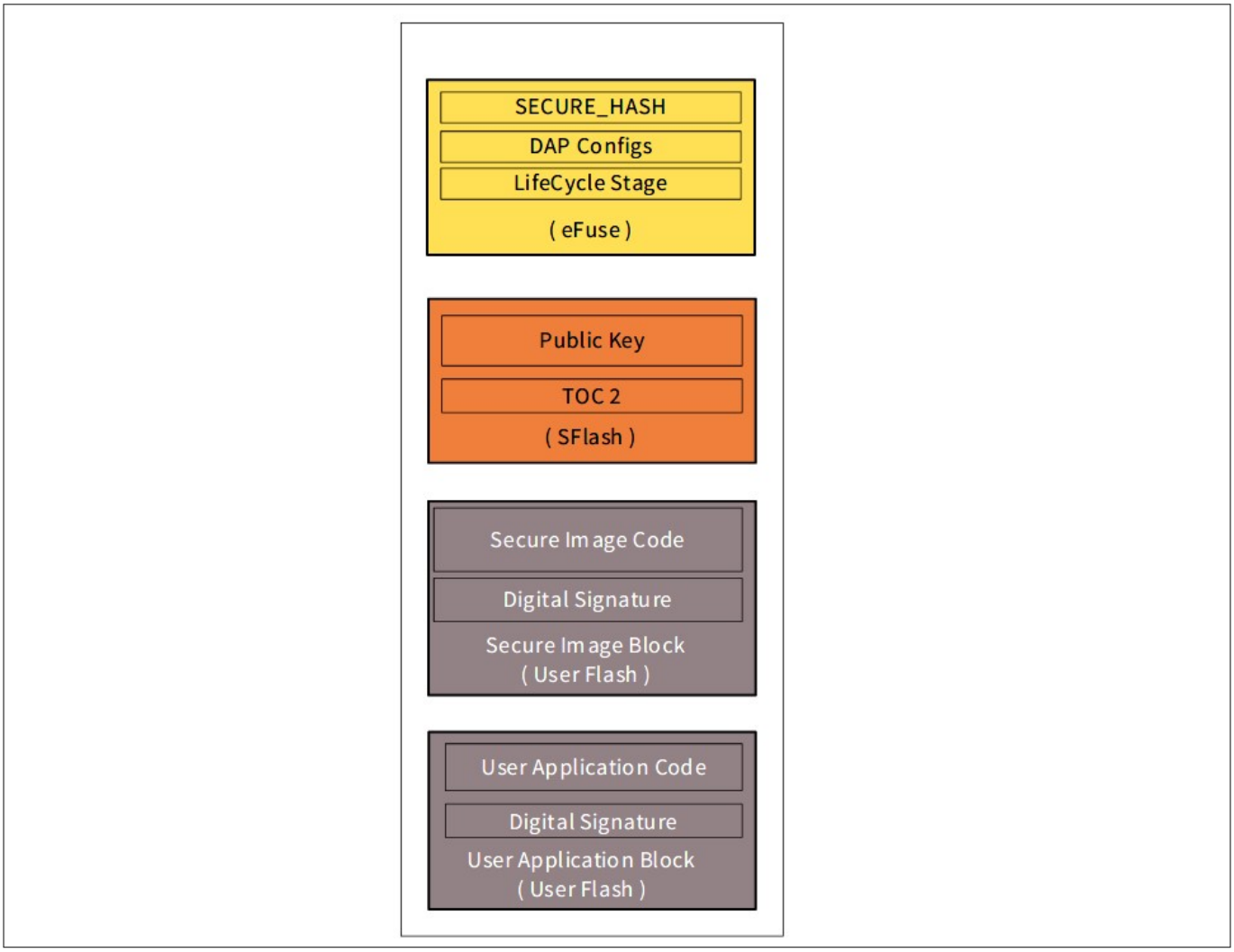


Figure 7 Secure system configuration

Building one block may be dependent on building another one. For example, TOC2 contains the start address of both the secure image block and the user application block. The SECURE_HASH stored in the eFuse depends on everything in the SFlash.

Note that some of these items are programmed implicitly through the appropriate SROM firmware.

8.1 TOC2

There are two table sections in the SFlash: TOC1 and TOC2.

TOC1 is used for internal purposes and is not user-editable. TOC2 is used to point to the location of each application. In TOC2, the secure image is indicated as the "user application" and the main user application is indicated as "CM7 or CM4 core x user application" (x shows the CPU number.)

There are two parameters that you may want to change for a faster boot sequence, at offset 0x1F8 (See Table 6):

- Boot clock frequency parameter, “CLOCK_CONFIG”
- Debug parameter, “LISTEN_WINDOW ”

You may change these in TOC2 to the values defined in the table.

Table 6 Elements of TOC2

Offset	Purpose	Default Value
0x00	Object size in bytes for CRC calculation starting from offset 0x00	0x00001FC
0x04	Magic number (Fixed: 0x01211220)	0x01211220
0x08	Null-terminated table of pointers representing the SMIF configuration structure	0x00000000
0x0C	Address of the first user application object	0x10000000
0x10	Format of the first user application object. 0: Basic, 1: CySAF, 2: Simplified	0x00000000 Set to 1 when Secured Boot
0x14	Address of the second user application object The second user application is validated if the first application validation failed.	0x00000000
0x18	Format of the second user application object. 0: Basic, 1: CySAF, 2: Simplified	0x00000000 Set to 1 when Secured Boot
0x1C	Address of the first CM7 or CM4 core1 user application object	0x00000000
0x20	Address of the second CM7or CM4 core1 user application object	0x00000000
0x24	Address of the First CM7 or CM4 core2 user application object	0x00000000
0x28	Address of the second CM7 or CM4 core2 user application object	0x00000000
0xFC	Enables the protection configuration for security enhancement when the magic number is valid. See Security-enhanced PPU configuration for details.	0x00000000 Magic number = 0xFEDEEDDF
0x100	Number of additional objects to be verified for SECURE_HASH	0x00000003

(table continues...)

Table 6 (continued) Elements of TOC2

Offset	Purpose		Default Value
0x104	Address of signature verification key (0 if none). The object is signature specific key. It is the public key in case of RSA.		0x00000000 Case of location in SFlash: 0x17006400
0x108	Address of application protection (This area must not be modified.)		0x17007600
0x10C	Reserved (This area must not be modified.)		0x00000000
0x110-0x1F0	Additional objects if required; 0s if none		0x00000000
0x1F8	TOC2_FLAGS: Controls the default configuration Bits [1:0]: CLOCK_CONFIG Flag to indicate the clock frequency configuration. The clock should stay the same after flash boot execution.		See the registers TRM [2] for default value.
	Value [1:0]	Description	
	0x0	8 MHz, IMO, no FLL	
	0x1	25 MHz, IMO + FLL	
	0x2	50 MHz, IMO + FLL	
	0x3	Use ROM boot clocks configuration (100 MHz)	
	Bits [4:2]: LISTEN_WINDOW Flag to determine the listen window to allow sufficient time to acquire the debug port		
	Value [4:2]	Description	
	0x0	20 ms	
	0x1	10 ms	
	0x2	1 ms	
	0x3	0 ms (No Listen window)	
	0x4	100 ms	
	Others	Reserved	
	Bits [6:5]: SWJ_PINS_CTL Flag to determine if SWJ pins are configured in SWJ mode by flash boot SWJ pins may be enabled later in the user code.		
	Value [6:5]	Description	
	0x0	Do not enable SWJ pins in flash boot. Listen window is skipped.	

(table continues...)

Table 6 (continued) Elements of TOC2

Offset	Purpose		Default Value
	0x1	Do not enable SWJ pins in flash boot. Listen window is skipped.	
	0x2	Enable SWJ pins in flash boot	
	0x3	Do not enable SWJ pins in flash boot. Listen window is skipped.	
	Bits [8:7]: APP_AUTH_CTL Flag to determine if the application image digital signature verification (authentication) is performed:		
	Value [8:7]	Description	
	0x0	Authentication is enabled.	
	0x1	Authentication is disabled.	
	0x2	Authentication is enabled (recommended).	
	0x3	Authentication is enabled.	
	Bits [10:9]: FB_BOOTLOADER_CTL Flag to determine if the internal bootloader in flash boot is disabled:		
	Value [10:9]	Description	
	0x0	Internal bootloader is disabled.	
	0x1	Internal bootloader is launched if other bootloader conditions are met. See the following for conditions.	
	0x2	Internal bootloader is disabled.	
	0x3	Internal bootloader is disabled.	

The bootloader is enabled when the following conditions are met:

- Two words at the start of the flash must be ‘0xFFFFFFFF’
- TOC2 is valid and the internal bootloader is enabled (default) by TOC2_FLAGS.FB_BOOTLOADER_CTL bits, or TOC2 is empty
- Protection state is not SECURE and not SECURE_DEAD
- No debugger connection happened during the one-second wait window

8 Configure a secure system

If enabling conditions of the bootloader are met, when erasing the first application, the bootloader will be launched and the second application will not be activated. Therefore, it is recommended to disable the bootloader (TOC2_FLAGS.FB_BOOTLOADER_CTL=0x2) if not required.

8.1.1 Configuration

To generate proper values for TOC2, include an instance of the following code in the project:

```
/** Flashboot parameters */
#define CY_SI_FLASHBOOT_FLAGS ((CY_SI_FLASHBOOT_CLK_100MHZ << CY_SI_TOC_FLAGS_CLOCKS_POS) \
                               | (CY_SI_FLASHBOOT_WAIT_20MS << CY_SI_TOC_FLAGS_DELAY_POS) \
                               | (CY_SI_FLASHBOOT_SWJ_ENABLE << CY_SI_TOC_FLAGS_SWJEN_POS) \
                               | (CY_SI_FLASHBOOT_VALIDATE_ENABLE <<
CY_SI_TOC_FLAGS_APP_VERIFY_POS) \
                               | (CY_SI_FLASHBOOT_FBLOADER_DISABLE <<
CY_SI_TOC_FLAGS_FBLOADER_ENABLE_POS))
/** TOC2 in SFlash */
CY_SECTION(".cy_toc_part2") __USED static const cy_stc_si_toc_t cy_toc2 =
{
    .objSize         = CY_SI_TOC2_OBJECTSIZE,           /* Offset+0x00: Object Size (Bytes)
excluding CRC */
    .magicNum        = CY_SI_TOC2_MAGICNUMBER,          /* Offset+0x04: TOC2 ID (magic number)
*/
    .smifCfgAddr     = 0UL,                             /* Offset+0x08: SMIF config list
pointer */
    .cm0pappAddr1    = CY_SI_SECURE_FLASH_BEGIN,        /* Offset+0x0C: App1 (CM0+ First User
App Object) addr */
    .cm0pappFormat1  = CY_SI_APP_FORMAT_CYPRESS,        /* Offset+0x10: App1 Format */
    .cm0pappAddr2    = CY_SI_USERAPP_FLASH_BEGIN,       /* Offset+0x14: App2 (CM0+ Second User
App Object) addr */
    .cm0pappFormat2  = CY_SI_APP_FORMAT_BASIC,          /* Offset+0x18: App2 Format */
    .cmx1appAddr1    = CY_SI_CM471_1stAPP_FLASH_BEGIN,  /* Offset+0x1C: App3 (CM7/CM4_1 1st
User App Object) addr */
    .cmx1appAddr2    = CY_SI_CM471_2ndAPP_FLASH_BEGIN,  /* Offset+0x20: App4 (CM7/CM4_1 2nd
User App Object) addr */
    .cmx2appAddr1    = CY_SI_CM472_1stAPP_FLASH_BEGIN,  /* Offset+0x24: App5 (CM7/CM4_2 1st
User App Object) addr */
    .cmx2appAddr2    = CY_SI_CM472_2ndAPP_FLASH_BEGIN,  /* Offset+0x28: App6 (CM7/CM4_2 1st
User App Object) addr */
    .reserved1       = {0UL},                           /* Offset+0x2C-0xFB: Reserved area
212Bytes */
    .securityMarker  = CY_SECURITY_NOT_ENHANCED,        /* Offset+0xFC Security Enhance Marker
*/
    .shashObj        = 3UL,                             /* Offset+0x100: Number of verified
additional objects (S-HASH)*/
    .sigKeyAddr      = CY_SI_PUBLIC_KEY,                /* Offset+0x104: Addr of signature
verification key */
    .swpuAddr        = CY_SI_SWPU_BEGIN,                /* Offset+0x108: Addr of SWPU Objects */
    .toc2Addr        = 0UL,                             /* Offset+0x10C:
TOC2_OBJECT_ADDR_UNUSED */
    .reserved2          = {0UL},                        /* Offset+0x110-0x1F4: Reserved area
232Bytes */
    .tocFlags        = CY_SI_FLASHBOOT_FLAGS,           /* Flashboot flags stored in TOC2 */
    .crc             = 0UL,                             /* Offset+0x1FC: Reserved area 1Byte */
};
```

8.2 User application block

Secure applications must be validated with a public Crypto key. To do this, the application must use the Infineon secure application format (CySAF) that includes a digital signature. The user application format is specified as “Format of First/Second User Application Object (offset=0x10/0x18)” in TOC2.

This allows flash boot to perform the validation during the boot process, before the application is executed. The application format encapsulates the application binary, application metadata, and an encrypted digital signature. Although there is a place for both CM0+ and CM7/CM4 images in this format, the secure image requires only the CM0+ image. The user application includes both images; see Figure 8.

