NXP Semiconductors

Reference Manual

Document identifier: KW47SRM

Rev. 3, 2025-12-02

KW47 Security Reference Manual

Contents

Figures

Tables

Chapter 1
About this Manual

1.1 Audience

This reference manual is intended for system software and hardware developers and applications programmers who want to develop products with this device. It assumes that the reader understands operating systems, microprocessor system design, and basic principles of software and hardware.

1.2 Organization

This manual begins with a global introduction of the chip, followed by chapters organized into *functional groups* that detail particular areas of functionality, such as system control, clocking, and timers. Each functional group can have two main types of chapters:

- *System-level* chapters contain information that applies to the components (modules) within the group.
- *Module-level* chapters contain technical descriptions of individual modules within the group.

Note that application-specific groups (such as timers) may only contain module-level chapters.

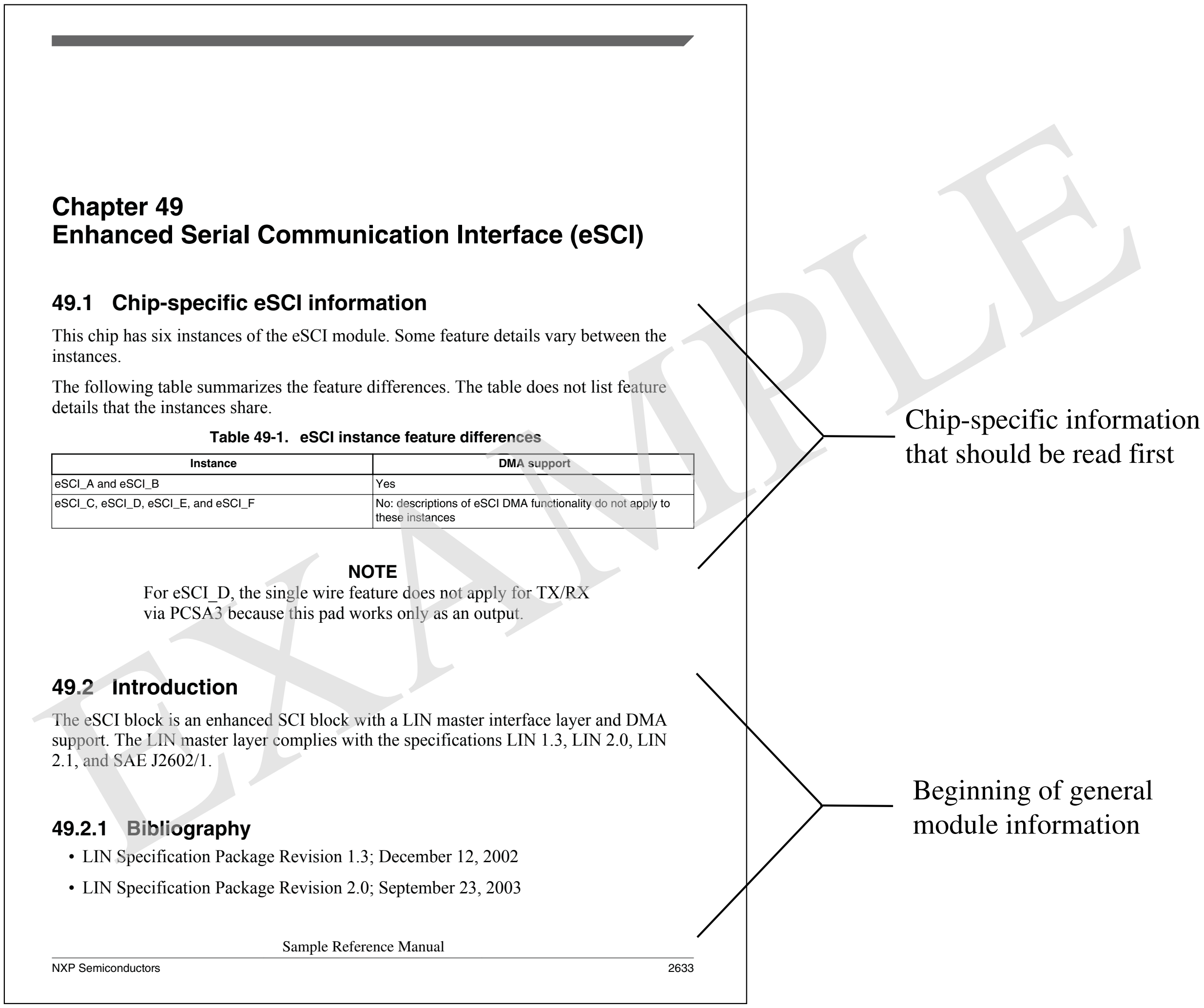
1.3 Module descriptions

Each module chapter has two main parts:

- **Chip-specific:** The first section, *Chip-specific [module name] information*, includes the number of module instances on the chip and possible implementation differences between the module instances, such as differences in FIFO depths or the number of channels supported. It may also include functional connections between the module instances and other modules. Read this section *first* because its content is crucial to understanding the information in other sections of the chapter.
- **General:** The subsequent sections provide general information about the module, including its signals, registers, and functional description.

NOTE

If there is a conflict between the chip-specific module information (first section) and the general module information (subsequent sections), the chip-specific information supersedes the general information.



Chapter 49
Enhanced Serial Communication Interface (eSCI)

49.1 Chip-specific eSCI information

This chip has six instances of the eSCI module. Some feature details vary between the instances.

The following table summarizes the feature differences. The table does not list feature details that the instances share.

Table 49-1. eSCI instance feature differences

Instance	DMA support
eSCI_A and eSCI_B	Yes
eSCI_C, eSCI_D, eSCI_E, and eSCI_F	No: descriptions of eSCI DMA functionality do not apply to these instances

NOTE

For eSCI_D, the single wire feature does not apply for TX/RX via PCSA3 because this pad works only as an output.

49.2 Introduction

The eSCI block is an enhanced SCI block with a LIN master interface layer and DMA support. The LIN master layer complies with the specifications LIN 1.3, LIN 2.0, LIN 2.1, and SAE J2602/1.

49.2.1 Bibliography

- LIN Specification Package Revision 1.3; December 12, 2002
- LIN Specification Package Revision 2.0; September 23, 2003

Sample Reference Manual

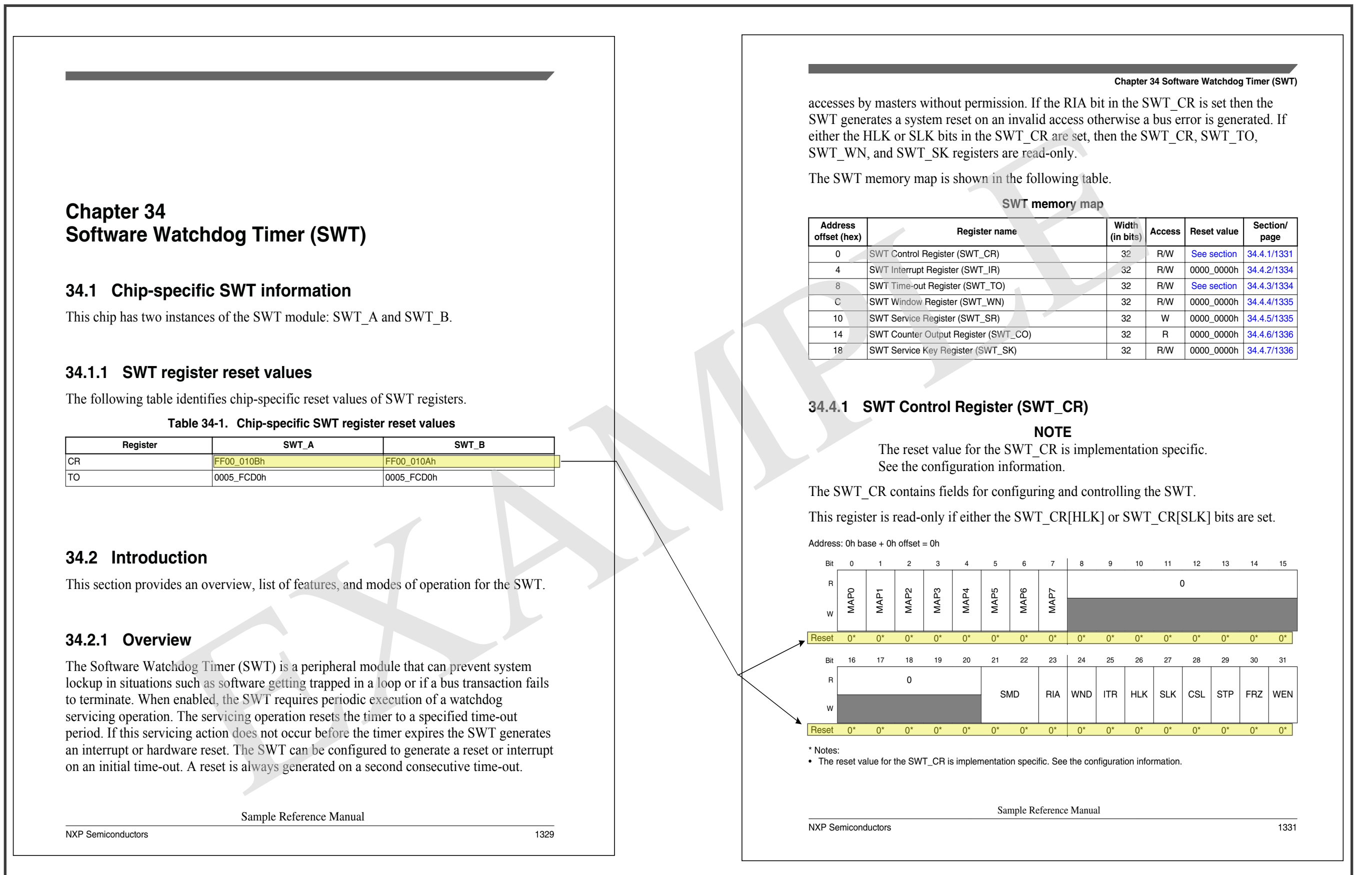
NXP Semiconductors 2633

EXAMPLE
Chip-specific information that should be read first
Beginning of general module information

Figure 1. Example: chapter chip-specific information and general module information

1.3.1 Example: chip-specific information that supersedes content in the same chapter

The example below shows chip-specific information that supersedes general module information presented later in the chapter. In this case, the chip-specific register reset values supersede the reset values that appear in the register diagram.



Chapter 34
Software Watchdog Timer (SWT)

34.1 Chip-specific SWT information

This chip has two instances of the SWT module: SWT_A and SWT_B.

34.1.1 SWT register reset values

The following table identifies chip-specific reset values of SWT registers.

Table 34-1. Chip-specific SWT register reset values

Register	SWT_A	SWT_B
CR	FF00_010Bh	FF00_010Ah
TO	0005_FCD0h	0005_FCD0h

34.2 Introduction

This section provides an overview, list of features, and modes of operation for the SWT.

34.2.1 Overview

The Software Watchdog Timer (SWT) is a peripheral module that can prevent system lockup in situations such as software getting trapped in a loop or if a bus transaction fails to terminate. When enabled, the SWT requires periodic execution of a watchdog servicing operation. The servicing operation resets the timer to a specified time-out period. If this servicing action does not occur before the timer expires the SWT generates an interrupt or hardware reset. The SWT can be configured to generate a reset or interrupt on an initial time-out. A reset is always generated on a second consecutive time-out.

Sample Reference Manual

NXP Semiconductors 1329

Chapter 34 Software Watchdog Timer (SWT)

accesses by masters without permission. If the RIA bit in the SWT_CR is set then the SWT generates a system reset on an invalid access otherwise a bus error is generated. If either the HLK or SLK bits in the SWT_CR are set, then the SWT_CR, SWT_TO, SWT_WN, and SWT_SK registers are read-only.

The SWT memory map is shown in the following table.

SWT memory map

Address offset (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
0	SWT Control Register (SWT_CR)	32	R/W	See section	34.4.1/1331
4	SWT Interrupt Register (SWT_IR)	32	R/W	0000_0000h	34.4.2/1334
8	SWT Time-out Register (SWT_TO)	32	R/W	See section	34.4.3/1334
C	SWT Window Register (SWT_WN)	32	R/W	0000_0000h	34.4.4/1335
10	SWT Service Register (SWT_SR)	32	W	0000_0000h	34.4.5/1335
14	SWT Counter Output Register (SWT_CO)	32	R	0000_0000h	34.4.6/1336
18	SWT Service Key Register (SWT_SK)	32	R/W	0000_0000h	34.4.7/1336

34.4.1 SWT Control Register (SWT_CR)

NOTE

The reset value for the SWT_CR is implementation specific.
See the configuration information.

The SWT_CR contains fields for configuring and controlling the SWT.

This register is read-only if either the SWT_CR[HLK] or SWT_CR[SLK] bits are set.

Address: 0h base + 0h offset = 0h

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	MAP0	MAP1	MAP2	MAP3	MAP4	MAP5	MAP6	MAP7	0							
W																
Reset	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*

Bit	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0					SMD		RIA	WND	ITR	HLK	SLK	CSL	STP	FRZ	WEN
W																
Reset	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*	0*

* Notes:
- The reset value for the SWT_CR is implementation specific. See the configuration information.

Sample Reference Manual

NXP Semiconductors 1331

Figure 2. Example: chip-specific information that supersedes content in the same chapter

1.4 Register descriptions

Module chapters present register information in:

- Memory maps including:
  - — Addresses
  - — The name and acronym/abbreviation of each register
  - — The width of each register (in bits)
  - — Each register's reset value
  - — The page number on which each register is described
- Register figures
- Field-description tables
- Associated text

The register figures show the field structure using the conventions in the following figure.