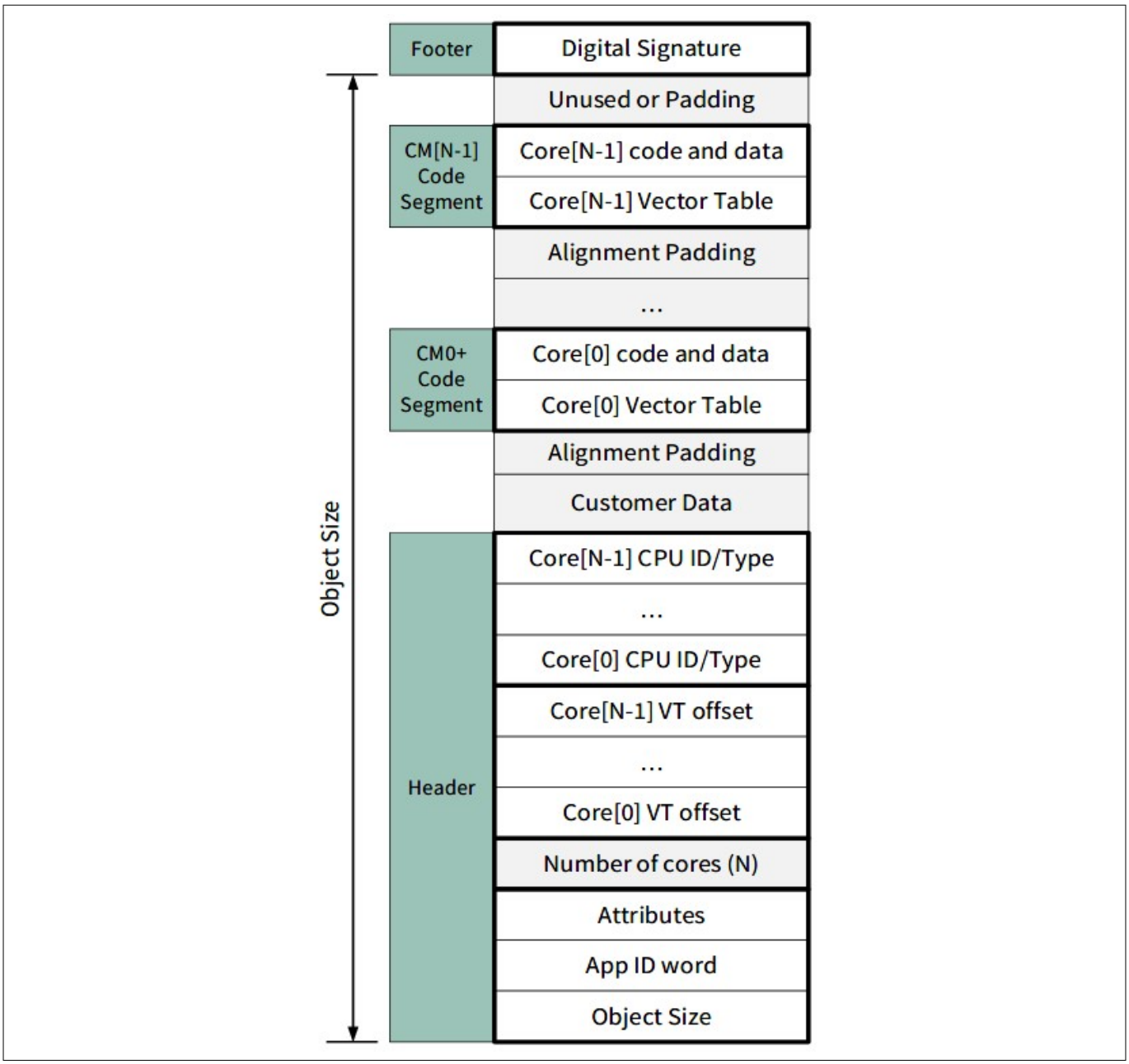


Figure 8 Secure application format

***Note**: CySAF is required to transition the lifecycle stage to SECURE or SECURE_W_DEBUG. Therefore, CySAF is also required when transitioning the lifecycle stage from NORMAL_PROVISIONED to RMA. See Appendix F - Transition to RMA lifecycle stage for transitioning the lifecycle stage from NORMAL_PROVISIONED to RMA.*

Table 7 lists the details of the header section in CySAF format. It defines the total size, the number of cores, the type of application, and the offset to each core application vector table.

Table 7 Header details

Offset	Size	Item	Description
0x00	4 bytes	Object Size	A flash image size in bytes (application size)
0x04	4 bytes	Application ID/Version	This value identifies the type of the flash image: Bit 31 - 28: Always 0 Bit 27 - 24: Major version Bit 23 - 16: Minor version Bit 15 - 0: Application ID. For example, 0x0000 - User application 0x8001 - Flash boot 0x8002 - Security library 0x8003 - Bootloader All other values - Reserved
0x08	4 bytes	Attribute	Reserved for future use
0x0C	4 bytes	Number of cores(N)	Number of cores used by the application
0x10 + (4*i)	4 bytes	Core(i) VT offset	Offset to the vector table from this address in Core(i) code segment
0x10+(4*N)+(4*i)	4 bytes	Core(i) CPU ID and core index	User-assigned CPU ID and core index: Bit 31 - 20: CPU ID. This is the part of value from the CPUID [15:4] register in an Arm® device. Bit 7 - 0: Core index The core index is used to distinguish between multiple cores of the same type. For example, consider a system consisting of CM0+ and two CM7_0/CM7_1s. The CM0+ is identified by CPUID=0xC60 and Core Index=0. The first CM7_0is identified by CPUID=0xC27 and Core Index=0. The second CM7_1 is identified by CPUID=0xC27 and Core Index=1.

To generate proper values for the application header, include an instance of the following code in the project:

```
/** Secure Application header */
CY_SECTION(".cy_app_header") __USED static const cy_stc_si_appheader_t cy_si_appHeader =
{
    .objSize        = CY_M0PLUS_SI_SIZE,
    .appId          = (CY_SI_APP_VERSION | CY_SI_APP_ID_SECUREIMG),
    .appAttributes  = 0UL,                          /* Reserved */
    .numCores       = 1UL,                          /* Only CM0+ */
    .core0Vt        = CY_SI_VT_OFFSET,              /* CM0+ VT offset */
    .core0Id        = CY_SI_CPUID | CY_SI_CORE_IDX,  /* CM0+ core ID */
};

/** Secure Image Digital signature (Populated by cymcuelftool) */
CY_SECTION(".cy_app_signature") __USED CY_ALIGN(4)
static const uint8_t cy_si_appSignature[CY_SI_SECURE_DIGSIG_SIZE] = {0u};
```

8.3 Secure boot RSA public key format

The algorithm for encrypting is SHA-256 + RSASSA PKCS v1.5 algorithms as the scheme for signature verification. You should define the RSA key location, and provide a pointer to the key in TOC2. The public key contents are checked by ROM boot HASH computation against any change, when placed in the SFlash.

The SFlash region stores the public key in a binary format. The modulus, exponent, and three coefficients are pre-calculated to speed up the validation. Figure 9 shows the format.

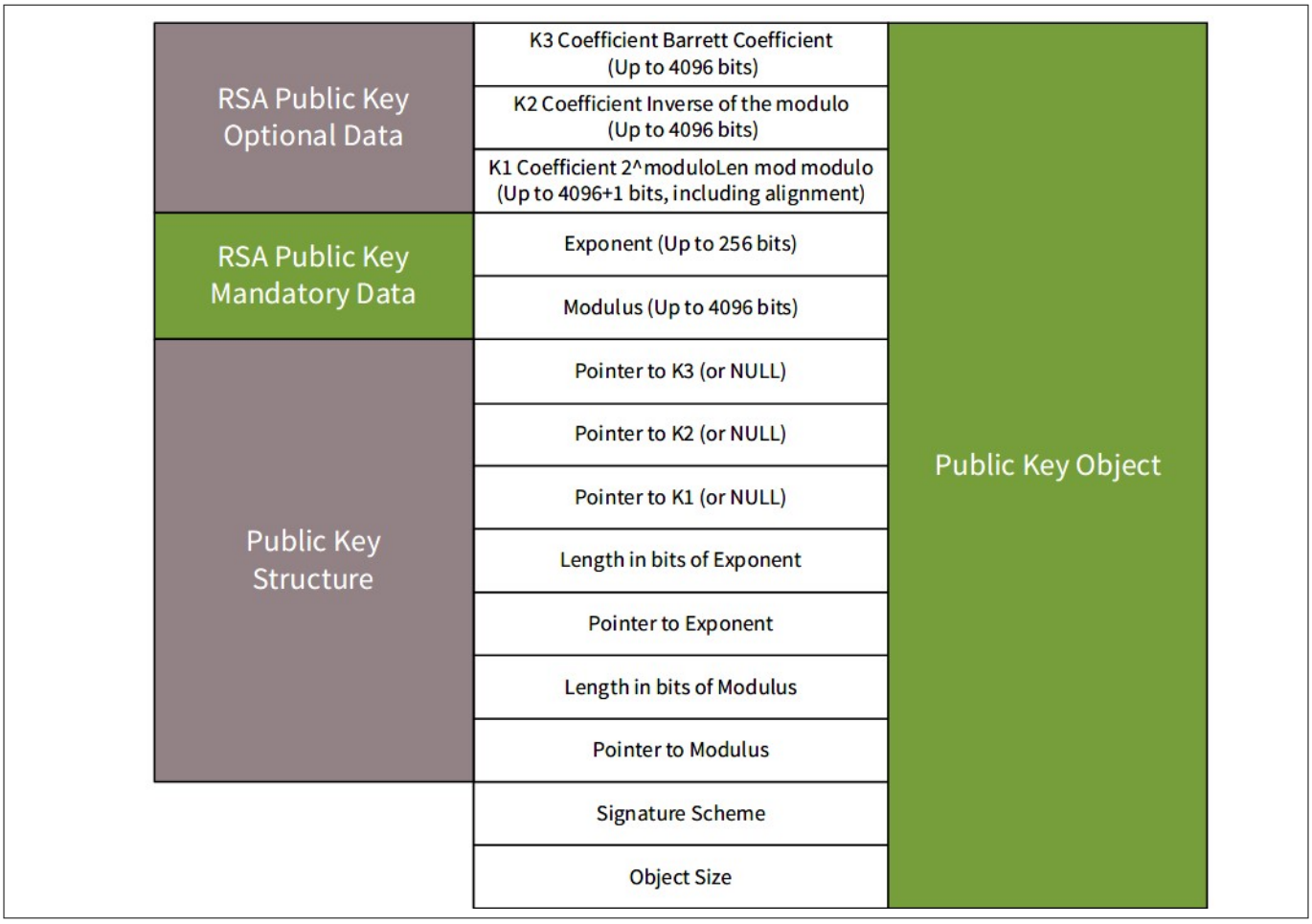


Figure 9 Public key format

The key is stored in three structures.

- The first structure “Key” is stored as an object that can easily be included in the SECURE_HASH calculation. The “Signature Scheme” defines the structure of the key. The “Object Size” contains the full size of the public key object, which contains the entire three structures
- The second structure contains the individual pieces of the public key: coefficients (K1, K2, K3), exponent (E), and modulus (N). These values must be stored in a little-endian list of bytes
- The third structure is a list of pointers to each piece of the public key, which is the format required for a call to the SROM firmware

To generate proper values for the public key format, include an instance of the following code in the project.

8 Configure a secure system

See Appendix A - Example of creating public and private keys for more details of generating and using the private and public keys.

```
/** Public key in SFlash */
CY_SECTION(".cy_sflash_public_key") __USED const cy_si_stc_public_key_t cy_publicKey =
{
    .objSize = sizeof(cy_si_stc_public_key_t),
    .signatureScheme = 0UL,
    .publicKeyStruct =
    {
        .moduloAddr         = CY_SI_PUBLIC_KEY + offsetof(cy_si_stc_public_key_t, moduloData),
        .moduloSize         = CY_SI_PUBLIC_KEY_SIZEOF_BYTE * CY_SI_PUBLIC_KEY_MODULOLENGTH,
        .expAddr            = CY_SI_PUBLIC_KEY + offsetof(cy_si_stc_public_key_t, expData),
        .expSize            = CY_SI_PUBLIC_KEY_SIZEOF_BYTE * CY_SI_PUBLIC_KEY_EXPLENGTH,
        .barrettAddr        = CY_SI_PUBLIC_KEY + offsetof(cy_si_stc_public_key_t, barrettData),
        .inverseModuloAddr  = CY_SI_PUBLIC_KEY + offsetof(cy_si_stc_public_key_t,
inverseModuloData),
        .rBarAddr           = CY_SI_PUBLIC_KEY + offsetof(cy_si_stc_public_key_t, rBarData),
    },
    .moduloData =
    { // N(Modulus)
    },
    .expData =
    { // E(Exponent)
    },
    .barrettData =
    { // K1
    },
    .inverseModuloData =
    { // K2
    },
    .rBarData =
    { // K3
    },
};
```

9 Appendix A - Example of creating public and private keys

This section explains how to generate a set of public and private keys, to format them to a 'C' format, and to update the source file with the new key.

There are two methods to generate key pairs and convert the public key into a C structure that can be embedded in the application. One method uses the **Edge Protect Tools** provided by Infineon, while the other uses **Python scripts** to perform the same process. The following sections introduce both methods.

9.1 Edge protect tools

Edge Protect Tools is a comprehensive security tools package built on Python, offering a command-lineinterface to interact with MCUs for various purposes. These purposes include: keys creations, image signing, and more. The tools are developed using the Python programming language and leverage several third-party Python modules to handle different tasks.

The Edge Protect Tools is packaged as a part of the Edge Protect Security Suite, this tool has two main functions for the XMC7000/XMC5000 device:

- **Create keys:** A key is a file used to authorize access to device data. There must be a common key pair between the secure device and user application. A device must be provisioned with a public key and the user application must be signed with a corresponding private key from the same pair
- **Sign a user application:** To run a user application on a secure device, the application must be signed with the key

Follow these steps to setup Edge Protect Security Suite.