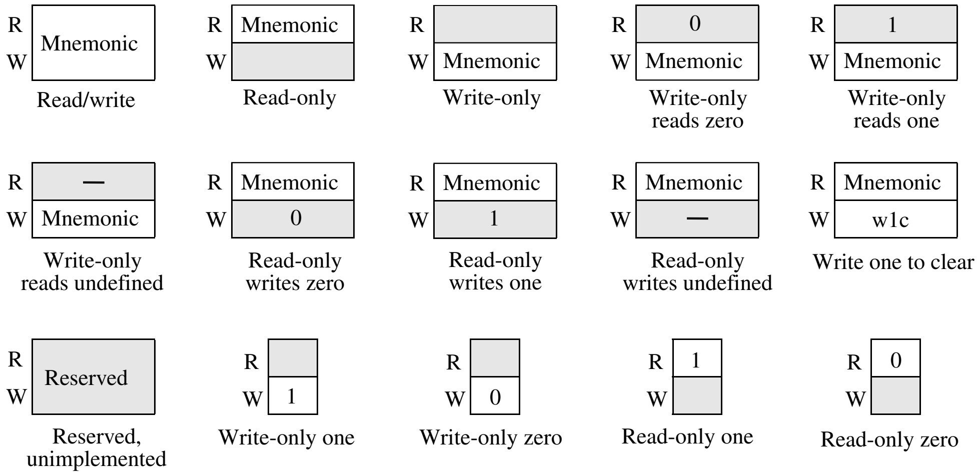


Figure 3. Register figure conventions

1.5 Conventions

1.5.1 Numbering systems

The following suffixes identify different numbering systems:

This suffix	Identifies a
b	Binary number. For example, the binary equivalent of the number 5 is written 101b. In some cases, binary numbers are shown with the prefix *0b*.
d	Decimal number. Decimal numbers are followed by this suffix only when the possibility of confusion exists. In general, decimal numbers are shown without a suffix.
h	Hexadecimal number. For example, the hexadecimal equivalent of the number 60 is written 3Ch. In some cases, hexadecimal numbers are shown with the prefix *0x*.

1.5.2 Typographic notation

The following typographic notation is used throughout this document:

Example	Description
placeholder, x	Items in italics are placeholders for information that you provide. Italicized text is also used for the titles of publications and for emphasis. Plain lowercase letters are also used as placeholders for single letters and numbers.
`code`	Fixed-width type indicates text that must be typed exactly as shown. It is used for instruction mnemonics, directives, symbols, subcommands, parameters, and operators. Fixed-width type is also used for example code. Instruction mnemonics and directives in text and tables are shown in all caps; for example, BSR.
SR[SCM]	A mnemonic in brackets represents a named field in a register. This example refers to the Scaling Mode (SCM) field in the Status Register (SR).
REVNO[6:4], XAD[7:0]	Numbers in brackets and separated by a colon represent either:

Table continues on the next page...

Table continued from the previous page...

Example	Description
	• A subset of a register's named field For example, REVNO[6:4] refers to bits 6–4 that are part of the COREREV field that occupies bits 6–0 of the REVNO register. • A continuous range of individual signals of a bus For example, XAD[7:0] refers to signals 7–0 of the XAD bus.

1.5.3 Special terms

The following terms have special meanings:

Term	Meaning
asserted	Refers to the state of a signal as follows: • An active-high signal is asserted when high (1). • An active-low signal is asserted when low (0).
deasserted	Refers to the state of a signal as follows: • An active-high signal is deasserted when low (0). • An active-low signal is deasserted when high (1). In some cases, deasserted signals are described as *negated*.
reserved	Refers to a memory space, register, field, or programming setting. Writes to a reserved location can result in unpredictable functionality or behavior. • Do not modify the default value of a reserved programming setting, such as the reset value of a reserved register field. • Consider undefined locations in memory to be reserved.
w1c	Write 1 to clear: Refers to a register bitfield that must be written as 1 to be "cleared."

Chapter 2 Security Overview

2.1 Disclaimer

As system security requirements and the attack surface evolves, it is important for customers to understand the types of attacks (especially advanced physical attacks) which NXP does not claim to protect against, or strongly mitigate, so that appropriate mitigation can be taken by the customer at the system level if necessary.

- This SoC has built-in tamper detection features which detect basic tamper events. However, NXP does not guarantee against advanced tamper attempts, including operation of the device beyond the defined specification limits. NXP does not guarantee the protection against semi-invasive and invasive attacks.
- This SoC has several built-in features addressing side channel attacks, e.g. mechanisms to make Differential Power Analysis (DPA) more difficult. However, there is no claim to be completely resistant. The effectiveness of these features has not been independently evaluated. Therefore NXP does not guarantee that the result will meet specific customer requirements.
- This SoC's security trust architecture relies on the strength of cryptographic algorithms and digital signatures. If these are subsequently determined to have inherent flaws, then the impact for each flaw must be evaluated and, in this case, NXP does not guarantee the underlying trust architecture claims.
- This SoC has some built-in access control mechanisms to support the logical separation of executed code. However, NXP does not guarantee that the device completely ensures logical separation by itself. Any vulnerabilities identified in Trusted Execution Environments or Hypervisor software may impact this separation and data integrity, and may require additional mitigations.

NXP recommends customers to implement appropriate design and operating safeguards based on defined threat models, to minimize the security risks associated with their applications and products.

2.2 Security features

By controlling access to the device as well as providing trusted computing environment, this chip offers various security features and tools that allow customers to protect their assets.

This chapter explains the main purpose and features of the following security-relevant SoC components:

- Secure Isolation components:
  - Arm’s TrustZone for armv8-M architecture included as part Cortex-M33
  - NXP’s Trusted Domain Resource Controller (TRDC)
- Cryptographic Accelerators:
  - EdgeLock Secure Enclave (ELE)
- Key Management:
  - Part of ELE
  - UDF to create device unique root of trust key
- OTP to store security configuration
- Anomaly Detection and Response sub-system:
  - Glitch detector for VDD Core
  - Low-Voltage (LVD) and High-voltage (HVD) Detectors
  - Windowed Watch Dog Timers (WDOG)
  - Passive tamper pins monitored in RTC domain (Always-on domain)
- Boot ROM supporting following security features:

- Secure boot providing ECDSA signature verification as per NIST P-256 and P-384 curves
- Secure update using SB3.1 file format. Supports firmware update mechanism with authenticity (EC P-384 signed) and confidentiality (AES-CMAC encrypted and SHA384 chained) protection
- Secure debug: certificate-based debug authentication mechanism using ECDSA P-256 and P-384 keys
- Secure provisioning
- Secure identity solutions

- Key Storage:
  - ELE Key store generated from UDF based device unique key

See the Reference Manual for more details.

Table 1. Cryptographic operations

Algorithm	Key lengths	Modes
AES	128, 192, 256	CBC, ECB, CTR, CCM, GCM
ECDH	192, 224, 256, 384, 521	NIST P-192 , P-224, P-256, P-384, and P-521 curves, BrainPool curves
MontDH	255	Curve25519
ECDH(E)	-	NIST P-192, P-224, P-256, P-384, and P-521 curves
SPAKE2+	-	-
ECDSA	192, 224, 256, 384, 521	NIST P-192, P-224, P-256, P-384, and P-521 curves, BrainPool curves
Ed25519	255	IETF RFC 8032
SHA-2	-	SHA-224, SHA-256, SHA-384, SHA-512
SHA-3	-	-
HMAC	-	SHA-224, SHA-256, SHA-384, SHA-512
CMAC	128, 256	AES
CKDF	128, 256	-
HKDF	-	RFC 5869 with SHA-1 or SHA-2

2.2.1 Immutable Root of Trust (RoT)

As defined by Trusted Computing Group, “an Immutable Root of Trust (RoT) is expected to remain identical across all devices within a set of device models based on a defined threat model. It is also expected not to change across time and, therefore, will behave the same during each device’s lifespan.”

Immutable RoT consists of truly immutable hardware logic, including analog and digital logic, read-only memory and one-time programmable memory. Immutable RoT is essential for guaranteeing any security feature,including secure boot, secure debug, lifecycle management, and a number of others. In this device, Immutable RoT is embedded in the ELE Boot ROM (immutable bootloader), and it relies on the ELE hadrware cryptographic accelerators for its cunction.

2.2.2 Lifecycle management

During its life, a device goes through several life-cycle states. Some of the life-cycle states are there to ease the testing and development when the device is physically present in different environments, e.g. manufacturing fab, silicon manufacturer's test floor, silicon manufacturer's inventory, OEM's contract manufacturer facilities, OEM facilities, in field, etc. While easing the testing and development, it is important to protect security assets available on a device in a given life-cycle state. For each life-cycle state access rights are defined, i.e. what kind of access to the device internals is allowed and under what conditions. Available assets and implemented asset protections are also defined. Transition between different life-cycle states is an irreversible process.

2.2.3 Secure boot

Secure Boot ensures authenticity, integrity and confidentiality of the device bootloader, firmware, and other software during the boot process and ensures that the intended secure life-cycle state is reached. ECDSA P-256 with SHA-256 or ECDSA P-384 with SHA-384 are there to guarantee authenticity and integrity of the firmware image. Immutable RoT is in charge of enforcing Secure Boot and it does so according to the policies defined by the life-cycle state.

2.2.4 Secure update

Secure update is the process used to securely update the firmware image in the field. A firmware update is due when the running firmware gets compromised or a new feature is about to be added. Secure update guarantees authenticity and confidentiality of the new image.

On this chip, the ROM supports secure update using SB3.1 file format. Supports firmware update mechanism with authenticity (ECDSA P-256/P-384 signed) and confidentiality (AES-CTR encrypted and SHA256 chained) protection.

2.2.5 Secure debug

While secure boot ensures that only properly signed (OEM-authentic) code can be executed on a device, protecting debug access is of utmost importance. Failing to do that compromises the whole notion of secure boot. For example, an attacker can attach a debugger and execute an arbitrary code, invalidating the purpose of secure boot.

On this chip, the ROM supports certificate-based debug authentication mechanism using ECDSA P-256 and P-384 keys.

2.2.6 Secure isolation

The chip has multiple processing and power domains with bus architecture partitioned for a balance of optimal performance and power consumption efficiency based on target use case applications. Having separate processing engine domains (CPU0, CPU1, and ELE) provides process isolation but these are interconnected through bus fabric with onchip memories and peripherals which need further cross-domain protections. The chip uses the following components to provide secure isolation:

- Segmenting SoC by functional domains with separate bus matrix
- Secure enclave with dedicated processing engine and cryptographic accelerators providing fully isolated security functions
- Further isolation within processing engine (PE) is provided using ARM TrustZone for Armv8-M architectures
  - Memory Protection Unit inside Cortex-M33 core. CPU0 has 8 MPU regions
  - Security Attribute Unit inside Cortex-M33 core. CPU0 has 8 SAU regions

2.2.7 Secure key storage

At boot stage Device Unique Key (DUK) is generated and kept in key storage in the ELE. DUK is used to generate key encryption key, which is used to protect user key kept in NVM flash. Both symmetric and asymmetric private key is supported as part of key storage.

2.2.8 Trust provisioning

Trust provisioning is a process used for creation of initial Device Identity keys. Its major objective is to provide a cryptographic proof of the device’s origin and to offer a set of tools to OEM for secure provisioning of their own assets. In a nutshell, a device-unique private-public key pair is created on every device, the public portion of which is collected and signed by NXP. That signed public key is installed back onto every device in a form of device-unique certificate, which serves as the actual proof of the device’s origin. The corresponding private key, together with other pre-installed key material, is then used for authentication and secure connection to the device, enabling secure provisioning of OEM assets even in a manufacturing environment OEM may not fully trust.

2.2.9 Secure key management

Secure key management is a process of securing valuable keys and various key material which are essential in maintaining security of the end-user, OEM and NXP assets. The process strongly relies on the Immutable RoT consisting of UDF, hardware logic and ROM. A device-unique master key is provided by UDF and only used for further key derivation. Part of the derivation data is supplied by the Immutable RoT, accurately reflecting the device’s state, making sure different keys are derived in different states. For example, different life-cycle state will often yield a different derived key. Similarly, when the debug port is open a different key will be derived than when the debug port is closed. All the platform keys reside within a security subsystem, hidden from the application core at all times.

2.2.10 Anomaly detection and reaction

Anomaly Detection and Reaction describes the processes or algorithms that analyze the device input and output such as sensor data, as well as the software integrity and application operation for abnormal events and, if required, trigger and execute an action.

Typically these actions encompass logging the anomaly, issuing a message to the cloud backend, resetting the device, and/or changing a life cycle state.

Chapter 3
Lifecycle States

3.1 Overview

EdgeLock secure enclave (ELE) controls the SoC lifecycle and transition between security states.

This section describes the lifecycle states, assets availability for each state, possible transitions, triggers and conditions for transition.

Lifecycle states of SoC play important role in protecting security sensitive assets and capabilities throughout the life of SoC.

Each identified lifecycle state defines specific capabilities, functionalities, keys and other assets available in it and any underlying restrictions to use those.

3.2 Lifecycle states and transitions

Figure 4 describes the lifecycle states and transitions, with the SoC state shown provisioned and/or configured as it is expected to be in the respective lifecycle state (e.g. when entering the respective state). A detailed description of each state, triggers and conditions for the transitions follows.