1. Download and install the Infineon Developer Center Launcher
2. Log-in using your Infineon credentials
3. Download and install the "ModusToolbox™ Edge Protect Security Suite" from Developer Center Launcher

***Note**: The default installation directory of the Edge Protect Security Suite in Windows operating system is C:/Users/<USER>/Infineon/Tools. After installing the Edge Protect Security Suite, add the Edge protect tools executable to system PATH variable (/ModusToolbox-Edge-Protect-Security-Suite-<version>/tools/edgeprotecttools/bin).*

Once you have installed the Edge Protect Tools package, run the following command and find the name of your target in the list of supported targets (xmc7100, xmc7200 or xmc5000).

```
$ edgeprotecttools device-list
```

You need to create a public-private key pair. These keys are used during secure boot enablement to sign the images to be validated by the BootROM. The following command generates a public-private key pair.

```
# Generate a new RSA-2048 key pair in the PEM format
$ edgeprotecttools -t xmc7200 create-key --key-type rsa2048 --output private.pem public.pem --
format PEM
```

Next, running the following command to converts RSA public key to file that can be included into application.

```
# Represents a key as a C structure to include it into application
$ edgeprotecttools convert-key --key-path public.pem --output secure_boot_key.c --fmt
secure_boot
```

9.2 Python scripts

9.2.1 Additional tools required

1. OpenSSL
2. Python 3 (required for one of the provided scripts that is used to format the public key)

There are several ways to generate RSA private and public keys. In the following method, you can use the modus-shell of ModusToolbox™ software, which already includes OpenSSL/Python.

9.2.2 Scripts

The following provides two scripts *rsa_keygen.bash* and *rsa_to_c.py* to convert the output from OpenSSL to a format compatible with C and the structures used by the secure image to store the public key.

Copy the two scripts *rsa_keygen.bash* and *rsa_to_c.py* to a work directory where the key pairs and a section of C code will be generated. The path of this work directory must not include spaces.

The batch script *rsa_keygen.bash* calls OpenSSL functions. The bash file creates a directory called "*keys_generated*". This creates two files containing the private and public keys generated with OpenSSL. These files are called "*rsa_private_generated.txt*" for the private key, and "*rsa_public_generated.txt*" for the public key.

Next, the bash file will call the Python script "*rsa_to_c.py*". This script formats the data to be compatible with C and the Infineon public key format from the generated public key file. The output is placed in the file *rsa_to_c_generated.txt* in the *keys_generated* directory along with the public and private key files.

The following commands show a *rsa_keygen.bash* for generating RSA-2048 private key and public key:

```
#!/bin/bash
set -e
OUT_DIR="keys_generated"
PRIV_NAME="rsa_private_generated.txt"
PUB_NAME="rsa_public_generated.txt"
MOD_NAME="rsa_to_c_generated.txt"
mkdir -p "$OUT_DIR"
# Generate the RSA-2048 public and private keys
openssl genrsa -out $OUT_DIR/$PRIV_NAME 2048
openssl rsa -in $OUT_DIR/$PRIV_NAME -outform PEM -pubout -out $OUT_DIR/$PUB_NAME
# Create C-code ready public key
python rsa_to_c.py $OUT_DIR/$PUB_NAME > $OUT_DIR/$MOD_NAME
```

9 Appendix A - Example of creating public and private keys

The following shows the *rsa_to_c.py* Python script for formatting the public key to C format:

```
#!/usr/bin/env python3
""" This script may be used to generate RSA public key modulus, exponent, and
    additional coefficients. Additional coefficients are optional and are used
    only to increase RSA calculation performance up to 4 times.
    The format of output may be defined by command line arguments and is either
    the raw HEX data, or the array.
"""
import sys, subprocess, os

if sys.version_info < (3,):
    integer_types = (int, long,)
    ## Used in convert_hexstr_to_list
else:
    integer_types = (int,)

def main():
    """ Main function: Build the strings to print out the public key modulus and exponent.
    """
    if len(sys.argv) < 2:
        print("Usage: %s <public_key_file_name> [-norev] [-out <file_name>]" % sys.argv[0])
        return 1
    isReverse = True
    out_file_name = ''
    for idx in range(len(sys.argv)):
        if "-norev" == sys.argv[idx]:
            isReverse = False
        if "-out" == sys.argv[idx]:
            out_file_name = sys.argv[idx+1]
    modulus_list = [] # list to collect bytes of modulus
    rsaExp = "" # string that will contain the parsed RSA exponent
    key_len = 0 # containt the length in bits of an RSA modulus
    try:
        # build openssl command line
        cmd_line = ['openssl', 'rsa', '-text', '-pubin', '-in',
                    sys.argv[1],
                    '-noout']
        output, error = subprocess.Popen(
            cmd_line, universal_newlines=True,
            stdout=subprocess.PIPE, stderr=subprocess.PIPE).communicate()
        # check for errors (warnings ignored)
        lines = error.split("\n")
        error_lines = []
        for line in lines:
            if (len(line) != 0) and (("WARNING:" in line) == False):
                error_lines.append(line)
        if len(error_lines) != 0:
            print ("OpenSSL call failed" + "\n" + " ".join(cmd_line) + "\n" + str(error_lines) )
            return 1
        modulus_found = False
        for line in output.split("\n"):
            if "Public-Key" in line:
```

9 Appendix A - Example of creating public and private keys

```
                # get length of RSA modulus
                key_len = int(''.join(filter(str.isdigit, line)))
            if "Modulus" in line:
                # modulus record is found
                modulus_found = True; continue
            if "Exponent" in line:
                modulus_found = False
                # Exponent record is found
                rsaExp = line.split(" ")[2][1:-1]
            if modulus_found:
                # Collect bytes of modulus to list
                modulus_list = modulus_list + line.strip().split(":")
    except subprocess.CalledProcessError as err:
        print ("OpenSSL call failed with errorcode=" + str(err.returncode) \
                + "\n" + str(err.cmd) + "\n" + str(err.output))
        return 1
    #normalize data
    # remove empty strings from modulus_list
    modulus_list = [i for i in modulus_list if i]
    if (len(modulus_list) == key_len // 8 + 1) and (int(modulus_list[0]) == 0):
        # remove first zero byte
        modulus_list.pop(0)
    # Check parsed data
    if not key_len:
        print ("Key length was not gotten by parsing." )
        return 1
    if len(modulus_list) != (key_len // 8):
        print ("Length of parsed Modulus (%s) is not equal to Key length (%s)." % (key_len,
len(modulus_list) * 8))
        return 1
    modulus_hex_str = "".join(modulus_list)
    (barret, inv_modulo, r_bar) = calculate_additional_rsa_key_coefs(modulus_hex_str)
    barret_list = convert_hexstr_to_list(barret, isReverse)
    # add three zero bytes
    barret_list = ([0]*3 + barret_list) if not isReverse else (barret_list + [0]*3)
    barret_str = build_returned_string(barret_list)
    barret_str = ".barrettData =\n{\n%s\n}," % barret_str
    inv_modulo_list = convert_hexstr_to_list(inv_modulo, isReverse)
    inv_modulo_str = build_returned_string(inv_modulo_list)
    inv_modulo_str = ".inverseModuloData =\n{\n%s\n}," % inv_modulo_str
    r_bar_list = convert_hexstr_to_list(r_bar, isReverse)
    r_bar_str = build_returned_string(r_bar_list)
    r_bar_str = ".rBarData =\n{\n%s\n}," % r_bar_str
    rsaExp_list = convert_hexstr_to_list(rsaExp, isReverse)
    rsaExp_list_len = len(rsaExp_list)
    if rsaExp_list_len % 4 != 0:
        rsaExp_list = ([0]*(4-(rsaExp_list_len % 4)) + rsaExp_list) if not isReverse \
                      else  (rsaExp_list + [0]*(4-(rsaExp_list_len % 4)))

    rsaExp_str = build_returned_string(rsaExp_list)
    rsaExp_str = ".expData =\n{\n%s\n}," % rsaExp_str
    # Check and apply isReverse flag
    if isReverse:
```

9 Appendix A - Example of creating public and private keys

```
        modulus_list.reverse()
    modulus_str = build_returned_string(modulus_list)
    modulus_str = ".moduloData =\n{\n%s\n}," % modulus_str
    if not out_file_name:
        print(modulus_str)
        print(rsaExp_str)
        print(barret_str)
        print(inv_modulo_str)
        print(r_bar_str)
    else:
        with open(out_file_name, 'w') as outfile:
            outfile.write(modulus_str + "\n")
            outfile.write(rsaExp_str  + "\n")
            outfile.write(barret_str  + "\n")
            outfile.write(inv_modulo_str + "\n")
            outfile.write(r_bar_str   + "\n")
    return 0

def extended_euclid(modulo):
    ''' Calculate greatest common divisor (GCD) of two values.
        Link: https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm
            formula to calculate: ax + by - gcd(a,b)
        parameters:
            a, b - two values witch is calculated GCD for.
        return:
            absolute values of x and y coefficients
        NOTE: pseudo-code of operation:
            x, lastX = 0, 1
            y, lastY = 1, 0
            while (b != 0):
                q = a // b
                a, b = b, a % b
                x, lastX = lastX - q * x, x
                y, lastY = lastY - q * y, y
            return (abs(lastX), abs(lastY))
    '''
    rInv = 1;
    nInv = 0;
    modulo_bit_size = modulo.bit_length()
    for i in range(modulo_bit_size):
        if not (rInv % 2):
            rInv = rInv // 2
            nInv = nInv // 2
        else:
            rInv = rInv + modulo;
            rInv = rInv // 2;
            nInv = nInv // 2;
            nInv = nInv + (1 << (modulo_bit_size - 1));
    return rInv, nInv

def calculate_additional_rsa_key_coefs(modulo):
    ''' Calculate three additional coefficients for modulo value of RSA key
        1. barret_coef - Barrett coefficient. Equation is: barretCoef = floor((2 << (2 * k)) /
```

9 Appendix A - Example of creating public and private keys

```
n);
            Main article is here: https://en.wikipedia.org/wiki/Barrett_reduction
        2. r_bar   - pre-calculated value. Equation is: r_bar = (1 << k) mod n;
        3. inverse_modulo - coefficient. It satisfying  rr' - nn' = 1, where r = 1 << k;
            Main article is here:  https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm
        parameter:
            modulo - part of RSA key
        return:
            tuple( barret_coef, r_bar, inverse_modulo ) as reversed byte arrays;
    '''
    if isinstance(modulo, str):
        modulo = int(modulo, 16)
    if modulo <= 0:
        raise ValueError("Modulus must be positive")
    if modulo & (modulo - 1) == 0:
        raise ValueError("Modulus must not be a power of 2")

    modulo_len = modulo.bit_length()
    barret_coef = (1 << (modulo_len * 2)) // modulo
    r_bar = (1 << modulo_len) % modulo
    inverse_modulo = extended_euclid(modulo)
    ret_arrays = (
        barret_coef,
        inverse_modulo[1],
        r_bar
    )
    return ret_arrays

def convert_hexstr_to_list(s, reversed=False):
    ''' Converts a string likes '0001aaff...' to list [0, 1, 170, 255].
        Also an input parameter can be an integer, in this case it will be
        converted to a hex string.
    parameter:
        s - string to convert
        reversed - a returned list have to be reversed
    return:
        a list of an integer values
    '''
    if isinstance(s, integer_types):
        s = hex(s)
    s = s[2 if s.lower().startswith("0x") else 0 : -1 if s.upper().endswith("L") else len(s)]
    if len(s) % 2 != 0:
        s = '0' + s
    l = [int("0x%s" % s[i:i+2], 16) for i in range(0, len(s), 2)]
    if reversed:
        l.reverse()
    return l

def build_returned_string(inp_list):
    ''' Converts a list to a C-style array of hexadecimal numbers string
    '''
    if isinstance(inp_list[0], int):
        inp_list = ['%02X' % x for x in inp_list]
```

```
    tmp_str = "    "
    for idx in range(0, len(inp_list)):
        if (idx % 8 == 0) and (idx != 0):
            tmp_str = tmp_str + "\n    "
        tmp_str = tmp_str + ( "0x%02Xu," % int(inp_list[idx], base=16) )
        if (idx % 8 != 7) and (idx != len(inp_list) - 1):
            tmp_str = tmp_str + " "

    return tmp_str

if __name__ == "__main__":
    main()
```

9.2.3 Running the scripts

The batch file *rsa_keygen.bash* must be called from the modus-shell of ModusToolbox™ software. After the script runs, verify that the following files have been generated in the *keys_generated* folder (b).

- *rsa_private_generated.txt* (Private RSA key)
- *rsa_public_generated.txt* (Public RSA key)
- *rsa_to_c_generated.txt* (Public key in C format)

An example of what the private key file (RSA-2048) should look like:

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAm4rqAvD0qKP2okMMRfwgPBhvitxf31m4Jp254b2F+BJxcH8g
rdaEaMd8G+6uR8Aqs7PqpKj4NIq0FK3hc1X62dywI5LRcfZ0fpVnv0O6KBhUc8Qx
fSYl+U5eiFSBvANRsZUiIhTn2muWVdJVsJ6Hjejb78FKOEfmjrrEtt1Mm5lep+9L
I6XMQMDsS//ZILgo0eYFOSNjJscleFVD4ThAZyrWx200NbneHgB7KmxSYgmop7ub
YDpiDzA0LgYCdq6KYMPDUGBlMUFCtJnC1zyIG+Wd72fgT+rrw/4uePQiJl75mpmY
6CWHlT+QvQqX2DbU6wlWK0UlGWSqEMi8cLN77wIDAQABAoIBAGZO6fuDSDoAl0JF
fbYuz3kXzY51w63ikqj7x/8+rPbuEqWfSPxvBGrzA6RPy1ywfqXy6pzh81iD/002
zfsSC8zvRwAvGwqtxBRa64G356Nl1W7MFo0t4X5pWO6tPyROvpZi9K+cNqDJcXoD
vo9MxvbFupyp88qvKGyrVqa9Sx1wrly20EPbIDBvu0LcHVMosQ+H4Hs5QR6kLoOl
xeupyrl5AH7QQq3yVkBg0Nkiv5M09L5NIpjxf/lgKbV5q8zd3P3B31nZMKx1GYXS
6hYqFMULq8B9oc2tQ4B1G0FfC1dT9nRyRxcWK7Z0+lEwqyQdmCUlWhQm0Jb+1j3d
sKng+bECgYEAzaLjPGlFWy72IljbnQ5eVyR453JvfTizC3+J6VUT2ONLL4768pNh
tXOJsr2dEcHWHZOM+pqjYYr/hqRV8biHwCITIW/Yi5YE5Pp5l/uoSUiXobRK5/dr
wWwZpUTPj2t7uvUMFdR3VGLmWni0Uz7rCM3ue7aQKX+YORVURZ0vdBkCgYEAwaM5
PsxEipW1jcf7kjqXaf+EdJNmV65A31oTHRvfYeI3tmy4ZUPjmsMXZ+dJN6ukX4XW
gQv5kTnFd0EVsuhv6IRJWgCk3QwH9QWoUyO4tyYchFtpWh/uHIB5aMUBCaH3j/UD
qljvqFuWtvCjCql7C02Y8trlHhsWdcPBlMQNsUcCgYEAo5wY7iTX0N/BZ7yJJyGa
f/z90TAkcl4dXmWgJazSOLtHQ9Sf50bJ8+O0SClDAjDWGfweq/lDSsWb3tV7p/Ho
2D6EOBNUGGNJC6IaAolw8LW4Jylae8tIycEnem4QMo0TCcIUVn4QX2y82Hi3CJDD
0eJ+/6FWfB4P0EMfJLAWZGkCgYEAqeTpTCTzGljiU4ScQJ+xy+nJsHLHf7S+7Hje
flK/OuW9lgxr+UWIhHgZqCtwvSyW6szF+LGmBXqXu9taC4AO+AZhSVt5SwzMSnnt
Rwz3e5IvwR1nIREU3OMIIxl+oJIAYMlLrwGbPK8nXFlj4+3xrGgoAyR1lGWV3juk
ND+T9+8CgYA+ByTcFr1fNWextJwgnFEWxGo+KSMEUoueDibirzRM5t7W7VWiH296
os7mf62BKlNMxLPCeCxPhHPEK7LXlOSb9RVdeU1lTXeo8d3RsCTYUJaqjsOr4j67
cxNOFqVx74QvCycW48CbAXxpGa+xg8lL9Eonimdkpf60FLxC0PyMsQ==
-----END RSA PRIVATE KEY-----
```

9 Appendix A - Example of creating public and private keys

An example of what the public key file (RSA-2048) should look like:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAm4rqAvD0qKP2okMMRfwg
PBhvitxf31m4Jp254b2F+BJxcH8grdaEaMd8G+6uR8Aqs7PqpKj4NIq0FK3hc1X6
2dywI5LRcfZ0fpVnv0O6KBhUc8QxfSYl+U5eiFSBvANRsZUiIhTn2muWVdJVsJ6H
jejb78FKOEfmjrrEtt1Mm5lep+9LI6XMQMDsS//ZILgo0eYFOSNjJscleFVD4ThA
ZyrWx200NbneHgB7KmxSYgmop7ubYDpiDzA0LgYCdq6KYMPDUGBlMUFCtJnC1zyI
G+Wd72fgT+rrw/4uePQiJl75mpmY6CWHlT+QvQqX2DbU6wlWK0UlGWSqEMi8cLN7
7wIDAQAB
-----END PUBLIC KEY-----
```

9.3 Installing the public key

The last step to updating the public key in the secure image is to copy the code in the generated *rsa_to_c_generated.txt* file to the *secure_boot_key.c* source file, which is part of the secure image project. An example of this file after being updated is shown below. The replacement code from the generated key data is shown below in bold. The following is an example of RSA-2048:

9 Appendix A - Example of creating public and private keys

```
/** Public key in SFlash */
CY_SECTION(".cy_sflash_public_key") __USED const cy_si_stc_public_key_t cy_publicKey =
{
    .objSize = sizeof(cy_si_stc_public_key_t),
    .signatureScheme = CY_SI_PUBLIC_KEY_RSA_2048,
    .publicKeyStruct =
    {
        .moduloAddr         = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_si_stc_public_key_t, moduloData),
        .moduloSize         = CY_SI_PUBLIC_KEY_SIZEOF_BYTE * CY_SI_PUBLIC_KEY_MODULOLENGTH,
        .expAddr            = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_si_stc_public_key_t, expData),
        .expSize            = CY_SI_PUBLIC_KEY_SIZEOF_BYTE * CY_SI_PUBLIC_KEY_EXPLENGTH,
        .barrettAddr        = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_si_stc_public_key_t, barrettData),
        .inverseModuloAddr  = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_si_stc_public_key_t, inverseModuloData),
        .rBarAddr           = (uint32_t)&(SFLASH->PUBLIC_KEY) +
offsetof(cy_si_stc_public_key_t, rBarData),
    }
```

```
.moduloData =
{
0xEFu, 0x7Bu, 0xB3u, 0x70u, 0xBCu, 0xC8u, 0x10u, 0xAAu,
0x64u, 0x19u, 0x25u, 0x45u, 0x2Bu, 0x56u, 0x09u, 0xEBu,
0xD4u, 0x36u, 0xD8u, 0x97u, 0x0Au, 0xBDu, 0x90u, 0x3Fu,
0x95u, 0x87u, 0x25u, 0xE8u, 0x98u, 0x99u, 0x9Au, 0xF9u,
0x5Eu, 0x26u, 0x22u, 0xF4u, 0x78u, 0x2Eu, 0xFEu, 0xC3u,
0xEBu, 0xEAu, 0x4Fu, 0xE0u, 0x67u, 0xEFu, 0x9Du, 0xE5u,
0x1Bu, 0x88u, 0x3Cu, 0xD7u, 0xC2u, 0x99u, 0xB4u, 0x42u,
0x41u, 0x31u, 0x65u, 0x60u, 0x50u, 0xC3u, 0xC3u, 0x60u,
0x8Au, 0xAEu, 0x76u, 0x02u, 0x06u, 0x2Eu, 0x34u, 0x30u,
0x0Fu, 0x62u, 0x3Au, 0x60u, 0x9Bu, 0xBBu, 0xA7u, 0xA8u,
0x09u, 0x62u, 0x52u, 0x6Cu, 0x2Au, 0x7Bu, 0x00u, 0x1Eu,
0xDEu, 0xB9u, 0x35u, 0x34u, 0x6Du, 0xC7u, 0xD6u, 0x2Au,
0x67u, 0x40u, 0x38u, 0xE1u, 0x43u, 0x55u, 0x78u, 0x25u,
0xC7u, 0x26u, 0x63u, 0x23u, 0x39u, 0x05u, 0xE6u, 0xD1u,
0x28u, 0xB8u, 0x20u, 0xD9u, 0xFFu, 0x4Bu, 0xECu, 0xC0u,
0x40u, 0xCCu, 0xA5u, 0x23u, 0x4Bu, 0xEFu, 0xA7u, 0x5Eu,
0x99u, 0x9Bu, 0x4Cu, 0xDDu, 0xB6u, 0xC4u, 0xBAu, 0x8Eu,
0xE6u, 0x47u, 0x38u, 0x4Au, 0xC1u, 0xEFu, 0xDBu, 0xE8u,
0x8Du, 0x87u, 0x9Eu, 0xB0u, 0x55u, 0xD2u, 0x55u, 0x96u,
0x6Bu, 0xDAu, 0xE7u, 0x14u, 0x22u, 0x22u, 0x95u, 0xB1u,
0x51u, 0x03u, 0xBCu, 0x81u, 0x54u, 0x88u, 0x5Eu, 0x4Eu,
0xF9u, 0x25u, 0x26u, 0x7Du, 0x31u, 0xC4u, 0x73u, 0x54u,
0x18u, 0x28u, 0xBAu, 0x43u, 0xBFu, 0x67u, 0x95u, 0x7Eu,
0x74u, 0xF6u, 0x71u, 0xD1u, 0x92u, 0x23u, 0xB0u, 0xDCu,
0xD9u, 0xFAu, 0x55u, 0x73u, 0xE1u, 0xADu, 0x14u, 0xB4u,
0x8Au, 0x34u, 0xF8u, 0xA8u, 0xA4u, 0xEAu, 0xB3u, 0xB3u,
0x2Au, 0xC0u, 0x47u, 0xAEu, 0xEEu, 0x1Bu, 0x7Cu, 0xC7u,
0x68u, 0x84u, 0xD6u, 0xADu, 0x20u, 0x7Fu, 0x70u, 0x71u,
```

9 Appendix A - Example of creating public and private keys

```
0x12u, 0xF8u, 0x85u, 0xBDu, 0xE1u, 0xB9u, 0x9Du, 0x26u,
0xB8u, 0x59u, 0xDFu, 0x5Fu, 0xDCu, 0x8Au, 0x6Fu, 0x18u,
0x3Cu, 0x20u, 0xFCu, 0x45u, 0x0Cu, 0x43u, 0xA2u, 0xF6u,
0xA3u, 0xA8u, 0xF4u, 0xF0u, 0x02u, 0xEAu, 0x8Au, 0x9Bu,
},
.expData =
{
0x01u, 0x00u, 0x01u, 0x00u,
},
.barrettData =
{
0x2Au, 0x9Bu, 0x24u, 0x67u, 0x71u, 0x2Du, 0x7Au, 0xF2u,
0x9Du, 0x65u, 0x82u, 0x8Fu, 0x62u, 0xDDu, 0x6Du, 0xA2u,
0x34u, 0x87u, 0xBCu, 0x7Au, 0x63u, 0xDFu, 0xBAu, 0x68u,
0x10u, 0x8Eu, 0x6Cu, 0x75u, 0x02u, 0xB0u, 0x3Bu, 0x71u,
0xDBu, 0xB3u, 0x45u, 0x33u, 0xEAu, 0x4Bu, 0x0Au, 0xE5u,
0xA8u, 0x36u, 0xBBu, 0x51u, 0x9Bu, 0x44u, 0x7Fu, 0x07u,
0x1Eu, 0xD2u, 0x37u, 0x9Du, 0x54u, 0x86u, 0x1Fu, 0xDEu,
0x92u, 0x39u, 0xD1u, 0x9Au, 0x1Eu, 0x79u, 0x04u, 0xF3u,
0x3Fu, 0xCAu, 0x88u, 0x94u, 0x5Au, 0xD8u, 0x34u, 0xCCu,
0x93u, 0x11u, 0x80u, 0x11u, 0x14u, 0xDCu, 0xFEu, 0xEEu,
0x3Au, 0xBCu, 0x95u, 0xA2u, 0x52u, 0x39u, 0xE5u, 0xBCu,
0x02u, 0xFCu, 0x98u, 0x61u, 0xAEu, 0xE9u, 0xA9u, 0x26u,
0x45u, 0xC8u, 0x23u, 0xCDu, 0x05u, 0x09u, 0xB0u, 0x4Bu,
0x76u, 0x9Bu, 0x84u, 0x77u, 0x6Bu, 0x8Du, 0xBDu, 0x87u,
0x25u, 0xF7u, 0x52u, 0x30u, 0x75u, 0x21u, 0x9Fu, 0xA2u,
0x41u, 0x3Du, 0xFBu, 0xB2u, 0x28u, 0x68u, 0x60u, 0x26u,
0xFEu, 0xEFu, 0x91u, 0x20u, 0xD2u, 0x7Cu, 0xACu, 0xF8u,
0x76u, 0x4Fu, 0x48u, 0x5Bu, 0xEFu, 0x86u, 0x7Cu, 0x7Cu,
0x5Fu, 0x24u, 0x5Au, 0x60u, 0x69u, 0x91u, 0x60u, 0xA9u,
0x92u, 0xC7u, 0xC0u, 0xFEu, 0xF1u, 0x8Eu, 0xA5u, 0xA7u,
0x64u, 0xB8u, 0x53u, 0x27u, 0x7Eu, 0x03u, 0xDAu, 0xF6u,
0xCEu, 0xB8u, 0xA6u, 0x66u, 0x07u, 0xCAu, 0x18u, 0xA6u,
0x05u, 0xC5u, 0x32u, 0x9Fu, 0x2Fu, 0xDDu, 0x69u, 0x75u,
0xAAu, 0x7Au, 0x45u, 0x2Au, 0x27u, 0xAFu, 0xC2u, 0x94u,
0xFCu, 0x66u, 0x25u, 0xD2u, 0x5Bu, 0x1Bu, 0x7Fu, 0xE3u,
0x17u, 0x0Du, 0xCAu, 0x08u, 0xE3u, 0x5Du, 0xE7u, 0x6Au,
0x3Eu, 0x99u, 0x3Au, 0xA2u, 0x5Au, 0xDCu, 0xECu, 0x24u,
0x60u, 0x0Eu, 0xF6u, 0xBEu, 0xCCu, 0x43u, 0xBCu, 0x72u,
0x06u, 0x62u, 0xE4u, 0xB5u, 0x17u, 0x0Fu, 0x8Eu, 0x4Eu,
0x60u, 0x75u, 0xDCu, 0x93u, 0xA2u, 0xBDu, 0x06u, 0xFCu,
0x55u, 0xACu, 0x78u, 0x60u, 0x92u, 0x0Du, 0x35u, 0x1Du,
0xDAu, 0xB2u, 0x9Au, 0xF5u, 0xE5u, 0x7Du, 0x56u, 0xA5u,
0x01u, 0x00u, 0x00u, 0x00u,
},
.inverseModuloData =
{
0xF1u, 0xECu, 0xB8u, 0x73u, 0x69u, 0xA8u, 0xEFu, 0xB8u,
0xDDu, 0xB0u, 0x48u, 0x1Du, 0xF2u, 0x47u, 0x39u, 0xD4u,
0xC1u, 0x88u, 0x9Au, 0x3Fu, 0x2Fu, 0x1Eu, 0x13u, 0xDCu,
0x85u, 0x88u, 0xDAu, 0xC7u, 0x04u, 0x76u, 0x89u, 0x28u,
0x64u, 0xABu, 0xF4u, 0x33u, 0x6Eu, 0x2Cu, 0x45u, 0x6Eu,
0x69u, 0xA6u, 0x63u, 0x0Fu, 0x1Cu, 0x21u, 0x4Eu, 0x1Cu,
```