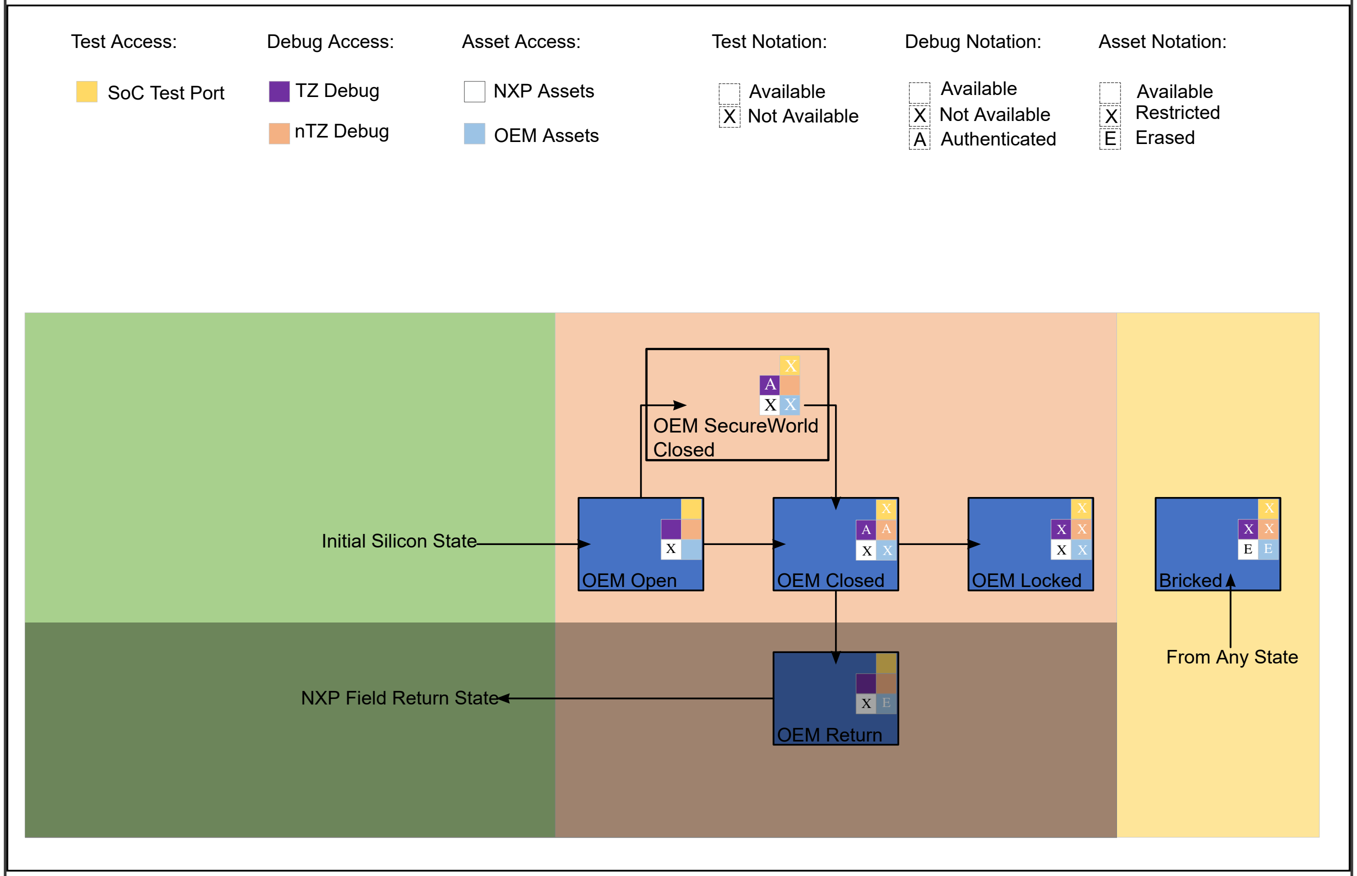


Figure 4. Security lifecycle states

NOTE

The Cortex-M33 with TrustZone technology provides secure and non-secure processing domains within a single core. Debugging of these processing domains are isolated and denoted as "TZ debug" for the secure domain and "nTZ debug" for the non-secure domain in the diagram above and throughout this chapter.

3.3 Lifecycle states

Lifecycle state is the combination of data items provisioned to the SoC, configuration of capabilities enabled and accessible, that aggregated result in a SoC state used at a given point in the process of transforming a die on a wafer into a real-life usable product running multiple layers of FW and software on said SoC. The lifecycle state is typically retained in fuses and enforced by HW and FW.

The table below lists the life cycle states and shows the debug and test port availability and asset access for each state.

Table 2. Lifecycle states

LIFECYCLE [0:7]	Lifecycle state	Debug authentication required to debug main core?	Debug authentication required to debug radio core?	ISP commands	NXP assets	OEM assets
0000_0111	OEM Open	No	No	All	Restricted	Available
0000_1111	OEM Secure World Closed	Yes	Yes	Limited	Restricted	Restricted
0001_1111	OEM Closed	Yes	Yes	Limited	Restricted	Restricted
1001_1111	OEM Locked	NA	NA	Limited	Restricted	Restricted
0011_1111	OEM Return	No	Yes	None	Restricted	Erased

3.4 Customer lifecycle state

The following sections give descriptions of the customer life cycle states—OEM - Open, OEM – Secure World Closed, OEM - Closed, and OEM - Locked. These states are intended for regular customer use, but some of the life cycle states are optional (for example, Secure World Closed, Locked, and OEM - Return). The customer life cycle states are described in detail along with information on all the valid state transitions.

3.4.1 Open lifecycle state (LIFECYCLE = 0x0)

The devices delivered from NXP are in the Open life cycle state. In this state, all customer chip functionality is accessible. This mode is recommended for early software development.

In the Develop state:

Test and debug ports are open by default. If the debug authentication field (CC_SOCU_xxx) are programmed, then they are used to determine debug access.

IFR0 region is write-once. Use IFR0 region to configure boot settings.

Secure boot is enabled. If an image is signed, signature verification is performed (against the OEM RoT fuses - CUST_PROD_OEMFW_AUTH_PUK), and the image will be allowed to run regardless of the signature verification result.

From the Open life cycle, the device can be advanced to Secure World Closed, or Closed.

3.4.1.1 Transitioning from open to secure world closed lifecycle state

Devices can be moved from Open to Secure World Closed life cycle state to allow for debug and development of non-secure world code (NS) while the secure world code and debug is protected.

Before transitioning from Open to Secure World Closed the following features must be configured:

- Secure Boot
  - — Root of Trust Key Hash - CUST_PROD_OEMFW_AUTH_PUK
  - — Optional: CUST_PROD_OEMFW_ENC_SK

  - Optional: IFR0 configurations
- TrustZone
  - TrustZone configuration can be done as part of the secure application code
  - TrustZone preset data can be included as part of the image manifest area
- Debug Authentication Settings
  - Optional: DCFG_CC_SOCU_L1[8:0]
  - Optional: DBG_AUTH_VU[15:0]
- Prince Encryption/Decryption for on-chip flash
  - Optional: NPX settings in IFR0

To transition from Open to Secure World Closed state the following fuse needs to be blown:

- LIFECYCLE: Fuse Index 0xA needs to be changed to 0000_1111

3.4.1.2 Transitioning from open to closed lifecycle state

Devices can be moved from Open to the Closed life cycle state when development is complete and product is ready to be deployed in the field.

Before transitioning from Open to Closed the following features must be configured:

- Secure Boot
  - Root of Trust Key Hash - CUST_PROD_OEMFW_AUTH_PUK
  - Optional: CUST_PROD_OEMFW_ENC_SK
  - Optional: IFR0 configurations
- TrustZone
- Debug Authentication Settings
  - Optional: DCFG_CC_SOCU_L1[8:0]
  - Optional: DCFG_CC_SOCU_L2[8:0]
  - Optional: DBG_AUTH_VU[15:0]
- Prince Encryption/Decryption for on-chip flash
  - Optional: NPX settings in IFR0

To transition from Open to Secure World Closed state the following fuse needs to be blown:

- LIFECYCLE: Fuse Index 0xA needs to be changed to 0001_1111

3.4.2 Secure world closed lifecycle state (LIFECYCLE = 0xF)

Secure World Closed is an optional life cycle that can be used to allow for development of NS world code while providing protection for S-world code.

In the Secure World Closed state:

- The test ports and S-world debug port are closed. The S-world debug port can optionally be opened using the debug authentication mechanism (which must have been enabled while the device was in the Open life cycle state).
- The NS-world debug port is open by default. If the debug authentication field (DCFG_CC_SOCU_L2) is programmed, then it is used to determine debug access.
- Secure boot is enabled.

- The primary image is responsible for configuring the TZ-M settings. Use of the TrustZone preset data to configure the TZ settings is recommended.
- Limited ISP commands are allowed (GetProperty, Reset, SetProperty, ReceiveSbFile, and TrustProvisioning).

From the Secure World life cycle, the device can be advanced to Closed state.

3.4.2.1 Transitioning from secure world closed to closed

Devices can be moved from Secure World closed to the Closed life cycle state when development is complete and product is ready to be deployed in the field.

Before transitioning from Secure World Closed to the Closed state the following optional features must be configured:

- NS-world debug authentication settings
  - — Optional: DCFG_CC_SOCU_L2[8:0]
  - — Optional: DBG_AUTH_VU[15:0]

To transition from Secure World Closed to the Closed state the following fuse must be blown:

- LIFECYCLE: Fuse Index 0xA needs to be changed to 0001_1111

Software must call the ROM APIs or request the service from the ELE.

3.4.3 Closed lifecycle state (0x1F)

The Closed state is the primary state intended to be used for deployment of products to end-customers in the field.

In the Closed state:

- Test and debug ports are closed by default. If debug authentication is enabled, then the authentication process can be used to reopen debug ports.
- Secure boot is enabled.
- Use of TZ-M is optional.
- Limited ISP commands are allowed (GetProperty, Reset, SetProperty, ReceiveSbFile, and TrustProvisioning).

From the Closed life cycle, the device can be advanced to Locked, or OEM-Returned state.

3.4.3.1 Transitioning from closed to locked lifecycle state

Devices can be moved from Closed to the Locked life cycle state if it is determined that field return functionality is not needed.

To transition from Closed to Locked the following fuse must be blow:

- LIFECYCLE: Fuse Index 0xA needs to be changed to 1001_1111

Software must call the ROM APIs or request the service from the ELE.

3.4.3.2 Transitioning from closed to OEM return lifecycle state

If a device deployed to the field is returned for reported failure, then the device can be moved from the Closed to the OEM-Returned state to disable normal operation and re-enable testing of the device.

To transition from Closed to OEM-Returned, NXP recommends the following:

1. Use debug authentication to enable permission to send the ‘Set FA Mode’ command through the debug mailbox.
2. Send ‘Set FA Mode’ debug mailbox command. This command triggers ROM to handle:
   a. Erase all OEM Keys in fuse
   b. Erase the internal flash
   c. Flush all temporary key registers

d. Sets the LIFECYCLE fuse = 0x3F

e. Reset the Device

3.4.4 Locked lifecycle state (LIFECYCLE = 0x9F)

The Locked state is an optional state intended to be used for deployment of products to end-customers in the field for products that do not support field return or failure analysis. Most of the behavior in this mode is the same as the In-field state, but the debug and test ports can never be fully opened again.

In the Locked state:

- Test and debug ports are closed. The debug authentication mechanism cannot be used to re-enable the debug port.
- Secure boot is enabled.
- Use of TZ-M is optional.
- Limited ISP commands are allowed (GetProperty, Reset, SetProperty, ReceiveSbFile, and TrustProvisioning).

From the Locked cycle, the device cannot be transitioned to any other state.

3.5 Field return states

The following section describes the field return life cycle states—OEM-Returned. These states are used to support various stages of field return debugging.

Stage 1: Initial investigation and failure duplication phase where the device remains in the Closed state. The debug authentication mechanism can be used to re-open the debug port(s) and gain additional information on the reported failure.

Stage 2: If the issue needs further control of the device for investigation beyond what the debug authentication process allows, then the device can be moved to the OEM-Returned life cycle state for additional debugging and testing.

Stage 3: If a silicon structural or manufacturing issue is suspected after stage one and stage two investigations, then the device is returned to NXP in the OEM-Returned state.

3.5.1 OEM-Returned lifecycle state (LIFECYCLE = 0x3F)

The OEM-Returned state can be used to debug products returned by end customers.

In the OEM-Returned life cycle state:

- On every boot ROM verifies that the flash and OEM assets are blank. If not, ROM erases these areas before opening debug access. This mechanism protects leakage of any residue data left during lifecycle state transition.
- Test and debug ports are open.
- ROM stays in a while(1) loop and does not pass execution to application code. The debug port can be used to load and then execute diagnostic firmware.

From the OEM-Returned lifecycle, users may return to NXP for further failure analysis.

Chapter 4 ROM Bootloader

4.1 Overview

ROM bootloader is the boot code resident in the internal read-only memory (ROM). The ROM bootloader begins its execution when the Cortex-M33 processor is released from reset. This chapter highlights the features supported by the bootloader and describes its execution flow logic.

4.2 Boot ROM

In this device, the ROM bootloader supports automated booting from internal flash and downloading image from serial interface (in-system programming via LPUART, LPI2C, LPSPI, CAN).

The ROM bootloader is in the memory region starting from the address 0x14800000. The following diagram shows the memory map in-use during ROM execution. The first 32 KB of Tightly Coupled Memory - System (STCM) (0x30000000 – 0x30008000) needs to be reserved for ROM bootloader execution.

NOTE

STCM has ECC enabled. To avoid STCM ECC error, users must always program word-aligned data to word-aligned address before the first time reading after power-on reset.

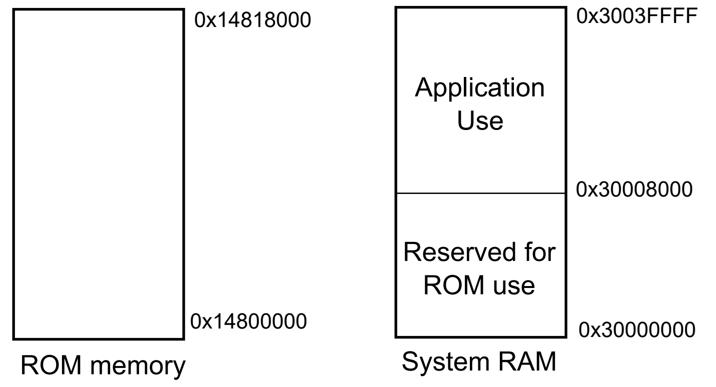


Figure 5. ROM bootloader memory usage

There are several alternate paths through the ROM bootloader:

- Normal boot path with security and TrustZone Mode (TZ-M) option.
- In-system programming (ISP) path.
- Debugger mailbox path which supports debug authentication.

The decision about which path to take depends on fuses, boot configuration in user IFR, boot pin, debug request and low power wakeup configuration. Figure below shows the top-level boot flow.