9 Appendix A - Example of creating public and private keys

```
0xD4u, 0x13u, 0xB0u, 0xEDu, 0x6Cu, 0x75u, 0xCFu, 0xD9u,
0xF5u, 0x12u, 0x7Eu, 0xB7u, 0xC0u, 0x4Eu, 0x64u, 0x71u,
0xC7u, 0xC7u, 0xF3u, 0x95u, 0x8Au, 0x7Fu, 0x83u, 0xADu,
0xC8u, 0x30u, 0x05u, 0xD3u, 0x93u, 0xD2u, 0x31u, 0x31u,
0x30u, 0x79u, 0x8Au, 0x7Eu, 0x69u, 0x47u, 0xCFu, 0x5Eu,
0x39u, 0xE4u, 0x84u, 0x88u, 0x2Fu, 0x84u, 0xB6u, 0xD2u,
0x3Eu, 0xA4u, 0xB2u, 0x4Fu, 0xBFu, 0x26u, 0x0Du, 0x0Bu,
0x5Eu, 0xFCu, 0xDEu, 0x08u, 0xECu, 0x94u, 0xF9u, 0x61u,
0xD9u, 0xE2u, 0x0Cu, 0xC7u, 0xCFu, 0xD5u, 0xA4u, 0xE2u,
0xC4u, 0xC6u, 0x2Fu, 0x01u, 0x13u, 0x8Au, 0xAAu, 0xEDu,
0xE5u, 0xAAu, 0xE9u, 0xEBu, 0x2Fu, 0x18u, 0x5Eu, 0xFFu,
0x4Eu, 0xC0u, 0xEBu, 0x24u, 0x37u, 0xD0u, 0x36u, 0x17u,
0xD5u, 0xCDu, 0x84u, 0x49u, 0xB3u, 0x6Du, 0xDFu, 0x9Cu,
0x80u, 0x30u, 0x00u, 0xD9u, 0x0Cu, 0x5Fu, 0xE8u, 0xC5u,
0x8Eu, 0x63u, 0x05u, 0xB7u, 0x70u, 0x37u, 0x7Du, 0x7Fu,
0x9Fu, 0x10u, 0xC7u, 0x9Fu, 0x24u, 0xA1u, 0x23u, 0x3Eu,
0x2Fu, 0x05u, 0x8Fu, 0xCDu, 0x38u, 0x68u, 0xA1u, 0x8Du,
0x1Du, 0xD3u, 0xF3u, 0x81u, 0x5Bu, 0x74u, 0x5Bu, 0xEBu,
0x33u, 0x1Fu, 0xC5u, 0x30u, 0x65u, 0x70u, 0x58u, 0x8Eu,
0x64u, 0x2Du, 0x51u, 0x12u, 0x91u, 0xC9u, 0x79u, 0x68u,
0x93u, 0x34u, 0x9Eu, 0xEEu, 0xB6u, 0x58u, 0x2Au, 0x3Cu,
0xDBu, 0x72u, 0xBFu, 0x3Cu, 0x94u, 0xD8u, 0x41u, 0x9Bu,
0xDAu, 0x06u, 0x50u, 0xE8u, 0xDEu, 0xE6u, 0x06u, 0x61u,
0x7Cu, 0xBCu, 0xF8u, 0x69u, 0xF8u, 0x5Du, 0x2Au, 0x35u,
0x2Cu, 0xBBu, 0x5Cu, 0x7Cu, 0x16u, 0x9Au, 0xFAu, 0x65u,
0x80u, 0x55u, 0x01u, 0x64u, 0x8Fu, 0xAFu, 0x69u, 0x53u,
},
.rBarData =
{
0x11u, 0x84u, 0x4Cu, 0x8Fu, 0x43u, 0x37u, 0xEFu, 0x55u,
0x9Bu, 0xE6u, 0xDAu, 0xBAu, 0xD4u, 0xA9u, 0xF6u, 0x14u,
0x2Bu, 0xC9u, 0x27u, 0x68u, 0xF5u, 0x42u, 0x6Fu, 0xC0u,
0x6Au, 0x78u, 0xDAu, 0x17u, 0x67u, 0x66u, 0x65u, 0x06u,
0xA1u, 0xD9u, 0xDDu, 0x0Bu, 0x87u, 0xD1u, 0x01u, 0x3Cu,
0x14u, 0x15u, 0xB0u, 0x1Fu, 0x98u, 0x10u, 0x62u, 0x1Au,
0xE4u, 0x77u, 0xC3u, 0x28u, 0x3Du, 0x66u, 0x4Bu, 0xBDu,
0xBEu, 0xCEu, 0x9Au, 0x9Fu, 0xAFu, 0x3Cu, 0x3Cu, 0x9Fu,
0x75u, 0x51u, 0x89u, 0xFDu, 0xF9u, 0xD1u, 0xCBu, 0xCFu,
0xF0u, 0x9Du, 0xC5u, 0x9Fu, 0x64u, 0x44u, 0x58u, 0x57u,
0xF6u, 0x9Du, 0xADu, 0x93u, 0xD5u, 0x84u, 0xFFu, 0xE1u,
0x21u, 0x46u, 0xCAu, 0xCBu, 0x92u, 0x38u, 0x29u, 0xD5u,
0x98u, 0xBFu, 0xC7u, 0x1Eu, 0xBCu, 0xAAu, 0x87u, 0xDAu,
0x38u, 0xD9u, 0x9Cu, 0xDCu, 0xC6u, 0xFAu, 0x19u, 0x2Eu,
0xD7u, 0x47u, 0xDFu, 0x26u, 0x00u, 0xB4u, 0x13u, 0x3Fu,
0xBFu, 0x33u, 0x5Au, 0xDCu, 0xB4u, 0x10u, 0x58u, 0xA1u,
0x66u, 0x64u, 0xB3u, 0x22u, 0x49u, 0x3Bu, 0x45u, 0x71u,
0x19u, 0xB8u, 0xC7u, 0xB5u, 0x3Eu, 0x10u, 0x24u, 0x17u,
0x72u, 0x78u, 0x61u, 0x4Fu, 0xAAu, 0x2Du, 0xAAu, 0x69u,
0x94u, 0x25u, 0x18u, 0xEBu, 0xDDu, 0xDDu, 0x6Au, 0x4Eu,
0xAEu, 0xFCu, 0x43u, 0x7Eu, 0xABu, 0x77u, 0xA1u, 0xB1u,
0x06u, 0xDAu, 0xD9u, 0x82u, 0xCEu, 0x3Bu, 0x8Cu, 0xABu,
0xE7u, 0xD7u, 0x45u, 0xBCu, 0x40u, 0x98u, 0x6Au, 0x81u,
0x8Bu, 0x09u, 0x8Eu, 0x2Eu, 0x6Du, 0xDCu, 0x4Fu, 0x23u,
```

9 Appendix A - Example of creating public and private keys

```
0x26u, 0x05u, 0xAAu, 0x8Cu, 0x1Eu, 0x52u, 0xEBu, 0x4Bu,
0x75u, 0xCBu, 0x07u, 0x57u, 0x5Bu, 0x15u, 0x4Cu, 0x4Cu,
0xD5u, 0x3Fu, 0xB8u, 0x51u, 0x11u, 0xE4u, 0x83u, 0x38u,
0x97u, 0x7Bu, 0x29u, 0x52u, 0xDFu, 0x80u, 0x8Fu, 0x8Eu,
0xEDu, 0x07u, 0x7Au, 0x42u, 0x1Eu, 0x46u, 0x62u, 0xD9u,
0x47u, 0xA6u, 0x20u, 0xA0u, 0x23u, 0x75u, 0x90u, 0xE7u,
0xC3u, 0xDFu, 0x03u, 0xBAu, 0xF3u, 0xBCu, 0x5Du, 0x09u,
0x5Cu, 0x57u, 0x0Bu, 0x0Fu, 0xFDu, 0x15u, 0x75u, 0x64u,
},
};
```

The *Cy_FB_Isvalidkey* function of flash boot can check whether the public key structure is valid. See the “Flash boot” chapter in the architecture TRM [2] for more details.

10 Appendix B - Requirements for generating a digital signature

The *cymcuelftool.exe* and *edge protect tools* can be used to generate a digital signature using the private key and adds digital signature to the ELF file. The generated digital signature is then decrypted using the public key stored in the SFlash and is validated by flash boot.

The following is an example of using the edge protect tools to sign the application in Infineon secure application format (CySAF).

```
 edgeprotecttools -t xmc7100 sign-cysaf --image image.elf --output image_signed.elf --key-path
private.pem
```

The *cymcuelftool.exe* tool is located in the ModusToolbox™ software tools folders. The following is an example of using the *cymcuelftool.exe* tool.

```
> cymcuelftool.exe –-sign [in.elf] [HASH ALGORITHM] –-encrypt [ENC_ALGORITHM]
–-key [PrivateKey] –-output [out.elf]
```

The tool requires the following symbols/sections for digital signature generation.

1. ".cy_app_signature" section:

   This section specifies the location the digital signature is written to:

   ```
   /** Secure Image Digital signature (Populated by cymcuelftool) */
   CY_SECTION(".cy_app_signature") __USED CY_ALIGN(4)
   static const uint8_t cy_si_appSignature[CY_SI_SECURE_DIGSIG_SIZE] = {0u};
   ```

2. "__cy_app_verify_start" and "__cy_app_verify_length" symbols:

   This symbol defines the first address and size of the memory area whose digital signature is calculated.

   ```
   __cy_app_verify_start = 0x10000000;
   __cy_app_verify_length = 0xFE00;
   ```

In this example, the digital signature calculation area is from 0x10000000 to 0x1000FE00. Note that in the digital signature generated, blanks in the calculation area expect "0".

11 Appendix C - Authentication of the main user application

The main user application is authenticated by the secure image. For CoT, the main user application must be authenticated before the main CPU (CM7 or CM4s) is activated.

XMC7000/XMC5000 MCU supports the `Cy_FB_VerifyApplication` function to authenticate user applications. This function is included in flash boot and can be executed from the user code. This function can be used to authenticate the other code with RSASSA-PKCS1-v1.5. See the “Flash boot” chapter in the architecture TRM [2] for more details.

In this example, a digital signature is generated using the same method as the first user application, and authentication is performed using the public key in the SFlash.

11 Appendix C - Authentication of the main user application

```
#define CY_SI_IMGVAL_VERIFYAPP_ADDR     ((volatile uint32_t *)0x17002040UL)
#define CY_SI_IMGVAL_VERIFYAPP_REG      (*(uint32_t *)CY_SI_IMGVAL_VERIFYAPP_ADDR)

typedef uint32_t (*sflash_verifyapp_func_t)(uint32_t param0, uint32_t param1, uint32_t param2,
                                    cy_stc_crypto_rsa_pub_key_t *param3);

__STATIC_INLINE uint32_t Cy_FB_VerifyApplication(uint32_t address, uint32_t length,
                                        uint32_t signature, cy_stc_crypto_rsa_pub_key_t
*publicKey)
{
    sflash_verifyapp_func_t fp = (sflash_verifyapp_func_t)CY_SI_IMGVAL_VERIFYAPP_REG;
    return ( fp(address, length, signature, publicKey) );
}

int main(void)
{
    bool isAppValid;
    cy_stc_crypto_rsa_pub_key_t *publicKey = NULL;
    uint32_t appDataAddr;
    uint32_t appDataLength;
uint32_t appSignatureAddr;

    /* enable interrupts */
    __enable_irq();

    // These symbols are defined in the linker script
    extern const char __cm7_0_vector_base_linker_symbol[];
    extern const char __cm7_1_vector_base_linker_symbol[];
    extern const char __cm7_0_si_size_linker_symbol[];
    extern const char __cm7_1_si_size_linker_symbol[];

    /* Verify the CM7_0 application */
    appDataAddr = (uint32_t)__cm7_0_vector_base_linker_symbol;
    appDataLength = (uint32_t)__cm7_0_si_size_linker_symbol;
    appSignatureAddr = (uint32_t)__cm7_0_vector_base_linker_symbol +
(uint32_t)__cm7_0_si_size_linker_symbol;
    publicKey = (cy_stc_crypto_rsa_pub_key_t *)(CY_SI_PUBLIC_KEY + CY_FB_PBKEY_STRUCT_OFFSET);
    isAppValid = Cy_FB_VerifyApplication(appDataAddr, appDataLength, appSignatureAddr,
publicKey);
    /* RSA Verify Error Handling Here */
    while(isAppValid == false);

    /* Verify the CM7_1 application */
    appDataAddr = (uint32_t)__cm7_1_vector_base_linker_symbol;
    appDataLength = (uint32_t)__cm7_1_si_size_linker_symbol;
    appSignatureAddr = (uint32_t)__cm7_1_vector_base_linker_symbol +
(uint32_t)__cm7_1_si_size_linker_symbol;
    publicKey = (cy_stc_crypto_rsa_pub_key_t *)(CY_SI_PUBLIC_KEY + CY_FB_PBKEY_STRUCT_OFFSET);
    isAppValid = Cy_FB_VerifyApplication(appDataAddr, appDataLength, appSignatureAddr,
publicKey);
    /* RSA Verify Error Handling Here */
    while(isAppValid == false);
```

11 Appendix C - Authentication of the main user application

```
    /* Enable CM7_0 */
    Cy_SysEnableCM7(CORE_CM7_0, (uint32_t)__cm7_0_vector_base_linker_symbol);

    /* To avoid race between CM7's while initializing the clocks */
    Cy_SysLib_Delay(1000);
    /* Enable CM7_1 */
    Cy_SysEnableCM7(CORE_CM7_1, (uint32_t)__cm7_1_vector_base_linker_symbol);

    Cy_SysPm_CpuSleepOnExit(true);
    /* Update system core clock for delay functions */
    SystemCoreClockUpdate();

    for(;;)
    {
        Cy_SysPm_CpuEnterDeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT);
    }
}
```

This function requires the parameters listed in Table 8:

Table 8 Cy_FB_VerifyApplication function parameters

Parameter	Description
appDataAddr	Starting address of the application area to be verified with secure signature
appDataLength	Length of the area to be verified
appSignatureAddr	Starting address of the signature inside the application residing in flash
publicKey	Pointer to the public key structure

12 Appendix D - Read device Unique ID

The Unique ID is a 12 bytes ID resides in the device SFlash and is unique per silicon die. To read the Unique ID from SFlash, use the ReadUniqueID API (System call 0x1F). The following code snippet demonstrates an example of the ReadUniqueID API.

```
#define CY_IPC_STRUCT  (Cy_IPC_Drv_GetIpcBaseAddress(CY_IPC_CHAN_SYSCALL)) /* IPC structure to
be used */
#define CY_SROM_DR_IPC_NOTIFY_STRUCT  (0x1UL)
#define CY_IPC_DRV_SUCCESS  (0x00U)

#define UID_OPCODE 0x1F000000  /* opcode to read uniqueID */

typedef struct
{
    uint32_t unique_id_0;
    uint32_t unique_id_1;
    uint32_t unique_id_2;
}read_uid_param_t;

read_uid_param_t uid_param_UID =
{
    .unique_id_0 = UID_OPCODE
};

if(Cy_IPC_Drv_SendMsgWord(CY_IPC_STRUCT, CY_SROM_DR_IPC_NOTIFY_STRUCT,
(uint32_t)&uid_param_UID) == CY_IPC_DRV_SUCCESS)
{
  printf("ReadUniqueID : Device unique ID is: 0x%08lx 0x%08lx 0x%08lx \r\n",
uid_param_UID.unique_id_0, uid_param_UID.unique_id_1, uid_param_UID.unique_id_2);
}
else
  printf("IPC call failed\r\n");
```

An example output of this call would be as follows:

```
ReadUniqueID : Device unique ID is: 0xa08e0f90 0x670c2f19 0x7b070900
```

The bit[31:28] of unique_id_0 always represents the system call status on return from the IPC call.

0xA0 indicates success and **0xF0** indicates failure of the system call.

The ReadUniqueID() system call returns a 11 bytes unique ID along with a 1 byte status. To make it a whole 12 bytes Unique ID, remove the status byte and append 0x00 at the end as shown below.



12 Appendix D - Read device Unique ID

Thus, the actual unique id will be as shown below

```
0x198e0f90 0x00670c2f 0x007b0709
```

For more information, refer to ReadUniqueID in the Architecture TRM [2] document.

13 Appendix E - Transition to SECURE/SECURE_WITH_DEBUG lifecycle stage

A device can be transitioned into the SECURE/SECURE_WITH_DEBUG lifecycle stage only from the NORMAL_PROVISIONED state and not vice versa. Similarly, the device once transitioned to SECURE stage, it cannot be moved to SECURE_WITH_DEBUG and vice versa.

During the transition to SECURE, it validates the FACTORY_HASH and programs SECURE_HASH, secure access restrictions, and dead access restrictions into eFuse. We use an IPC call to transition the device to various lifecycle stages.

The following code snippet demonstrates how to move the device from NORMAL_PROVISIONED to SECURE/ SECURE_WITH_DEBUG based on the appropriate opcode provided to the IPC call.

```
#define CY_IPC_STRUCT  (Cy_IPC_Drv_GetIpcBaseAddress(CY_IPC_CHAN_SYSCALL)) /* IPC structure to
be used */
#define CY_SROM_DR_IPC_NOTIFY_STRUCT  (0x1UL)
#define CY_IPC_DRV_SUCCESS  (0x00U)

/* opcode for LCS transitions */
#define SEC_LCS_OPCODE  0x2F000100
#define SEC_DBG_LCS_OPCODE  0x2F000000

uint32_t secure_lcs = SEC_LCS_OPCODE;

if(Cy_IPC_Drv_SendMsgWord(CY_IPC_STRUCT, CY_SROM_DR_IPC_NOTIFY_STRUCT, (uint32_t)&secure_lcs)
== CY_IPC_DRV_SUCCESS)
{
    printf("SECURE LCS status: 0x%08lx \r\n", secure_lcs);
}
else
   printf("IPC call failed\r\n");
```

If the LCS transition is successful, the status code is returned back in the same argument that is used to send the opcode.

The bits [31:28] in the returned argument represents **0xA** if success and **0xF** in case of failure.

For more information on the SECURE Lifecycle stage, refer to Transition to Secure in the Architecture TRM document.

14 Appendix F - Transition to RMA lifecycle stage

The customer transitions the part to the RMA lifecycle stage when they want to perform failure analysis on the part.

To transition a device to the RMA lifecycle stage, it is mandatory to know the device Unique ID and the customer's private key that is paired with a public key stored in the SFlash. The customer must implement at least two special commands that can be sent from outside the device via UART, SPI, I^2C, CAN, and so forth.

- First command: Read the internal Unique device ID (Appendix D - Read device Unique ID)
- Second command: Invoke the transition to the RMA lifecycle stage

If Sys-DAP is available, these two special commands can be run via Sys-DAP.

The customer must generate a certificate to transition the part with a specific Unique ID to the RMA lifecycle stage. After the part has been transitioned to RMA, the customer must provide another certificate that is signed by using their private key for failure analysis on the part.

***Note**: When transitioning to the RMA lifecycle stage, it is recommended to call with the ROM boot default clock configuration settings. See "Blow Fuse Bit" in the architecture TRM [2] for more details.*

***Note**: To run the TransitiontoRMA and OpenRMA APIs, each part requires a different certificate because each part has a different ID (Unique ID). Therefore, the part and certificate are recommended to be linked by marking, if a customer returns multiple parts for failure analysis.*

Do the following to transition the device to the RMA stage:

1. If the device is in the NORMAL_PROVISIONED lifecycle stage:
   - If the device does not have public key, write the public key to the SFlash
   - If the device is programmed with the user code without the CySAF format, the device needs to be reprogrammed with the CySAF format or programmed with a dummy CySAF format image before transitioning to the SECURE or SECURE_W_DEBUG lifecycle stage. See User application block for details
   - Transition the device to the SECURE or SECURE_WITH_DEBUG lifecycle stage

   This process (1) is not required if the device is in the SECURE or SECURE_W_DEBUG lifecycle stage
2. Erase all sensitive or proprietary code stored in the device. This may be performed with a special command or with a special code image described earlier. However, note that the device with a protection status of DEAD cannot transition to the RMA lifecycle stage. Note that it is possible to transition the device to the DEAD protection state by erasing sensitive or proprietary code. The public key stored in the SFlash cannot be erased because it is used to transition to the RMA lifecycle stage, and to open the RMA device later

   ***Note**: The device may transition to the DEAD state by erasing the secure image or digital signature for authentication. As a result, you cannot transition to the RMA lifecycle stage. If you need to erase your code, you will need to prepare and write a signed dummy code and digital signature, or implement a second application in advance. See RMA for more details*
3. Read the device unique ID stored in the device's SFlash. This can be done by invoking a system call 0x1F (ReadUniqueID), and then sending the ID out via the communication interface or Sys-DAP
4. Use the unique ID and the customer's private key that is paired with the public key stored internally in the SFlash to generate a certificate. Table 9 shows the format of the certificate
5. Send a command to the device that includes this certificate. Implement code to accept this certificate to invoke the transition to RMA system call (0x28) and pass the certificate as its parameter

14 Appendix F - Transition to RMA lifecycle stage

***Note**:* *See Requirements for transition to RMA lifecycle stage for conditions on TransitiontoRMA API execution*

6. After the device is reset or power cycled, it will wait in a state for a single command from the debug port to open RMA (System call 0x29). The device will only allow the OpenRMA API until the OpenRMA API runs successfully

***Note**:* *See Requirements for transition to RMA lifecycle stage for conditions OpenRMA API execution*

Table 9 RMA certificate format

Object	Bytes
Object Size	4 bytes
Command ID	4 bytes TransitiontoRMA: 0x120028F0 OpenRMA: 0x120029F0
Unique ID	11 bytes
Zero Padding	1 byte
Digital Signature	Up to 256 bytes for RSA-2048 Up to 384 bytes for RSA-3072 Up to 512 bytes for RSA-4096

***Note**:* *The certificates and digital signatures for TransitiontoRMA API and OpenRMA API are different.*

14 Appendix F - Transition to RMA lifecycle stage

The following code snippet demonstrates how to transition a device into the RMA lifecycle stage from an application.

```
#define IPC_STATUS_WAIT_TIME_S    60u  /* maximum wait time for IPC Lock */
#define DELAY_IPC_STATUS_CHECK_MS    1000u  /* delay for checking the IPC status */
#define CY_OPCODE_STS_Msk    (0xF0000000UL)  /* The status mask of the SROM API return value */
#define CY_OPCODE_SUCCESS    (0xA0000000UL)  /* The command completed with no errors */

typedef struct {
    uint32_t opcode;
    uint32_t obj_size;
    uint32_t cmd_id;
    uint32_t unique_id_0;
    uint32_t unique_id_1;
    uint32_t unique_id_2;
    uint8_t signature[256];
} transit_rma_param_t;

/* RMA Certificate to be sent to the device */
transit_rma_param_t rmaParam =
{
    .opcode = 0x28000000,
    .obj_size = 0x00000014,
    .cmd_id = 0x120028F0,
    .unique_id_0 = 0x198e0f90,
    .unique_id_1 = 0x00670c2f,
    .unique_id_2 = 0x007b0709,
    .signature = {0x3d, 0x16, 0x27, 0x94, 0x72, 0x34, 0x34, 0x72, 0x5d, 0xc6, 0x03, 0x60, 0x1e,
0x83, 0x4e, 0xd1,
0xeb, 0xf9, 0x26, 0xc7, 0x57, 0xd8, 0xf1, 0x34, 0xe0, 0x4b, 0xc5, 0xd0, 0xd0, 0x27, 0x8d, 0x08,
0xf6, 0x7c, 0x8f, 0xfd, 0x71, 0xbb, 0xfa, 0xdf, 0x56, 0xed, 0xe0, 0xbb, 0x65, 0x5d, 0x39, 0x49,
0x85, 0x4f, 0x06, 0xeb, 0x39, 0xeb, 0x92, 0x6f, 0x9e, 0x90, 0xf2, 0x74, 0x8c, 0x8b, 0x16, 0x8f,
0x6c, 0xe1, 0xf6, 0xaa, 0xf8, 0x60, 0x2f, 0xd7, 0xfc, 0x8f, 0x4c, 0xea, 0x98, 0xb6, 0xb1, 0xfd,
0x28, 0x9f, 0x2a, 0xcc, 0x81, 0xd7, 0xbc, 0xae, 0x2d, 0x65, 0x94, 0xa7, 0xa1, 0xf6, 0xd3, 0x6c,
0xec, 0x04, 0x10, 0x58, 0x3d, 0xac, 0x08, 0xdd, 0x1c, 0xfe, 0xd7, 0x2b, 0xca, 0x78, 0xc2, 0x16,
0x55, 0x68, 0xc1, 0x76, 0x02, 0x93, 0xcd, 0x17, 0x7c, 0x0e, 0xb3, 0xe4, 0xad, 0xe4, 0xa2, 0xbe,
0xff, 0x87, 0x87, 0xc1, 0xfc, 0x35, 0xc6, 0x2b, 0xbf, 0x29, 0x1e, 0x09, 0x6a, 0xfe, 0x5d, 0xa2,
0x2a, 0x68, 0x56, 0xdb, 0xb9, 0x19, 0x1d, 0x0d, 0x6f, 0xca, 0xd6, 0xa0, 0xf4, 0xdb, 0x09, 0xb4,
0x77, 0x3e, 0xd8, 0x90, 0x40, 0x1e, 0x79, 0xd1, 0x09, 0xee, 0x7f, 0x7d, 0xfa, 0x2a, 0xd6, 0xc0,
0x4d, 0x3c, 0x79, 0x97, 0xcc, 0x01, 0x38, 0x08, 0x84, 0xac, 0x9e, 0x16, 0x8b, 0x66, 0xf0, 0x3e,
0x57, 0x91, 0x42, 0x4c, 0x28, 0xa3, 0x73, 0xcb, 0x18, 0x25, 0x75, 0x8e, 0x35, 0xdb, 0x91, 0x81,
0x20, 0x95, 0xad, 0x38, 0xa2, 0xe7, 0x00, 0xe4, 0x8e, 0xa1, 0x7a, 0xe3, 0xca, 0xc1, 0xc4, 0x84,
0x1d, 0xd1, 0x37, 0x91, 0x33, 0xfb, 0xff, 0xd4, 0x5e, 0xd5, 0x5e, 0x82, 0xe9, 0x5b, 0xfe, 0x82,
0x09, 0x79, 0x42, 0xfd, 0x4e, 0x4d, 0xdd, 0xeb, 0xa6, 0xb3, 0xf4, 0x81, 0x26, 0xd5, 0xad, 0x28}
};

if(Cy_IPC_Drv_SendMsgWord(CY_IPC_STRUCT, CY_SROM_DR_IPC_NOTIFY_STRUCT, (uint32_t)&rmaParam) ==
CY_IPC_DRV_SUCCESS)
     {
    uint32_t elapsedTime = 0;

    /* Wait for the IPC structure to be freed */
        while (elapsedTime < IPC_STATUS_WAIT_TIME_S)
```

14 Appendix F - Transition to RMA lifecycle stage

```
        {
            if (Cy_IPC_Drv_IsLockAcquired(CY_IPC_STRUCT) == false)
                break;

            /* Delay by 1s before checking again */
            Cy_SysLib_Delay(DELAY_IPC_STATUS_CHECK_MS);
            elapsedTime++;
        }

        /* The result of the SROM API call is returned to the opcode variable */
        if ((rmaParam.opcode & CY_OPCODE_STS_Msk) == CY_OPCODE_SUCCESS)
        {
            printf("\r\nTransition to RMA successful!!!");
        }
        else
        {
            printf("\r\nTransition to RMA Failed!!! ... ERR_STATUS: 0x%lx\r\n",
rmaParam.opcode);
        }

    }
    else
         printf("\r\nTransition to RMA Failed!!! ... ERR_STATUS: 0x%lx\r\n", rmaParam.opcode);
```

In the above code snippet, the Unique ID and the signature have to be replaced with the actual silicon being used.