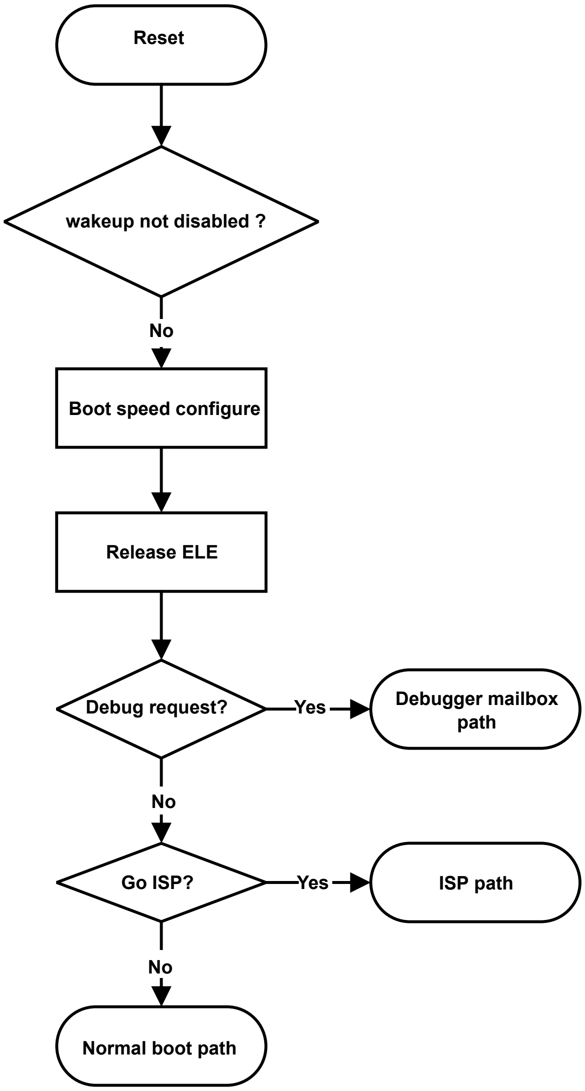


Figure 6. Top-level boot flow

4.2.1 Lifecycle and fuses

The table and diagram below describe the lifecycle states and transitions. ROM behaviour is different at different lifecycle state. The lifecycle fuse needs to be programmed correctly, otherwise ROM bootloader will not boot. The NA in the table means the feature is not available.

> NOTE
>
> In addition to the lifecycle fuses, there are a handful of other fuse fields that will be used by the ROM bootloader during its operation. See the Fusemap spreadsheet attached to this document for details about these fuse fields.

Table 3. Lifecycle states

LIFECYCLE	Lifecycle state [0:7]	Debug authentication required to debug main core?	Debug authentication required to debug radio core?
0000_0111	OEM Open	No	Yes

Table continues on the next page...

Table 3. Lifecycle states (continued)

LIFECYCLE	Lifecycle state [0:7]	Debug authentication required to debug main core?	Debug authentication required to debug radio core?
0000_1111	OEM Secure World Closed	Yes[1]	Yes
0001_1111	OEM Closed	Yes	Yes
1001_1111	OEM Locked	NA	NA
0011_1111	OEM Return	No	Yes

1. For secure world only.

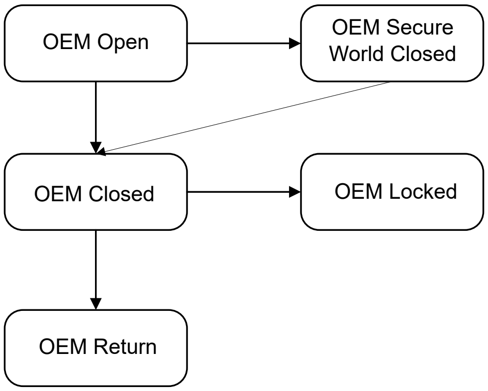


Figure 7. Lifecycle transitions

ROM bootloader provides the following ways to do the fuse reading and programming:

- nboot API (see nboot API for details about fuse programming using nboot API);
- ISP fuse-program and fuse-read command (see FuseProgram command and FuseRead commandfor details);
- sbloader “programFuses” command in sb3.1 file (see receive-sb-file for details).

4.2.2 User IFR (IFR0) configuration

User IFR, known as IFR0 is reserved for software usage. First sector of User IFR is not able to be erased and is write-once. Other three sectors are erasable and programmable.

The allocation of User IFR pages is as follows:

Table 4. User IFR allocation

Sector	Start address	End address	Name	Description
0	0x02000000	0x2001FFF	ROMCFG	ROM Bootloader configurations
1	0x02002000	0x2003FFF	User	Reserved for customer usage
2	0x02004000	0x2005FFF	CMAC table	Used to save hashes of multiple boot components.
3	0x02006000	0x2007FFF	OTACFG	Used for Over-the-Air update

4.2.2.1 ROM bootloader configuration

Sector 0 of User IFR is ROM and ISP configuration, which provide configuration for ROM bootloader.

Table 5. ROM configuration fields

Offset	Size (bytes)	Configuration Field	Description
0x0000	1	Restore Flash Logical Window (FLW) Flag	Bit 0 – FLW state, whether enable dual image boot • 1 = Uninitialized • 0 = Normal, use specified mapping Bit 7:1 – Reserved
	7	Reserved	Reserved
	8	FLW region definition	Fields to be used for FLW configuration
0x0010	1	Configure NPX for Normal Boot	Bit 0 – Configure NPX to use PRINCE • 1 = Not configure NPX • 0 = Configure NPX Bit 1: • 0=Global lock enable • 1=Global lock disable Bit 2: • 0=System lock enable • 1=System lock disable Bit 3: • 0=Global decryption enable • 1=Global decryption disable Bit 4: • 0=Global encryption enable • 1=Global encryption disable Bit [7:5] – Reserved
	1	Sticky NPX Configuration for Waking up from Deep Power Down	Bit 0 – Configure NPX required: • 1 = Not required; ROM follows Low-power wakeup path • 0 = NPX config. required; ROM follows normal boot path Bit [7:1] – Reserved
	6	Reserved	Reserved
	56	NPX Region Definitions	Setting for NVM PRINCE XEX, the inline flash encryption IP module
0x0050	1	Boot Configuration	Bit 0 - Enable ISP or not • 0 = BOOT_CFG pin disabled • 1 = BOOT_CFG pin enabled Bit [2:1] – Boot speed

Table continues on the next page...

Table 5. ROM configuration fields (continued)

Offset	Size (bytes)	Configuration Field	Description
			• 00 = Normal boot (48 MHz) • 10 = Fast boot (96 MHz) • 01, 11 = Normal boot (default) Bit 3 - Boot low power • 0 = lower DCDC voltage • 1 = normal DCDC voltage See Table 6 for details. Bits [7:4] - Reserved
	15	Reserved	Reserved
0x0060	1	Secure boot option	The secure boot failure log will show in secure boot fail alert pin when this bit and secure boot fail alert pin are enabled. Bit 0 – Reserved Bit 1 – Secure boot option • 0 = Enable • 1 = Disable (default) Bit [7:2]– Reserved
	1	Secure boot fail alert pin	Set the boot fail alert pin Bit [7:5] – Port Selection • 0x0~0x4 = PortA ~PortE • 0x5~7 Reserved Bit [4:0] – Pin Selection • 0x0~0x1F = Pin0~Pin31
	14	Reserved	Reserved
0x0070	16	Reserved	Reserved
0x0080	64	Boot Image Base Address	Start address of Boot image stored in the first 32 bits of a 16-byte phrase; the value should be aligned to 32 KB. There are four 16-byte slots available. The ROM Bootloader does search the slots from address 0x80 and the first address not being 0xFFFFFFFF will be the boot address. Therefore, it is better to program IFR0 from address 0xB0 to 0x80, which allows to update the boot address 4 times.
0x00C0	64	Reserved	Reserved
0x0100	1	Peripherals Enable	Bit 0 – LPUART1 peripheral for ISP • 0 = LPUART1 disabled

Table continues on the next page...

Table 5. ROM configuration fields (continued)

Offset	Size (bytes)	Configuration Field	Description
			• 1 = LUPART1 enabled Bit 1 – LPI2C1 peripheral for ISP • 0 = LPI2C1 disabled • 1 = LPI2C1 enabled Bit 2 – LPSPI1 peripheral for ISP • 0 = LPSPI1 disabled • 1 = LPSPI1 enabled Bit 3 – CAN peripheral for ISP • 0 = CAN0 disabled • 1 = CAN0 enabled Bit [7:4] – Reserved
	3	Reserved	Reserved
	2	Peripheral Detection Timeout	If 0xFFFF, defaults to no timeout. If not 0xFFFF, use this value as timeout in milliseconds for active peripheral detection.
	2	Reserved	Reserved
	1	I2C Target Address	If 0xFF, defaults to 0x10 for LPI2C target address. If not 0xFF, use this value as the 7-bit LPI2C target address.
	1	Reserved	Reserved
	2	Reserved	Reserved
	2	CANTxID	TxID
	2	CANRxID	RxID
0x110	4	ELE Loadable FW Entry Address	ELE loadable firmware resides address
	12	Reserved	Reserved
0x120	4	Secure Hash based image verification feature (for faster subsequent boot)	0xFFFFFFFF – Disabled (ECDSA based verification will be used). If 0x48534148 (“HASH”) – Enabled.
	12	Reserved	Reserved
0x130	16	Reserved	Reserved
0x140	1	Go ISP mode options	If enter ISP mode, the log will be recorded on the corresponding pin when this bit and GoIspModeFailAlertPin are enable. Bit 0 : reserved

Table continues on the next page...

Table 5. ROM configuration fields (continued)

Offset	Size (bytes)	Configuration Field	Description
			Bit 1 : Secure boot option • 0 = enable • 1 = 1 disable (default) Bit [7:2] - Reserved
	1	Go ISP mode fail alert pin	Set the ISP log pin Bit [4:0] - Pin selection • Value: 0x0~0x1f - Pin0~Pin31 Bit [7:5] - Port selection • Value: 0x0~0x4 - PORTA~PORTE
	14	Reserved	Reserved

Restore FLW Flag and FLW Region Definitions fields are Flash Logic Window configurations used for dual image boot. When dual image boot is enabled, there are two boot images on the internal flash. ROM bootloader first tries to boot the boot image with latest version, if fails, ROM bootloader tries to boot the other boot image. To enable the dual image boot, Restore FLW Flag needs to be enabled, Boot Image Base Address field and FLW Region Definition fields need to be configured correctly. Boot Image Base Address indicates the start address of one boot image. FLW Region Definitions indicate the start address and the size of the alternative boot image. Here is the FLW Region Definition structure:

```
struct flw_region {
   uint32_t abase;
   uint32_t bcnt; };
```

The FLW region descriptors simply consist of FLW ABASE, the starting address of the alternative boot image, and, FLW BCNT, the size of the alternative boot image in 32 KB blocks.

If dual image boot is not enabled, ROM bootloader boots from Boot Image Base Address. If Boot Image Base Address is not configured, ROM bootloader will boot from the beginning of flash.

Configure NPX for Normal Boot, NPX Region Definitions fields are used for NPX feature. See NPX configuration for more details about NPX.

Bit 0 of Boot Configuration field is used to enable ISP path for ROM bootloader. BOOT_CFG pin will be enabled if this bit is set. The pull-down on the BOOT_CFG pin is to ensure the device does not enter ISP by default. When BOOT_CFG pin is enabled and the BOOT_CFG pin is not pulled down, ROM bootloader will enter ISP path.

Boot Speed bits will be loaded by ROM bootloader to decide how to configure the power and clock. Bit 1:2 of Boot Configuration indicates the boot speed:

Table 6. Boot speed

Boot speed value in IFR0	Clock source	CPU_CLK/BUS_CLK/SLOW_CLK	DCDC & CORE_LDO voltage
11/00/01: Normal Boot (default)	FRO_192M	48/48/24 MHz	CORE_LDO: 1.0 V DCDC: 1.8 V or 1.25 V

Table continues on the next page...

Table 6. Boot speed (continued)

Boot speed value in IFR0	Clock source	CPU_CLK/BUS_CLK/SLOW_CLK	DCDC & CORE_LDO voltage
10: Fast Boot	FRO_192M	96/96/24 MHz	CORE_LDO: 1.1 V DCDC: 1.8 V or 1.35 V

Secure Boot Failure Options and Secure Boot Failure Alert Pin Selection provide options for users to decide what ROM bootloader should do when secure boot fails. See Security features of boot ROM for more details.

The ELE Loadable FW Entry Address indicates the flash address where the ELE loadable firmware resides. If configured, ROM bootloader tries to load the firmware to the ELE. See Secure firmware update for more details about the ELE loadable firmware.

4.2.2.2 IFR0 Sector 2 (hash structure/cmac table maintenance)

This IFR0 dedicated page is used to store hash for boot images in a structured format as shown below.

Table 7. IFR0 sector 2 configuration fields

Offset	Size (bytes)	Field description	Description
0x0000	4	Hash update needed	0xFFFFFFFF - Yes 0x00000000 – No
	12	Reserved	Reserved
0x0010	16	16 byte CMAC for application image	Hash of main flash primary boot image data at 0x0, or Hash of Image data at ABASE (if FLW is enabled)
0x0020	4	Hash update needed	0xFFFFFFFF - Yes 0x00000000 – No
	12	Reserved	Reserved
0x0030	16	16 byte CMAC for fallback image	Hash of fallback image data (if “Image 1 Base Address” field in IFR0 page0 is not 0xFFFFFFFF)
0x0040	4	Hash update needed	0xFFFFFFFF - Yes 0x00000000 – No
	12	Reserved	Reserved
0x0050	16	16 byte CMAC for radio image	Hash of radio (NBU) flash image data
0x0060	4	Hash update needed	0xFFFFFFFF - Yes 0x00000000 – No
	12	Reserved	Reserved
0x0070	16	16 byte CMAC for s200 image	Hash of s200 image data

4.2.2.3 Over-the-air (OTA) update configuration

Table 8. OTA update configuration

Offset	Size (bytes)	Configuration Field	Description
0x00	4	Update available	0x746f4278u = Indicates update is available
	12	Reserved	Reserved
0x10	4	Update(sb3) dumped location	0x74784578 = sb3 is dumped in external flash
			Other values = sb3 is dumped in internal flash
0x14	4	Baud rate (in bps)	Value used to configure LPSPI NOR flash if update(sb3) is dumped in external flash.
0x18	4	Update(sb3) dump address	Start address (internal/external flash) where update(sb3) is dumped.
0x1c	4	Update(sb3) file bytes	Size of update(sb3) file in number of bytes.
0x20	4	FW update status	0x5ac3c35a = Indicates update was success 0x4412d283/0x2d61e1ac = Indicates failure to process sb3 file OR failure to erase/write update status to OTACFG page
	12	Reserved	Reserved
0x30	16	Feature unlock key	To use FW update feature offered by ROM (instead of extended bootloader) 16-byte key must be programmed. updKey[16] = {0x61, 0x63, 0x74, 0x69, 0x76, 0x61, 0x74, 0x65, 0x53, 0x65, 0x63, 0x72, 0x65, 0x74, 0x78, 0x4d}

4.2.3 Normal boot path

ROM bootloader follows this path when device goes through a WARM reset or PoR if not getting request to go to another path. ROM bootloader validates the application code residing in the internal flash and boot when validation is successful. By default, when there is no fuse/IFR0 boot configuration programmed, ROM bootloader does validation for boot image based on image type. It does basic SP and PC value check for raw binary application; CRC check for CRC image and ECDSA authentication for signed image. Secure boot can be enabled and enforced by setting OEM_SEC_BOOT_EN fuse. Radio image sits on radio flash is not validated by ROM bootloader by default. NBU_SEC_BOOT_EN can be used to enable the authentication for radio image during boot time. If authentication for radio image is enabled but failed during boot time, then radio core will be locked, and it can't be released from reset.

Note that WDOG0 is disabled during normal boot path and enabled before jumping to the boot image. Meanwhile, the WDOG0 timeout value (TOVAL) has been set to longest (0xFFFF).

Figure 8 shows the flow of normal boot.

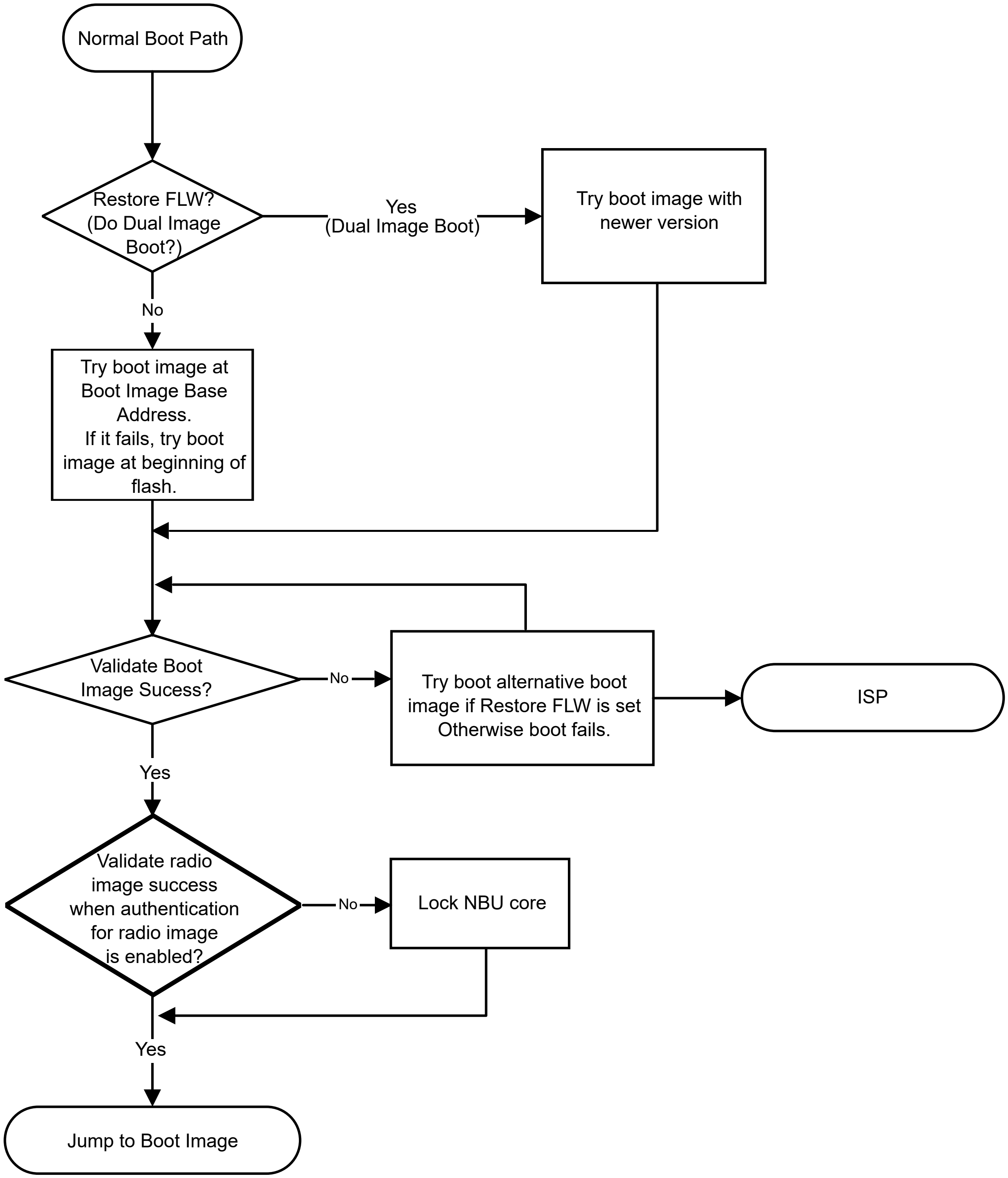

Normal Boot Path
Restore FLW? (Do Dual Image Boot?)
Yes (Dual Image Boot)
Try boot image with newer version
No
Try boot image at Boot Image Base Address. If it fails, try boot image at beginning of flash.
Validate Boot Image Sucess?
No
Try boot alternative boot image if Restore FLW is set Otherwise boot fails.
ISP
Yes
Validate radio image success when authentication for radio image is enabled?
No
Lock NBU core
Yes
Jump to Boot Image

Figure 8. Normal boot flow

4.2.3.1 Dual image boot

Boot ROM supports dual image boot, useful in scenario where primary image fails to boot for any reason so secondary image (as a backup) can boot. This is achieved in conjunction with Flash Logical Window feature. It will allow developing for position independent code. That means same linker file can be used for creating multiple application images to be treated as primary (experimental or latest firmware image) and secondary (as a backup redundant image).

Use case example of dual boot:

1. “Restore FLW Flag” bit 0 is set to indicate ROM to use FLW feature.
2. Primary image is placed at an arbitrary address (e.g. 0x10000) and “Boot Image Base Address” field of ROMCFG in user IFR0 is programmed accordingly.
3. Secondary image (fallback/backup) is placed at address 0x20000 and “ABASE” field in FLW region definitions of ROMCFG in user IFR0 is programmed accordingly.
4. Make sure BCNT field of FLW region definitions is set correctly to indicate image size as multiple of 32 KB block size.
5. Boot ROM will compare versions of primary and secondary image.
6. Boot ROM will try to boot latest versioned image. At the same time older versioned image base address is stored in ROM runtime context.
7. If both images have same version, then image located at address specified by “Boot Image Base Address” field of ROMCFG in user IFR0 is considered for boot.
8. For any reason (mainly authentication failure or rollback protection) latest versioned image fails to boot then boot ROM falls back to the older versioned image.

4.2.4 Faster subsequent boot

Boot ROM supports secure boot primarily using an ECDSA verification. And it can also support secure boot using hash (cmac) based image verification for faster subsequent boot.

- The user needs to:
  - Program IFR0 sector 0 at offset 0x120 to request secure hash based image verification.
  - Include an sbloader command to erase IFR0 sector 2 for every FW update.
- The ROM handles this request with the following steps:
  - First time after resetting to boot or trying to boot using jump/execute sbloader command after successful FW update, requires ROM to compute hash of image data.
  - Boot ROM computes hash (cmac) for multiple boot components, namely main flash image (application) and radio FW, and stores it in an IFR0 sector 2 in structured format. So it can be used to achieve faster boot time from next boot attempt.
  - But for very first time post FW update, after calculating hash of image data, boot ROM will still use ECDSA based image verification mechanism to perform secure boot.
  - For all subsequent FW updates, it is user’s responsibility to erase IFR0 sector 2. So that boot ROM understands it needs to calculate hash for multiple bootable components and store it in a structured format.
  - If no FW update is required/performed then triggering multiple resets could utilize faster secure boot sequence.
- If hash based image authentication fails
  - Images are still expected to have signature. So boot ROM would fall back to ECDSA based verification as a primary secure boot mechanism.

4.2.5 Low power wakeup path

The ROM bootloader provides a fast low power wakeup mechanism that supports all low power modes, including:

- Power Down (PD) Mode
- Deep Power Down (DPD) mode
- Smart Power Switch (SPS) Mode

When the WAKEUP_DIS flag is cleared (WAKEUP_DIS = 0) and the sticky NPX configuration for DPD wakeup is not enabled, the ROM bootloader can execute low power wakeup path after waking up from low power mode. ROM bootloader does normal boot if low power wakeup path fails.

Figure 9 illustrates the low power wakeup path.

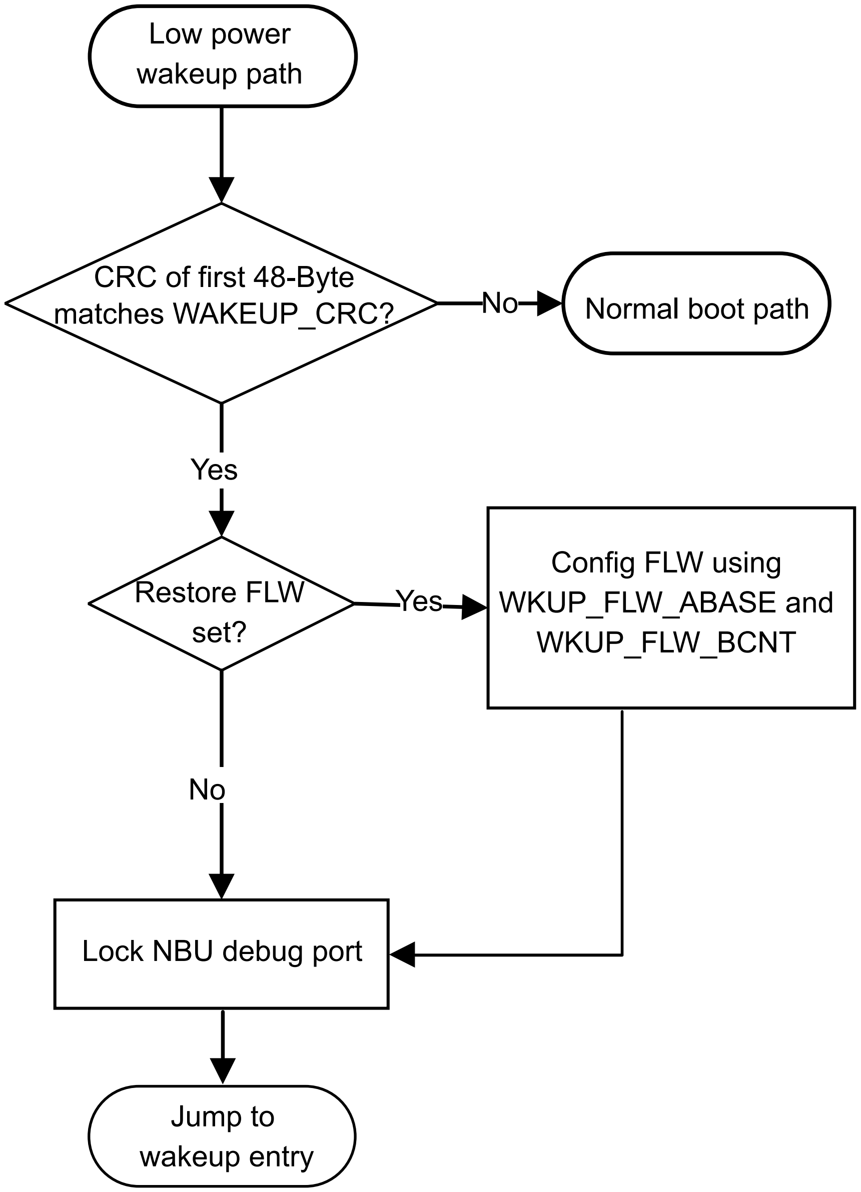


Figure 9. Low-power wakeup flow

4.2.5.1 Wakeup from PD mode and DPD mode

When the wakeup path is enabled (i.e., WAKEUP_DIS = 0) and the system wakes up from Power Down (PD) or Deep Power Down (DPD) mode, the ROM bootloader enters the low power wakeup path if the WAKEUP register in SPC0 is set to a non-zero value.

If the CRC value preloaded into REG[0] of REGFILE1 matches the CRC computed over the first 48 bytes of the wakeup image, the ROM bootloader transfers control to the wakeup image at the entry point specified in the WAKEUP register in SPC0. Otherwise, the ROM bootloader exits the low power wakeup path and proceeds with the normal boot sequence.

If the Flash Logic Window (FLW) must be configured for the low power wakeup path, the following values must be correctly set. The ROM bootloader uses WAKEUP_FLW_ABASE and WAKEUP_FLW_BCNT to configure the FLW for low power wakeup path.

- FLW restore flag in IFR0
- WAKEUP_FLW_ABASE in REG[1] of REGFILE1
- WAKEUP_FLW_BCNT in REG[2] of REGFILE1 (Bit 31: Valid bit, Bit 30: Lock bit, Bits 0–14: Block count)

NOTE

If the system boots successfully via the low power wakeup path, the ROM bootloader does not modify WDOG0. In this path, the ELE remains unreleased, and the CM3 debug port remains locked.

4.2.5.2 Wakeup from SPS mode

Compared to PD and DPD modes, SPS mode offers significantly lower leakage power. In SPS mode, all RAM instances are automatically power-gated, resulting in complete data loss across RAM - except for the last 8 KB of RAM (TCM8), which can remain powered and retain data.

This retained memory region enables the ROM bootloader to execute a fast low power wakeup path. The last two words of TCM8 are reserved by the ROM bootloader for:

- WAKEUP Entry Point at offset 0x1FFC
- WAKEUP_CRC at offset 0x1FF8

During wakeup, if the CRC value stored in WAKEUP_CRC matches the CRC computed over the first 48 bytes of the wakeup image, the ROM bootloader transfers control to the wakeup image at the entry point specified in WAKEUP Entry. If the CRC check fails, the ROM bootloader exits the low power wakeup path and proceeds with the normal boot path.

When using the low power wakeup path from SPS mode, the FLW restore flag in IFR0 must not be configured. FLW restoration is not supported in this wakeup scenario.

4.2.6 Firmware update feature

ROM Bootloader has firmware update feature. It can be used for updating main flash as well as radio flash firmware.

Use case example of remote radio firmware update:

1. Main flash contains a customer's application image along with NXP delivered OTA client.
2. NXP OTA client helps in fetching image over-the-air via wireless protocol.
3. Image blocks fetched over-the-air are dumped in either internal or external flash unused regions.
4. Once the entire image is received, OTA client must indicate and provide meta data information for update to be performed in User IFR0 OTACFG page. (See OTA update configuration).
5. Application is responsible of triggering a software reset upon which boot ROM uses data provided in OTACFG page to start firmware update.
6. Upon successful update, IFR0 OTACFG page is erased and “FW update status” is updated by boot ROM and reset is triggered to follow "Normal boot path".