The **transit_rma_param_t** structure mentioned in the above code snippet is in RMA certificate format.

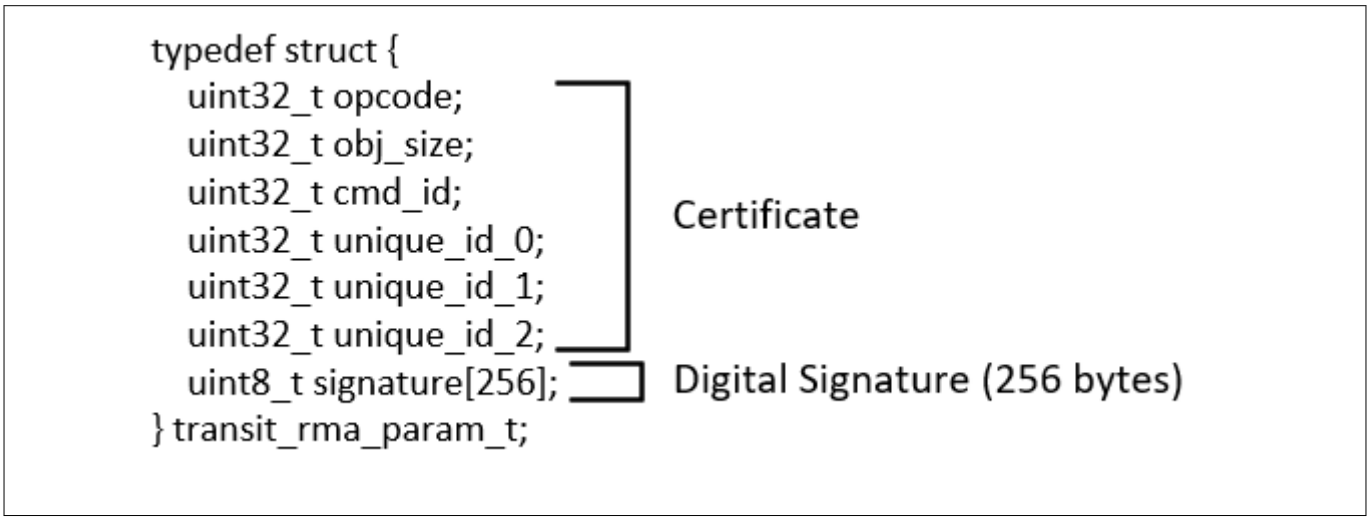


Figure 10 RMA certificate structure

***Note**: The key used to generate the certificate and signing the image for the secured boot should be the same. If the key is different, RMA fails.*

***Note**: The device should be in a SECURE or SECURE_WITH_DEBUG lifecycle stage before it is moved to RMA.*

14 Appendix F - Transition to RMA lifecycle stage

***Note**: The next section mentions how to create a certificate. The signature in the certificate is in big endian format when generated on Linux machines. It has to be converted to little endian format while copying to RMA certificate structure.*

Once the above code is executed, the device will immediately switch to RMA mode.

14.1 Generate certificate

This section describes how to generate a certificate in Linux using OpenSSL.

1. Use the `ReadUniqueID` API (System call 0x1F) to read the Unique ID in the SFlash as mentioned in Appendix D - Read device Unique ID

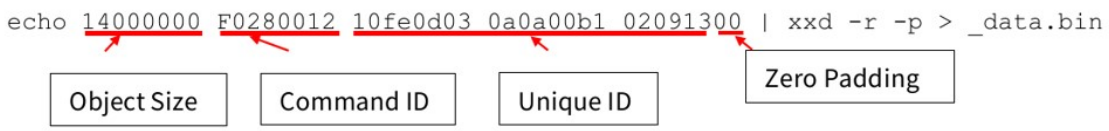
   The following is an example of the read result of readUniqueID.

   0xa00dfe10: First byte is the system call status (0xa0: indicating success/0xf0: indicating fail) Others: **Unique ID_0**

   0x000a0a03: **Unique ID_1**

   0x130902b1: **Unique ID_2**

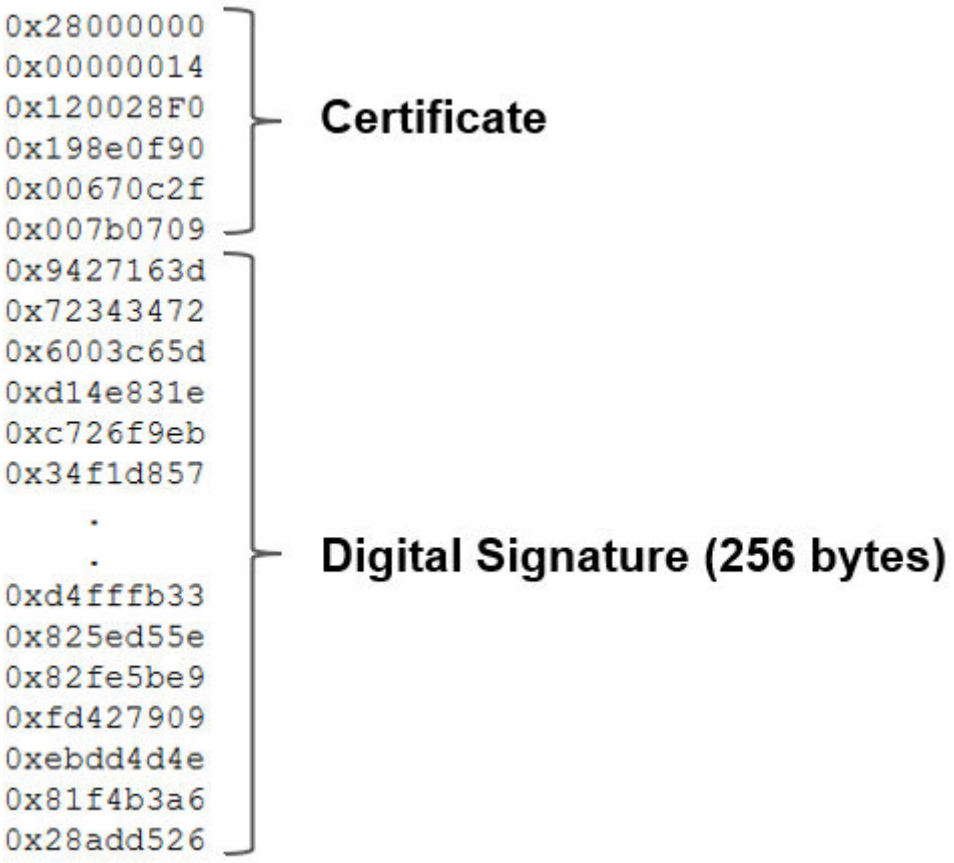
2. Enter the following commands from the Linux environment along with appropriate device UniqueID. The following is an example of transition to RMA certificate and digital signature generation




```
openssl dgst -sha256 -sign rsa_private.txt _data.bin > _signature.bin

cat _signature.bin | xxd -p -c 64 > _tmp1.hex

echo 00000028 > _tmp2.hex

echo 14000000 F0280012 10fe0d03 0a0a00b1 02091300 >> _tmp2.hex

cat _tmp1.hex >> _tmp2.hex

sed -r "s/\s*(\w{2})(\w{2})(\w{2})(\w{2})/0x\4\3\2\1\n/g" _tmp2.hex | sed -r
"/^$/d" > _output_transtorma.hex

rm _data.bin _tmp1.hex _tmp2.hex
```

The generated *_output_transtorma.hex* file contains the certificate along with the digital signature as shown below:

14 Appendix F - Transition to RMA lifecycle stage



15 Appendix G - Configure application protection

This section describes how you can configure and add SWPU, which is application protection for the customer system. SWPU consists of FWPU, ERPU, and EWPU. Application protection can be configured with up to 16 entries for FWPU and up to 4 entries for ERPU and EWPU.

Each SWPU has a master/slave structure, that is, it protects the protection structure by protection structure. The slave attribute indicates the access attribute to resources and the master protection attribute indicates access attribute to the slave. Therefore, changes in the SWPU slave attribute requires master-defined attributes.

Application protection consists of the following elements. See the “Protection unit” chapter in the Architecture TRM [2] for more details.

- `PU_OBJECT_SIZE`: Number of configured elements (4 bytes)
- `N_FWPU`: Number of FWPU objects. FWPU has up to 16 regions (4 bytes)
- `FWPUx_SL_ADDR`: Configures the FWPUx base address (4 bytes)
- `FWPUx_SL_SIZE`: Configures the FWPUx region size and FWPUx enable (4 bytes)
- `FWPUx_SL_ATT`: Configures the FWPUx slave attribute (4 bytes)
- `FWPUx_MS_ATT`: Configures the FWPUx master attribute (4 bytes)
- `N_ERPU`: Number of ERPU objects. ERPU has up to 4 regions (4 bytes)
- `ERPUy_SL_OFFSET`: Configures the ERPUy base address offset (4 bytes)
- `ERPUy_SL_SIZE`: Configures the ERPUy region size and ERPUy enable (4 bytes)
- `ERPUy_SL_ATT`: Configures the ERPUy slave attribute (4 bytes)
- `ERPUy_MS_ATT`: Configures the ERPUy master attribute (4 bytes)
- `N_EWPU`: Number of EWPU objects. EWPU has up to 4 regions (4 bytes)
- `EWPUy_SL_OFFSET`: Configures the EWPUy base address offset (4 bytes)
- `EWPUy_SL_SIZE`: Configures the EWPUy region size and ERPUy enable (4 bytes)
- `EWPUy_SL_ATT`: Configures the EWPUy slave attribute (4 bytes)
- `EWPUy_MS_ATT`: Configures the EWPUy master attribute (4 bytes)

Suffix “x” indicates 0 to 15, and suffix “y” indicates 0 to 3.

15.1 Configuration

The following shows the steps to configure application protection. The application protection address must match the "Address of Application Protection (offset = 0x108)" in TOC2. To ensure security, do not change the default value (0x17007600).

15 Appendix G - Configure application protection

1. Set the number of regions for each SWPU

```
// deification of application protection
#define N_FWPU (1UL) /**< Number of flash write protection Max 16 */
#define N_ERPU (1UL) /**< Number of efuse read protection Max 4 */
#define N_EWPU (1UL) /**< Number of efuse write protection Max 4 */
```

2. Configure each SWPU according to the number of SWPUs

```
   /*************************************
   *   Application Protection
   *************************************/
   CY_SECTION(".cy_sflash_app_prot") __USED static const cy_stc_si_app_prot_t
cy_si_appprot =
   {
    .objSize                = OBJECT_SIZE,            /* Application Protection Object
Size (in bytes) */
    .n_fwpu                 = N_FWPU,                 /* Number of FWPU Max 16 */
    .fwpu0_adr.addr30       = 0x10000000>>2ul,        /* Add region if you need */
    .fwpu0_size.region_size = 0x200,                  /* in bytes (multiple of 4) */
    .fwpu0_size.enable      = APP_PROT_ENABLE,        /* FWPU0 enable */
    .fwpu0_sl_att.urw       = APP_PROT_ALLOW,         /* FWPU0 Slave Attribute */
    .fwpu0_sl_att.prw       = APP_PROT_ALLOW,         /* FWPU0 Slave Attribute */
    .fwpu0_sl_att.ns        = APP_PROT_ALLOW,         /* FWPU0 Slave Attribute */
    .fwpu0_sl_att.pc_mask   = 0x00FF,                 /* FWPU0 Slave Attribute */
    .fwpu0_ms_att.urw       = APP_PROT_ALLOW,         /* FWPU0 Master Attribute */
    .fwpu0_ms_att.prw       = APP_PROT_ALLOW,         /* FWPU0 Master Attribute */
    .fwpu0_ms_att.ns        = APP_PROT_ALLOW,         /* FWPU0 Master Attribute */
    .fwpu0_ms_att.pc_mask   = 0x00FF,                 /* FWPU0 Master Attribute */
    .n_erpu                 = N_ERPU,                 /* Number of ERPU Max 4 */
    .erpu0_offset.offset    = 0x68,                   /* ERPU0 offset (Default) */
    .erpu0_size.region_size = 0x18,                   /* ERPU0 region size (Default) */
    .erpu0_size.enable      = APP_PROT_ENABLE,        /* ERPU0 enable (Default) */
    .erpu0_sl_att.urw       = APP_PROT_ALLOW,         /* ERPU0 Slave Attribute (Default)
*/
    .erpu0_sl_att.prw       = APP_PROT_ALLOW,         /* ERPU0 Slave Attribute (Default)
*/
    .erpu0_sl_att.ns        = APP_PROT_ALLOW,         /* ERPU0 Slave Attribute (Default)
*/
    .erpu0_sl_att.pc_mask   = 0x00FF,                 /* ERPU0 Slave Attribute (Default)
*/
    .erpu0_ms_att.urw       = APP_PROT_ALLOW,         /* ERPU0 Master Attribute
(Default) */
    .erpu0_ms_att.prw       = APP_PROT_ALLOW,         /* ERPU0 Master Attribute
(Default) */
    .erpu0_ms_att.ns        = APP_PROT_ALLOW,         /* ERPU0 Master Attribute
(Default) */
    .erpu0_ms_att.pc_mask   = 0x00FF,                 /* ERPU0 Master Attribute
(Default) */
    .n_ewpu                 = N_EWPU,                 /* Number of EWPU Max 4 */
    .ewpu0_offset.offset    = 0x68,                   /* EWPU0 offset (Default) */
    .ewpu0_size.region_size = 0x18,                   /* EWPU0 region size (Default) */
```