If the external flash is used, LPSPI1 is configured to controller mode and pin assignment is showed in table below.

Table 9. LPSPI1 pin assignment when external flash used and LPSPI1 configured to master mode

Pin name	Pin assignment	Alt
LPSPI1_SCK	PTB2	2
LPSPI1_SIN	PTB1	2

Table continues on the next page...

Table 9. LPSPI1 pin assignment when external flash used and LPSPI1 configured to master mode (continued)

Pin name	Pin assignment	Alt
LPSPI1_SOUT	PTB3	2
LPSPI1_DATA2	PTB5	2
LPSPI1_DATA3	PTB4	2
LPSPI1_SS0	PTB0	2

4.2.7 Clock

4.2.7.1 Clock gating

The heart of the clocking architecture is the System Clock Generator (SCG) module. The SCG controls multiple clock sources (FIRC/SIRC/SOSC/ROSC/CORECLK/BUSCLK/SLOWCLK) that can then be distributed to the Module Reset and Clock Control (MRCC) module. Clock gating of modules helps users to only consume power for modules that are needed for their end application. The clock to each module can be gated on or off via a programmable register.

Each peripheral has independent clock gating for both the peripheral interface clock and the peripheral functional clock. This clock gating is controlled via MRCC module. MRCC provides a clock select field, a clock disable, a divider of the selected clock source, and a peripheral reset bit that provides independent resetting of the peripheral. MRCC routes interface clocks and functional clocks for all peripherals.

The CPU_CLK, BUS_CLK, and SLOW_CLK are always enabled in the various RUN and Wait modes. If these clocks are not used in low-power stop modes, they will be disabled via the Core Mode Controller (CMC) and System Power Controller (SPC) modules.

4.2.7.2 Module/Peripheral clocking

Most peripheral clocks by default are disabled, ROM will un-gate the peripheral clocks as needed by the boot flow and also enables functional clock.

Here are some of the clocks used by ROM and their setup values:

- TRDC CPU_CLK
- CRC0 BUS_CLK
- LPSPI1 BUS_CLK
- LPUART1 BUS_CLK
- LPI2C1 BUS_CLK
- CAN BUS_CLK
- ELE CPU_CLK
- GPIO{B/C} CPU_CLK
- PORT{A/B} CPU_CLK

4.3 Security features of boot ROM

The secure part of ROM bootloader provides following basic operations:

- Secure boot
- Secure firmware update
- Security related miscellaneous functions

The ROM bootloader provides an API to allow integration of loader operations into customer applications.

4.3.1 Functional description

4.3.1.1 Secure boot

Secure boot provides guarantee that unauthorized code cannot be executed on a given product. It involves the device's ROM always executing when coming out of reset. The ROM will then examine the first user executable image resident in internal flash memory to determine the authenticity of that code. If the code is authentic, then control is transferred to it. This establishes a chain of trusted code from the ROM to the user boot code. The method used in this architecture to verify the authenticity of the boot code is explained in Image signature verification.

Device could be configured to boot plain images during development. In such case ROM does not check image to be booted, or perform only CRC32 checking, depending on configuration.

4.3.1.2 Secure firmware update

If firmware updates are to be performed in the field when secure boot is enabled, then a secure firmware update mechanism is preferred. Otherwise inauthentic firmware may be written to the device, causing it to not boot. In the most basic sense, secure firmware update simply performs an authentication of the new firmware prior to committing it to memory. In this case, the chain of trust is extended from the old, currently executing, code to the new code.

Another use case for secure firmware update is to hide the application binary code during transit over public media such as the web. This is accomplished by encrypting the firmware update image. As the new firmware is written into device memory, it is decrypted.

In this architecture, both cases of secure firmware update are supported. The SB file format is encrypted and digitally signed. SB file can be loaded via secure interfaces such as LPUART, etc. or can be provided to ROM API as complete binary file.

4.3.1.3 Rollback protection

Boot ROM provides a rollback protection (in terms of firmware version enforcement) in following ways:

1. Preventing firmware update if update file version is older than the version allowed by the system.
2. Preventing boot (secure boot failure) if firmware version is older than version allowed to boot.

For this device, 512-bits in fuse map are dedicated to maintaining versioning information about multiple software components. These software components are CM33 secure firmware, CM33 non-secure firmware, radio firmware, ELE loadable firmware. See the Fusemap spreadsheet attached to this document, which also shows virtual version counter fuses providing actual integer value of version as opposed to normal versioning fuses providing value based on number of set bits.

Use case example of rollback protection for firmware update using sb3:

1. Consider CM33_S_VER_CNT fuse is programmed as 0x1.
2. Firmware needs to be updated. So new sb3 update file is generated using "checkFwVersion" as one of the supported sbloader commands. Refer to elftosb user's guide to understand more. Example: {"checkFwVersion": {"counterId": "secure", "value": "0x3"}}
3. User can utilize ISP, kb API or remote update feature of ROM to update firmware image. Before firmware update starts boot ROM checks allowable firmware version of secure CM33 image update.
4. In this case actual firmware version in fuse 0x1 is less than value 0x3 for firmware update sb3 command, firmware update can proceed.
5. If CM33_S_VER_CNT fuse is programmed as value 0x3 or above, then failure from "checkFwVersion" would terminate firmware update process.

Use case example of rollback protection for secure boot:

1. Consider CM33_S_VER_CNT fuse is programmed as 0x1.
2. On a reset, boot ROM will follow normal boot path trying to authenticate CM33 image.

3. If a "firmwareVersion" of CM33 image is 0x0 or lower than allowable version to boot as per CM33_S_VER_CNT then rollback is detected, and image authentication fails resulting in boot failure. Refer to elftosb User's guide to understand a way to generate image with "firmwareVersion" field.
4. If a "firmwareVersion" of CM33 image is greater than or equal to a value programmed in fuse CM33_S_VER_CNT then rollback is not detected, and boot process can continue.

4.3.1.4 Extending the chain of trust

Once secure boot has transferred CPU control to user code, that code may need to load additional pieces of code. This establishes another link in the chain of trust. The process can continue for any number of nested sub-modules, with each parent code module authenticating the chain. Another use case is to authenticate boot code for one or more secondary CPU cores prior to releasing them from reset.

The loader API is used from customer code to verify signatures on the additional code images. Using the API to verify signatures gives complete control to the customer code over what additional code must be signed and how that code is organized in memory.

4.3.1.5 Miscellaneous functions

ROM provides support for various security related additional functionalities:

- Support for the load of TrustZone-M pre-configuration during ROM secure boot
- Support of booting from encrypted internal Flash regions using NPX
- Debug Authentication

4.3.2 Keys

There are four types of secure boot keys.

- Root of Trust Key (RoTK)
  - — Public part of RoTK ($\mathrm{RoTK}_{\mathrm{public}}$)
  - — Private part of RoTK ($\mathrm{RoTK}_{\mathrm{private}}$)
- Root of Trust Key Table Hash (RoTKTH)
- Image Signing Key (ISK)
- SB3 Key Derivation Key (SB3KDK)

1 to 4 asymmetric EC keypairs should be generated as a RoTK (RoTK can be based on secp256r1 or secp384r1 and all generated certificates need to be of the same EC type, cannot be mixed). The current device revision supports only secp384r1, and the secp256r1 will be supported in the next revision.

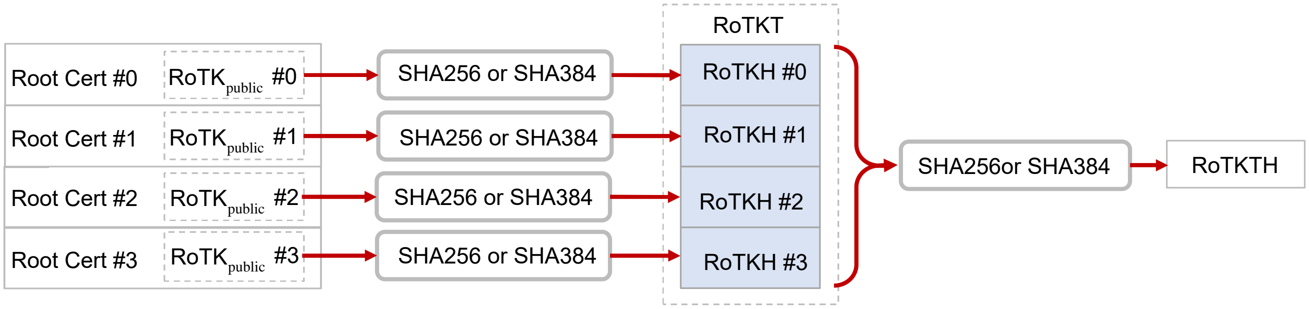


Figure 10. RoTKTH calculation

If only 1 RoTK is specified for RoTKTH, then the RoTKTH value is directly SHA256 or SHA384 of $\mathrm{RoTK}_{\mathrm{public}}$. If more than 1 root certificates is specified, RoTKTH is calculated as hash of hashes of $\mathrm{RoTK}_{\mathrm{public}}$, as shown in Figure 10. Whether SHA256 or

SHA384 will be applied depends on selected EC of RoTK. If secp256r1, then SHA256 is used, if secp384r1, then SHA384 is used as hash algorithm.

The RoTKTH value is used as root of trust for boot images, SB3.1 firmware update container and debug authentication.

ISK is used for signed boot image or SB3.1 firmware update container. The use of this key is optional. Basically, it is EC asymmetric key based on secp256r1 or secp384r1. Limitation is, the key length must be the same as or less than RoTK. Refer to Certificate block for more details about ISK.

The 256-bit SB3KDK key is used only for SB3.1 firmware update container as input for encryption key derivation process.

RoTKTH and SB3KDK keys are stored in device fuses, optional ISK key is only part of certificate block and signed by one selected RoTK key.

Up to 4 RoTK keys may be revoked, and up to 16 ISK keys may be revoked. Revocation is controlled by fuses. Refer to Certificate block for more details.

4.3.2.1 Signed image structure

The signed image container is intended to be used primarily for boot images and other pieces of code that are executed in place. However, it could also be used for images that are copied into RAM from an external device prior to execution.

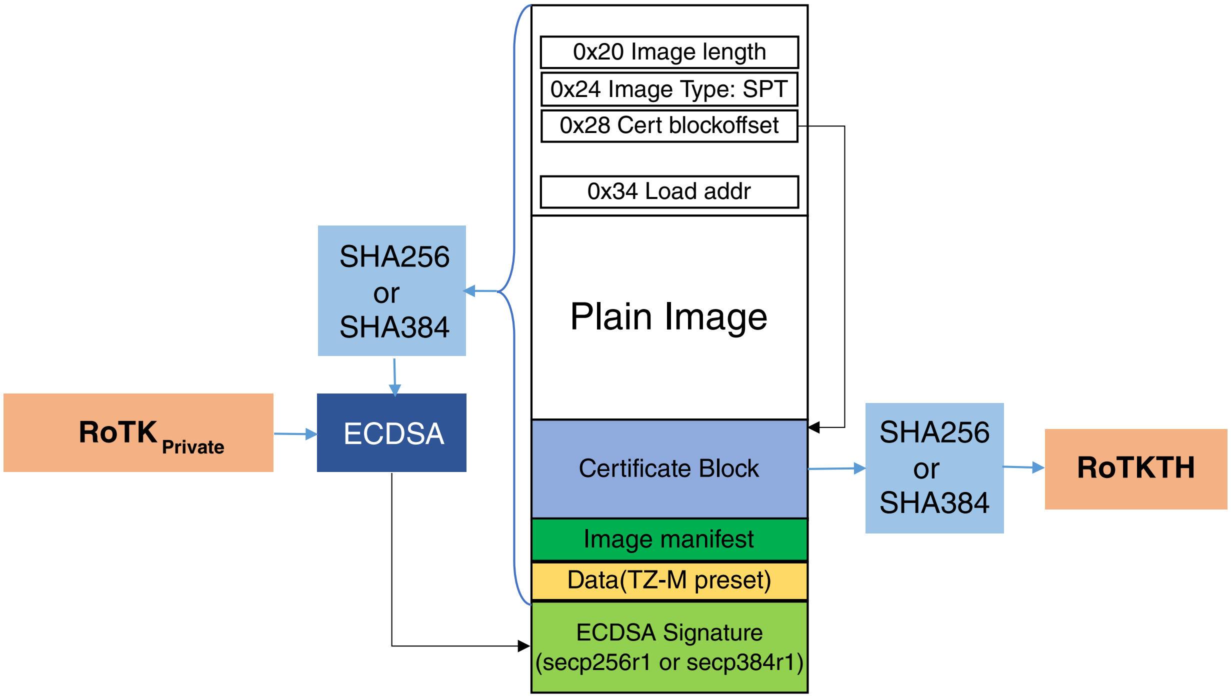


Figure 11. nboot signed image format

Vector table is standard Arm cortex M33 vector table. Boot image is reusing three reserved words in it for secure boot purpose:

Table 10. Boot image vector table

Offset	Field	Size (bytes)	Description
0x00	Arm cortex M33 specific	32	Arm specific data

Table continues on the next page...

Table 10. Boot image vector table (continued)

Offset	Field	Size (bytes)	Description
0x20	imageTotalLength	4	Total image length in bytes, including signatures etc.
0x24	imageType	4	Information about image type, XIP_PLAIN_CRC/ XIP_PLAIN_SIGNED and TZM settings (containing TZ-M preset data or not)
0x28	certificateBlockOffset	4	Offset from start of header block to the certificate block. This allows the signed image verification code to verify the signature over the header block. In case of XIP_PLAIN_CRC image type, offset contains CRC32 checksum of the whole image excluding offset 0x28.
0x34	Image Link Address	4	Image execution address or address of exception vector table

Table 11. Details of imageType (word at offset 0x24)

Bit 31 -- bit 16	Reserved	Shall be set to 0.
Bit 15 - bit 14	Reserved	Shall be set to 0.
Bit 13	TZ-M Preset	0: No TZ-M peripherals preset. 1: TZ-M peripherals preset. The TZ-M related peripherals are configured by bootloader based on data stored in extended header.
Bit 12 -- bit 8	Reserved	Shall be set to 0.
Bit 7 -- bit 0	Image Type	0x0: plain image 0x4: XIP plain image signed 0x5: XIP plain image with CRC 0x6: SB3 firmware 0x7: SB3 ELE firmware signed by NXP 0x8: XIP plain image signed by NXP 0x44: NBU XIP plain image signed 0x48: NBU XIP plain image signed by NXP Other values are reserved.

For calculation of digital signature of the signed boot image using ECDSA algorithm, SHA256 or SHA384 digest of complete image including additional data (certificate block, image manifest, trust-zone preset data) according configuration in certificate block is used as input to ECDSA algorithm. Certificate block used in signed boot image is the same as Certificate block used in SB3.1 firmware update container and will be described in detail in following chapter about SB3.1 firmware update container.

The boot image header is compatible with SB3.1 firmware update container.

Integer value starting on offset 0x28 in image vector table contains number of bytes to start the certificate block counting from the image beginning. Certificate block is directly followed by image manifest and optional trust-zone preset data (TRDC configuration).

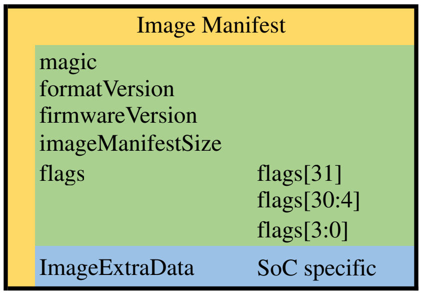


Figure 12. Image Manifest

Table 12. Image Manifest

Image Manifest	Offset	Data Type (little endian)	Description
magic	0x00u	uint32_t	Fixed 4-byte string "imgm" without the trailing NULL, 0x6D676D69u.
formatVersion	0x04u	uint32_t	Fixed value "1.0", 0x00010000u, major = 1, minor = 0.
firmwareVersion	0x08u	uint32_t	Value compared with monotonous counter stored in device fuses (CM33_S_VER_CNT_VIRTUAL). If value is lower than value in fuses, image is rejected and not started during secure boot (rollback protection).
imageManifestSize	0x0Cu	uint32_t	Total size of image manifest in bytes including image extra data.
flags	0x10u	uint32_t	**flags[bit 31]** - Set to 1 if precalculated image digest included after image signature. Used for concurrent execution of ECDSA verify and hash calculation. **flags[bit 30:4]** - Reserved for future use. **flags[bit 3:0]** - Included hash type: SHA256 = 0x1, SHA384 =0x2, 0x0 if no digest attached.
ImageExtraData	0x14u	uint8_t[size]	Optional SoC specific section, such as Trust-Zone preset data. Variable size depending on content, data must be aligned to 4 bytes.

4.3.2.1.1 Image signature verification

The sequence for verifying the signature of a signed image is as follows.

1. Validate pointer to certificate block at offset 0x28 in the image's vector table.
2. Validate certificate block header.
3. Validate RoTK$_{public}$ in the certificate block matches the RoTKTH in fuses.
4. Extract full RoTK$_{public}$ from certificate block and install in a key slot in the ELE. If ISK is not used, 5 and 6 are skipped.
5. Validate image signing certificate block, including that RoTKpublic is part of RoTKTH value in fuses and corresponding RoTKprivate ECDSA signed the ISK block.
6. Extract full image signing public key and install in a key slot in the ELE.
7. Compute SHA256 or SHA384 hash of the entire image contents, including the certificate block. Verify ECDSA signature using image signing public key that was previously extracted. Hash algorithm is selected based on configuration in certificate block. Refer to Certificate block for more details.

8. Report success or failure to caller.

The block diagram shows the image signature verification flow and relationship to operations performed by the ELE on behalf of the CM33 ROM.

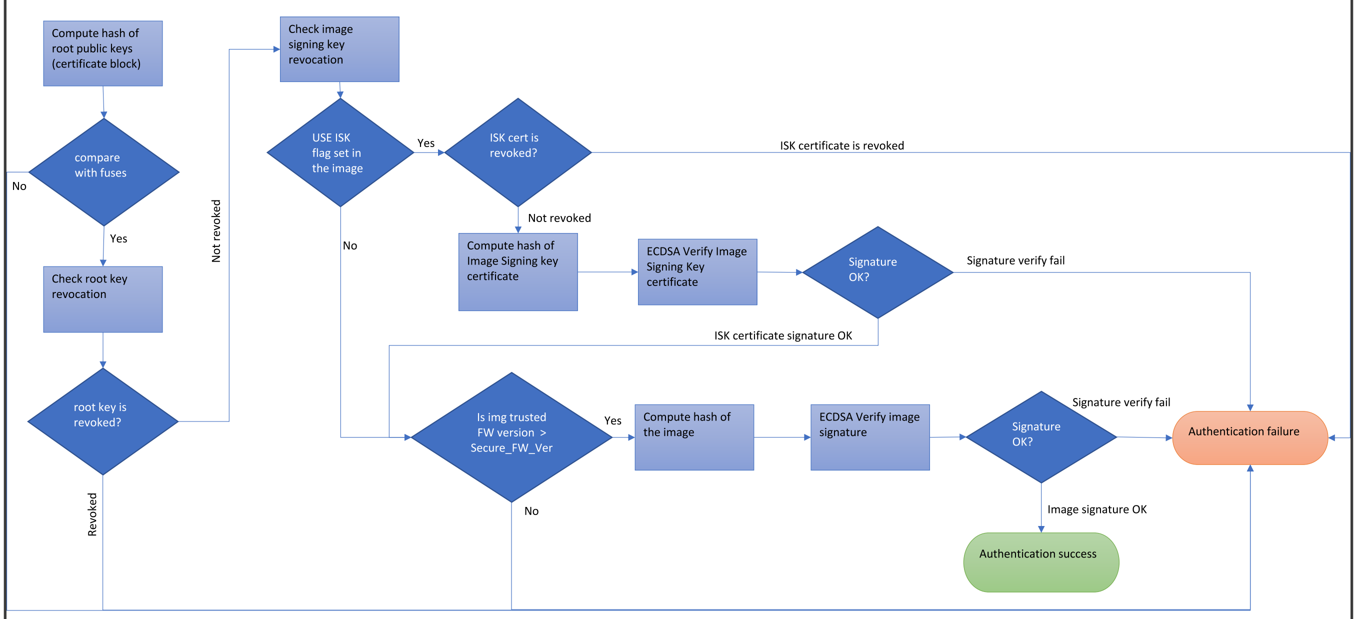

Compute hash of root public keys (certificate block)
compare with fuses
No
Yes
Check root key revocation
root key is revoked?
Revoked
Not revoked
Check image signing key revocation
USE ISK flag set in the image
Yes
No
ISK cert is revoked?
ISK certificate is revoked
Not revoked
Compute hash of Image Signing key certificate
ECDSA Verify Image Signing Key certificate
Signature OK?
Signature verify fail
ISK certificate signature OK
Is img trusted FW version > Secure_FW_Ver
Yes
No
Compute hash of the image
ECDSA Verify image signature
Signature OK?
Signature verify fail
Authentication failure
Image signature OK
Authentication success

Figure 13. nboot image verification

4.3.2.1.2 Secure boot failure

The process for handling a failure to verify the signature of the boot image is controlled through some flags in IFR.

1. Send a "clear all keys" request to the ELE. (A full security violation would reset the device)
2. If a boot failure pin is specified in IFR, assert it.
3. Depending on secure boot failure option, either:
   - Enter infinite WFI loop.
   - Jump to ISP path for firmware update

4.3.2.2 SB3.1 firmware update container

The Secure Binary (SB) container brings secure and easy way to upload or update firmware in embedded device during either the manufacturing process or end-customer's device lifecycle.

The SB container in version 3.1 (SB3.1) uses latest cryptographical algorithms to ensure highest possible authenticity and confidentiality of carried firmware. The security level of SB3.1 is configurable and adding possibility to reach Commercial National Security Algorithm Suite (CNSA) level of security based on project performance/boot time vs security requirements. Authenticity of SB3.1 container is ensured by digital signature based on Elliptic Curve Cryptography (ECC) and confidentiality of SB3.1 container is ensured by use of Advance Encryption System (AES) in Cipher Block Chain (CBC) mode.

4.3.2.2.1 SB3.1 structure

SB3.1 is characterized as chain of blocks (figure SB3.1 structure), and each Block i contains hash digest of block i+1.

Besides the hash digest of Block 1, the block 0 also contains digital signature, which guarantee authenticity of hash digest of block 1 and thus the whole chain. By digital signature verification of Block 0 followed by gradual verification of all hashes in chain for all following blocks, authenticity of whole SB3.1 chain is verified. The last block in chain (Block N) contains only zeroes instead of hash digest value.

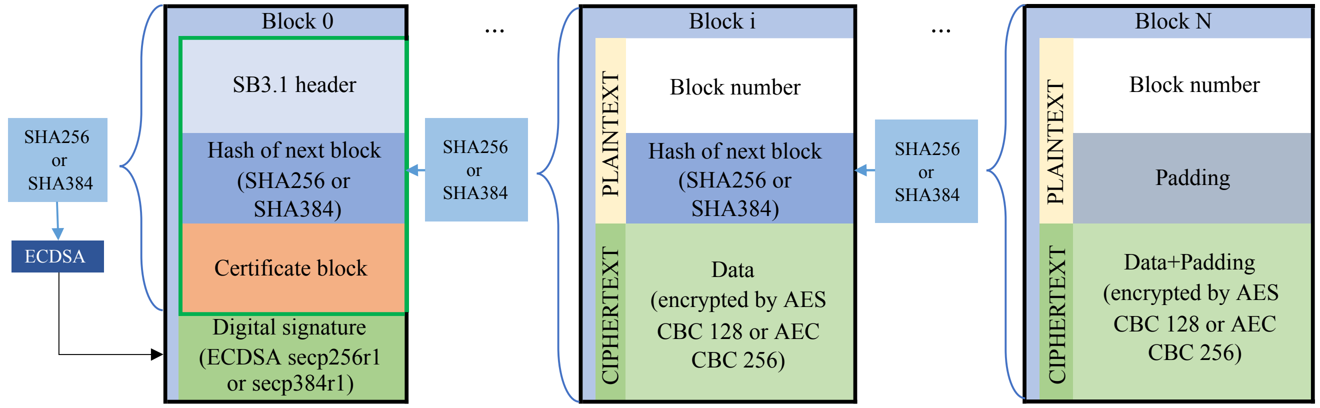

Block 0
SB3.1 header
Hash of next block (SHA256 or SHA384)
Certificate block
Digital signature (ECDSA secp256r1 or secp384r1)
SHA256 or SHA384
ECDSA
...
Block i
PLAINTEXT
Block number
Hash of next block (SHA256 or SHA384)
CIPHERTEXT
Data (encrypted by AES CBC 128 or AEC CBC 256)
SHA256 or SHA384
...
Block N
PLAINTEXT
Block number
Padding
CIPHERTEXT
Data+Padding (encrypted by AES CBC 128 or AEC CBC 256)
SHA256 or SHA384

Figure 14. SB3.1 structure

4.3.2.2.2 SB3.1 Block 0

Block 0, also known as SB3.1 manifest, is different from the rest of SB3.1 blocks. Block 0 (see full content of Block 0 in Figure 16) does not contain any firmware data, instead, it holds the configuration and other data needed for authentication process. It is compatible with signed boot image format, where SB3.1 header and Hash of the next block acts as image data, so the same function can be used for authentication of signed boot image and SB3.1 Block 0. The size of Block 0 is various and depends on the selected level of security.

Hash algorithm used for hash of next block calculation depends on the type of elliptic curve used for digital signature of Block 0. If secp256r1 is selected, then SHA256 is used, if secp384r1 is selected, then SHA384 is used.

The digital signature of Block 0 is calculated over whole Block 0 (green framed data in Figure 15 and Figure 16)) by use of SHA256 if secp256r1 is selected for ECDSA digital signature, or SHA384 if secp384r1 is selected for ECDSA digital signature.

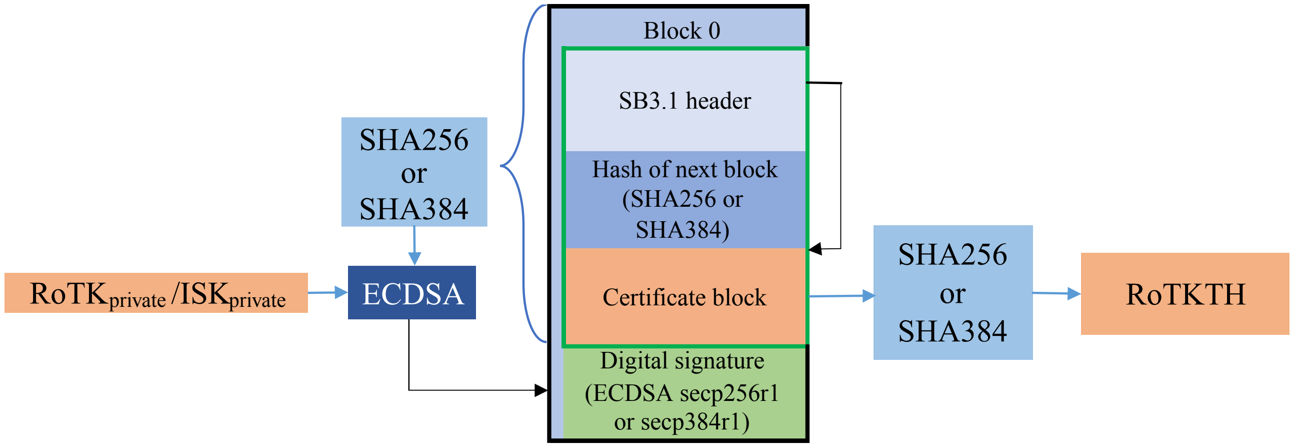

Block 0
SB3.1 header
Hash of next block (SHA256 or SHA384)
Certificate block
Digital signature (ECDSA secp256r1 or secp384r1)
SHA256 or SHA384
ECDSA
RoTK$_{private}$ /ISK$_{private}$
SHA256 or SHA384
RoTKTH

Figure 15. SB3.1 Block 0

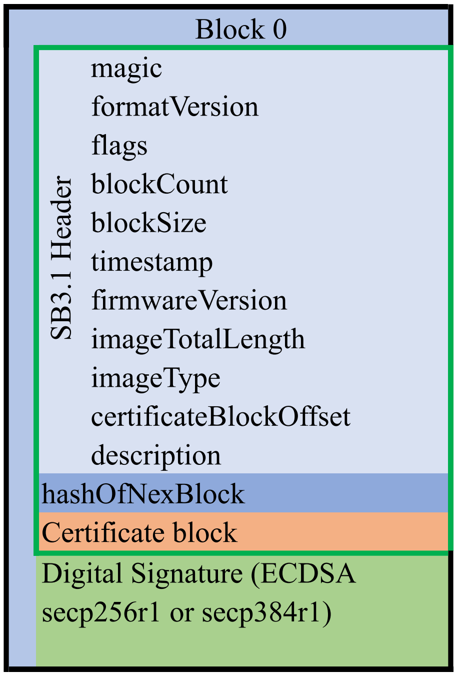


Figure 16. SB3.1 Block 0 full structure

- **SB3.1 Header** – object
  - **magic** – uint32_t, little endian

    Fixed 4-byte string "sbv3" without the trailing NULL, 0x33766273u.
  - **formatVersion** – uint32_t, little endian

    Fixed to "3.1", 0x00030001u, major = 3, minor = 1.
  - **flags** – uint32_t, little endian

    Not used, reserved for future use.
  - **blockCount** – uint32_t, little endian

    Total number of data blocks (not including the block 0).
  - **blockSize** – uint32_t, little endian

    Size in bytes of one data block. All data blocks have the same size.
  - **timestamp** – uint64_t, little endian

    Cryptographic nonce, e.g. number of seconds from 01/01/2000. Value used as one of the inputs into SB3.1 key derivation process.
  - **firmwareVersion** – uint32_t, little endian

    Version number of the included firmware.
  - **imageTotalLength** – uint32_t, little endian

    Total block 0 length in bytes, including block 0 signature.
  - **imageType** – uint32_t, little endian

    Image type, 0x06u for regular SB3.1, other values reserved.
  - **certificateBlockOffset** – uint32_t, little endian

Offset from start of block 0 to the certificate block. This allows the signed image verification code to verify the signature over the block 0.