15 Appendix G - Configure application protection

```
    .ewpu0_size.enable      = APP_PROT_ENABLE,          /* EWPU0 enable (Default) */
    .ewpu0_sl_att.urw       = APP_PROT_ALLOW,           /* EWPU0 Slave Attribute (Default)
*/
    .ewpu0_sl_att.prw       = APP_PROT_ALLOW,           /* EWPU0 Slave Attribute (Default)
*/
    .ewpu0_sl_att.ns        = APP_PROT_ALLOW,           /* EWPU0 Slave Attribute (Default)
*/
    .ewpu0_sl_att.pc_mask   = 0x00FF,                   /* EWPU0 Slave Attribute (Default)
*/
    .ewpu0_ms_att.urw       = APP_PROT_ALLOW,           /* EWPU0 Master Attribute
(Default) */
    .ewpu0_ms_att.prw       = APP_PROT_ALLOW,           /* EWPU0 Master Attribute
(Default) */
    .ewpu0_ms_att.ns        = APP_PROT_ALLOW,           /* EWPU0 Master Attribute
(Default) */
    .ewpu0_ms_att.pc_mask   = 0x00FF,                   /* EWPU0 Master Attribute
(Default) */
  }
```

Table 10 shows the details for each element.

Table 10 Elements descriptions

Element	Description
.objSize	Number of configured elements
.n_fwpu	Number of FWPUs
.n_erpu	Number of ERPUs
.n_ewpu	Number of EWPUs
.fwpu0_adr.addr30	Base address of FWPU0. 4-byte aligned.
.fwpu0_size.region_size	FWPU0 region size
.fwpu0_size.enable	FWPU0 enable: 0: Disable 1: Enable
.fwpu0/erpu0/ewpu0_sl_att.urw	FWPU0/EWPU0 slave attribute for user write restriction: 0: Prohibit 1: Allow EWPU0 slave attribute for user read restriction: 0: Prohibit 1: Allow
.fwpu0/erpu0/ewpu0_sl_att.prw	FWPU0/EWPU0 slave attribute for privileged write restriction: 0: Prohibit 1: Allow EWPU0 slave attribute for privileged read restriction: 0: Prohibit 1: Allow

(table continues...)

Table 10 (continued) Elements descriptions

Element	Description
.fwpu0/erpu0/ewpu0_sl_att.ns	FWPU0/EWPU0 slave attribute for non-secure write restriction: 0: Prohibit 1: Allow EWPU0 slave attribute for non-secure read restriction: 0: Prohibit 1: Allow
.fwpu0/erpu0/ ewpu0_sl_att.pc_mask	FWPU0/ERPU0/EWPU0 slave attribute for PC restriction. The PC number corresponds to the bit number. 0: Prohibit 1: Allow (When pc_mask is 0x0005, PC0 and PC2 are allowed)
.fwpu0_ms_att.urw	FWPU0/EWPU0 master attribute for user write restriction. 0: Prohibit 1: Allow EWPU0 master attribute for user read restriction. 0: Prohibit 1: Allow
.fwpu0_ms_att.prw	FWPU0/EWPU0 master attribute for privileged write restriction. 0: Prohibit 1: Allow EWPU0 master attribute for privileged read restriction. 0: Prohibit 1: Allow
.fwpu0_ms_att.ns	FWPU0/EWPU0 master attribute for non-secure write restriction. 0: Prohibit 1: Allow EWPU0 master attribute for non-secure read restriction. 0: Prohibit 1: Allow
.fwpu0_ms_att.pc_mask	FWPU0/ERPU0/EWPU0 master attribute for PC restriction. The PC number corresponds to the bit number. 0: Prohibit 1: Allow (When pc_mask is 0x0005, PC0 and PC2 are allowed)
.erpu0/ewpu0_offset.offset	ERPU0 base address offset
.erpu0/ewpu0_size.region_size	ERPU0 region size
.erpu0/ewpu0_size.enable	ERPU0 enable. 0: Disable 1: Enable

16 Appendix H - Normal access restriction

The following example provides a configuration of the normal access restriction (NAR). NAR determines DAP restrictions for NORMAL_PROVISIONED and SECURE_W_DEBUG. NAR is deployed in flash booting. See the "BootROM" chapter in Architecture TRM [2].

The example can configure the normal access restrictions and Normal Dead access restrictions for M0+ DAP, M7/M4 DAP, and System DAP.

16.1 Configuration

The following code shows the steps to configure normal access restriction. This section is located at 0x17001A00 in the SFlash.

```
#define CY_SI_CM0_ENABLE                (0UL)  /**< CM0 ACCESS PORT ENABLE */
#define CY_SI_CM0_DISABLE_TMP           (1UL)  /**< CM0 ACCESS PORT TEMPORARY DISABLE */
#define CY_SI_CM0_DISABLE               (2UL)  /**< CM0 ACCESS PORT PERMANENTLY_DISABLE */
#define CY_SI_CMx_ENABLE                (0UL)  /**< CM7/CM4 ACCESS PORT ENABLE */
#define CY_SI_CMx_DISABLE_TMP           (1UL)  /**< CM7/CM4 ACCESS PORT TEMPORARY DISABLE */
#define CY_SI_CMx_DISABLE               (2UL)  /**< CM7/CM4 ACCESS PORT PERMANENTLY_DISABLE */
#define CY_SI_SYS_ENABLE                (0UL)  /**< SYS ACCESS PORT ENABLE */
#define CY_SI_SYS_DISABLE_TMP           (1UL)  /**< SYS ACCESS PORT TEMPORARY DISABLE */
#define CY_SI_SYS_DISABLE               (2UL)  /**< SYS ACCESS PORT PERMANENTLY_DISABLE */

/* Access Restriction */
#define CY_SI_NAR_NORMALACCESSRESTRICTION ((CY_SI_CM0_ENABLE << CY_SI_CM0_AP_POS) \
                                        | (CY_SI_CMx_ENABLE << CY_SI_CMx_AP_POS) \
                                        | (CY_SI_SYS_ENABLE << CY_SI_SYS_AP_POS)) \
                                        | 0x80                              /* Fixed value */

#define CY_SI_NAR_NORMALDEADACCESSRESTRICTION ((CY_SI_CM0_ENABLE << CY_SI_CM0_AP_POS) \
                                        | (CY_SI_CMx_ENABLE << CY_SI_CMx_AP_POS) \
                                        | (CY_SI_SYS_ENABLE << CY_SI_SYS_AP_POS))

CY_SECTION(".cy_sflash_nar") __USED static const cy_stc_si_nar_t cy_nar =
{
    .nar         = CY_SI_NAR_NORMALACCESSRESTRICTION,       /* Normal Access Restrictions */
    .ndar        = CY_SI_NAR_NORMALDEADACCESSRESTRICTION,   /* Normal Dead Access Restrictions
*/
};
```

Table 11 shows the details of each element.

Table 11 NAR setting

Element	Description
CY_SI_CM0/CMx/SYS_ENABLE	Set to Enable M0, M7/M4, or SYS-DAP
CY_SI_CM0/CMx/ SYS_DISABLE_TMP	Set to Disable M0, M7/M4, or SYS-DAP
CY_SI_CM0/CMx/SYS_DISABLE	Set to Permanently Disable M0, M7/M4, or SYS-DAP

See Table 5 for setting details.

References

The following are the XMC7000 family series datasheets and technical reference manuals. Contact Technical support to obtain these documents, code snippets, Peripheral Driver Library (PDL), and tools used in this application note. Note that the obtained sample program does not conform to industrial standards, so it is used only as an example for secure system configuration and cannot be used for production purposes.

1. Device datasheet
   - XMC7200 datasheet 32-bit Arm® Cortex®-M7 microcontroller XMC7000 family
   - XMC7100 datasheet 32-bit Arm® Cortex®-M7 microcontroller XMC7000 family
   - XMC5100 datasheet 32-bit Arm® Cortex®-M4F microcontroller XMC5000 family
   - XMC5200 datasheet 32-bit Arm® Cortex®-M4F microcontroller XMC5000 family
   - XMC5300 datasheet 32-bit Arm® Cortex®-M4F microcontroller XMC5000 family
2. Reference manual
   - XMC7000 MCU family architecture reference manual
   - XMC7200 MCU registers reference manual
   - XMC7100 MCU registers reference manual
   - XMC5000 MCU architecture technical reference manual
   - XMC5100/5200 MCU registers reference manual
   - XMC5300 MCU registers reference manual
3. Application notes
   - Getting started with XMC7000 MCU on ModusToolbox™ software
   - Getting started with XMC5000 MCU on ModusToolbox™ software

Revision history

Document revision	Date	Description of changes
**	2022-05-26	Initial release
*A	2024-08-23	Added note in What is a secure system? Added a note in RMA for Open RMA behavior in the SECURE_WITH_DEBUG lifecycle stage. Added a note in Code signing for boot authentication of application software. Added protection condition of Normal Dead Access Restriction in Table 3, and Note. Updated references and links.
*B	2024-09-27	Template update Added Appendix D - Read device Unique ID Added Appendix E - Transition to SECURE/SECURE_WITH_DEBUG lifecycle stage Updated Appendix F - Transition to RMA lifecycle stage Updated Generate certificate
*C	2026-08-18	Updated to support XMC5000 family Added edge protect tools in Appendix A - Example of creating public and private keys

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2026-08-18

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2026 Infineon Technologies AG

All Rights Reserved.

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-hzl1721980135051

Important notice

Products which may also include samples and may be comprised of hardware or software or both ("Product(s)") are sold or provided and delivered by Infineon Technologies AG and its affiliates ("Infineon") subject to the terms and conditions of the frame supply contract or other written agreement(s) executed by a customer and Infineon or, in the absence of the foregoing, the applicable Sales Conditions of Infineon. General terms and conditions of a customer or deviations from applicable Sales Conditions of Infineon shall only be binding for Infineon if and to the extent Infineon has given its express written consent.

For the avoidance of doubt, Infineon disclaims all warranties of non-infringement of third-party rights and implied warranties such as warranties of fitness for a specific use/purpose or merchantability. Infineon shall not be responsible for any information with respect to samples, the application or customer's specific use of any Product or for any examples or typical values given in this document.

If this document is marked as 'Preliminary', 'Target', 'Draft' or 'Proposal', Infineon reserves the right to change all information given in this document at any time without notice. Subject to the development and release of a Product for series supply by Infineon, the technical specifications of the Product are set forth in the relevant datasheet provided by Infineon without such marking.

The data contained in this document is exclusively intended for technically qualified and skilled customer representatives. It is the responsibility of the customer to evaluate the suitability of the Product for the intended application and the customer's specific use and to verify all relevant technical data contained in this document in the intended application and the customer's specific use. The customer is responsible for properly designing, programming, and testing the functionality and safety of the intended application, as well as complying with any legal requirements related to its use.

Unless otherwise explicitly approved by Infineon, Products may not be used in any application where a failure of the Products or any consequences of the use thereof can reasonably be expected to result in personal injury.

However, the foregoing shall not prevent the customer from using any Product in such fields of use that Infineon has explicitly designed and sold it for, provided that the overall responsibility for the application lies with the customer. Infineon expressly reserves the right to use its content for commercial text and data mining (TDM) according to applicable laws, e.g. Section 44b of the German Copyright Act (UrhG). If the Product includes security features:

Because no computing device can be absolutely secure, and despite security measures implemented in the Product, Infineon does not guarantee that the Product will be free from intrusion, data theft or loss, or other breaches ("Security Breaches"), and Infineon shall have no liability arising out of any Security Breaches.

If this document includes or references software:

The software is owned by Infineon under the intellectual property laws and treaties of the United States, Germany, and other countries worldwide. All rights reserved. Therefore, you may use the software only as provided in the software license agreement accompanying the software.

If no software license agreement applies, Infineon hereby grants you a personal, non-exclusive, non-transferable license (without the right to sublicense) under its intellectual property rights in the software (a) for software provided in source code form, to modify and reproduce the software solely for use with Infineon hardware products, only internally within your organization, and (b) to distribute the software in binary code form externally to end users, solely for use on Infineon hardware products. Any other use, reproduction, modification, translation, or compilation of the software is prohibited.

For further information on the Product, technology, delivery terms and conditions, and prices, please contact your nearest Infineon office or visit www.infineon.com.