— **description** – uint8_t[16]

Free text field for file description.

- **hashOfNexBlock** – uint8_t[32] or uint8_t[48]

  Hash is computed over whole Block 1 after encryption of data section. Size is 32 bytes (SHA256) or 48 bytes (SHA384). Hash algorithm is selected based on EC type used for signing of Block 0. Secp256r1 -> SHA256 or secp384r1 -> SHA384.

- **Certificate block** – uint8_t[variable size]

  Refer to Certificate block for details.

- **Digital Signature** – uint8_t[64] or uint8_t[96]

  The ECDSA signature of hash (SHA256 or SHA384) over the header + hash of next block + certificate block data. Hash algorithm is based on EC type used for signature (secp256r1 or secp384r1). Secp256r1 -> SHA256 or secp384r1 -> SHA384. Signature coordinates (r,s) stored in big-endian.

4.3.2.2.3 SB3.1 Block i (Data block)

Block i, also known as data block, contains data payload separated into fixed size data chunks. The data chunk size is currently set to 256 bytes. Data blocks are numbered from 1 to N. The total number of data blocks (N) in SB3.1 is corresponding to the size of payload divided by data chunk size. When payload is not aligned to block size, in last block (Block N) padding is added in form of zeros to have data chunk aligned to data chunk size. Hash of next block for last block is filled with zeroes. See full structure details in Table 2 - Block i (data block).

Each data chunk is encrypted by AES CBC 128 or AES CBC 256 algorithm using unique key per block. Keys are derived from SB3KDK key, which is external input to SB3.1 key derivation process.

SB3.1 key derivation process is following NIST SP 800-108 (Recommendation for Key Derivation Using Pseudorandom Functions) specification. Refer to SB3.1 key derivation process for more details.

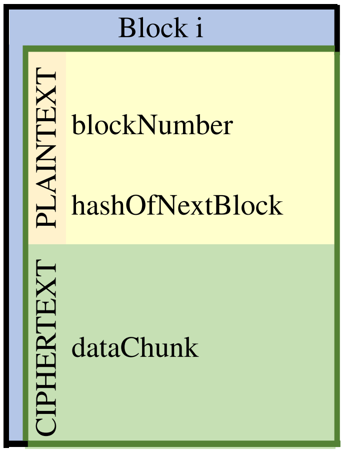


Figure 17. Block i

- **blockNumber** – uint32_t, little endian

  Number of current block, starting from 1 to N.

- **hashOfNextBlock** – uint8_t[32] or uint8_t[48]

  SHA256 or SHA384 digest of whole block i+1 (blockNumber || Next Block Hash[32]/[48] || dataChunk[blockSize]). Hash algorithm is selected based on EC type used for signing of Block 0. Secp256r1 -> SHA256 or secp384r1 -> SHA384.

- **dataChunk** – uint8_t[blockSize]

Payload data encrypted by aes_cbc128(FW_KBLK_128[i]) or aes_cbc256(FW_KBLK_256[i]). AES algorithm is selected based on EC type used for signing of Block 0. Secp256r1 -> aes_cbc128 or secp384r1 -> aes_cbc256.

4.3.2.2.4 SB3.1 key derivation process

SB3.1 key derivation process accords with NIST SP 800-108 (Recommendation for Key Derivation Using Pseudorandom Functions) specification. As pseudo random function (PRF) uses CMAC algorithm and key derivation function is running in counter mode, SB3.1 derivation function is named CMAC Key Derivation Function (CKDF).

SB3.1 Key derivation inputs 256-bit long symmetric key named SB3 Key Derivation Key (SB3KDK). After that, derive Firmware Key Derivation Key (FW_KDK) with CKDF function by use of SB3KDK and timestamp as part of derivation data. FW_KDK is then used as CKDF input for derivation of key for each data block named FW_KBLK(i), where block number is used as part derivation data. Detailed structure of derivation data is described later in the chapter.

The size of derived FW_KDK and FW_KBLK(i) is driven by EC type used for signing of Block 0. If secp256r1 is used, then FW_KDK and FW_KBLK(i) have 128 bits. If secp384r1 is used, then FW_KDK and FW_KBLK(i) have 256 bits.

Table 13. SB3.1 key derivation process

Key		Description
Key derivation key:	FW_KDK = CKDF(SB3KDK, timestamp)	Initial key derivation key is derived from SB3KDK
Key encryption/ decryption keys:	FW_KBLK(0) = N/A	Block 0 is not encrypted.
	FW_KBLK(1) = CKDF(FW_KDK, 0x1)	Encryption/decryption key for block 1.
	FW_KBLK(2) = CKDF(FW_KDK, 0x2)	Encryption/decryption key for block 2.
	…	…
	FW_KBLK(N) = CKDF(FW_KDK, N)	Encryption/decryption key for block N.

Pseudo code of used key derivation algorithm (CKDF):

For i = 1 to n, do

1. K(i) := PRF (KI, Label || Context || [L]2 || [i]2)
2. result(i) := result(i-1) || K(i).

Table 14. Key derivation data

Input	Data type (length)	FW_KDK	FW_KBLK(i)
K_I	uint8_t[32]	Symmetric 256bit key -> PCK.	FW_KDK
Label	uint8_t[12] (96bits)	Timestamp (uint64_t) value from SB3.1 Block 0 header in little endian with added zero padding to 96 bits. Example: "A170AD270000000000000000".	BlockNumber (uint32_t) value from SB3.1 Block i header in little endian with added zero padding to 96 bits. Example for block 14: "0E0000000000000000000000".
Context	uint8_t[12] (96bits)	If FW_KDK is 128 bits, value is fixed to "0000000000000000c0010020". If FW_KDK is 256 bits, value is fixed to "0000000000000000C0010021".	If FW_KBLK(i) is 128 bits, it is fixed to "0000000000000000C0100020". If FW_KBLK(i) is 256 bits, it is fixed to "0000000000000000C0100021".
L	uint32_t - Big endian	Value is corresponding to derived key size. If FW_KDK is 128 bits, it is fixed to	Value is corresponding with derived key size. If FW_KBLK(i) is 128 bits, it is fixed

Table continues on the next page...

Table 14. Key derivation data (continued)

Input	Data type (length)	FW_KDK	FW_KBLK(i)
		"00000080". If FW_KDK is 256 bits, it is fixed to "00000100".	to "00000080". If FW_KBLK(i) is 256 bits, it is fixed to "00000100".
i	uint32_t - Big endian	Counter. If FW_KDK is 128 bits, only one iteration is executed with value "00000001". If FW_KDK is 256 bits, two iterations are done with value "00000001" followed by value "00000002".	Counter. If FW_KBLK(i) is 128 bits, only one iteration is executed with value "00000001". If FW_KBLK(i) is 256 bits, two iterations are done with value "00000001" followed by value "00000002".

Key derivation example:

1. FW_KDK 256bit derivation
   - SB3KDK: "24e517d4ac417737235b6efc9afced8224e517d4ac417737235b6efc9afced82"
   - Data1: "8b71ad2700000000000000000000000000000000c00100210000010000000001"
   - Data2: "8b71ad2700000000000000000000000000000000c00100210000010000000002"
   - Res1 = PRF(PCK, Data1) = "68fd9ef140290488eca5736aa9f4b4a5"
   - Res2 = PRF(PCK, Data2) = "cf437c8618809047ec1d46f70523481a"
2. FW_KBLK(3) 256bit derivation
   - FW_KDK: "68fd9ef140290488eca5736aa9f4b4a5cf437c8618809047ec1d46f70523481a"
   - Data1: "0300000000000000000000000000000000000000c01000210000010000000001"
   - Data2: "0300000000000000000000000000000000000000c01000210000010000000002"
   - Res1 = PRF(FW_KDK, Data1) = "4b2afc98b4ca03fc0de090be76d3beb2"
   - Res2 = PRF(FW_KDK, Data2) = "729fb4b3149b3ea05f414a2dd0a193ce"
   - FW_KBLK(3): "4b2afc98b4ca03fc0de090be76d3beb2729fb4b3149b3ea05f414a2dd0a193ce"

4.3.2.2.5 Certificate block

Certificate block is the most important part of SB3.1 and signed boot image. The Certificate block used in SB3.1 and corresponding signed boot image has version 2.1 which is only for elliptic curve cryptography (ECC). Certificate block for generation requires 1 to 4 key pairs, based on secp256r1 or secp384r1, EC type can't be mixed (only secp256r1 or secp384r1). Hash digest of public key(s) is stored in Root of Trust Key Table (RoTKT), size of Root of Trust Key Hash (RoTKH) record depends on EC type. If secp256r1 is provided, then SHA256 is used, if secp384r1 is provided, then SHA384 is used.

The hash algorithm rule, which can be applied on all ECDSA signatures in current certificate block design, is that if secp256r1 curve is used for signing, then SHA256 digest of signed data should be used as input; if secp384r1 curve is used for signing, then SHA384 digest of signed data should be used as input.

Hash digest (SHA256 or SHA384 depending on root certificate EC type) of RoTKH records is stored in non-volatile memory of the device (Fuse, PFR), which locks the possibility of later change of Root certificate and creates Root of Trust i the device. The final hash of hashes is named Root of Trust Key Table Hash (RoTKTH). If only one root certificate is used, then RoTKTH is calculated as hash (SHA256 or SHA384 depending on root certificate EC type) directly from public key.

Next important input is Image Signing Key (ISK). If ISK is not used, then ISK certificate section is removed from Certificate block and one from provided root certificates is used for signature of whole Block 0 or boot image. If ISK certificate is provided, then one from provided root certificates is used for signing of ISK public key and ISK private key then signing whole Block 0 or boot image. ISK EC type can have only same or lower size in compare to root certificate. This means if RoTKs are based on secp256r1, then ISK can be only secp256r1, if root certificates are based on secp384r1, then secp256r1 or sec384r1 can be used for ISK. The private part of ISK key pair needs to be provided as external input for ECDSA signature of boot image or Block 0.

The signature of ISK certificate is done by hash calculation over RoTRecord and iskCertificate (green framed data in figure Certificate block). The hash algorithm is selected based on EC type of RoTK, if secp256r1 is provided, then SHA256 is used, if secp384r1 is provided, then SHA384 is used. The private part of selected root key pair needs to be provided as external input for ECDSA signature.

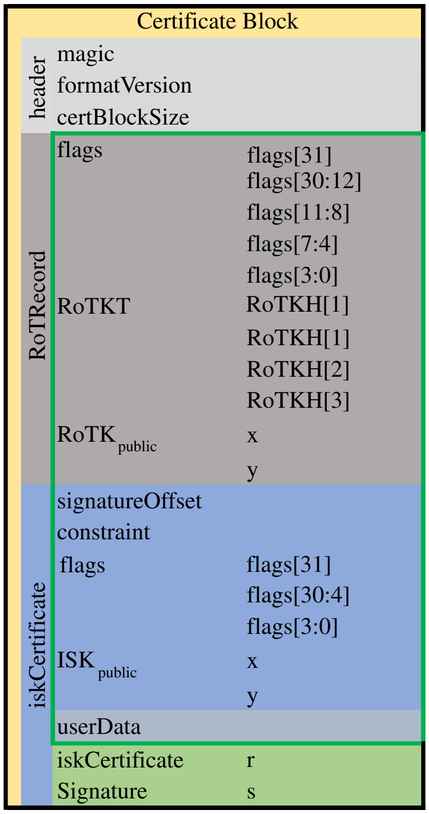


Figure 18. Certificate Block

- **SB3.1 Header** – object
  - **magic** – uint32_t, little endian

    Fixed 4-byte string "chdr" without trailing NULL, 0x72646863.
  - **formatVersion** – uint32_t, little endian

    Fixed to "2.1", 0x00020001, major = 2, minor = 1.
  - **certBlockSize** – uint32_t, little endian

    Total size of Certificate block in bytes.
- **RoTRecord** – object
  - **flags** – uint32_t, little endian
    - **flags[bit 31]**: NoCA flag, if set to 0, used RoTK acts as Certificate Authority and is used to sign ISK certificate, not full image. If set to 1, used RoTK does not act as Certificate Authority and signs directly the full image or SB3 Block0. If NoCa flag is set to 1, then iskCertificate section is not present in certificate block.
    - **flags[bits 30:12]**: Reserved for future use.
    - **flags[bits 7:4]**: Used root cert number [0-3] (specify root cert used to ISK/image signature), used only when more than 1 root certificate is specified.
    - **flags[bits 3:0]**: Type of root certificate, secp256r1 = 0x1u or secp384r1 = 0x2u, other values are reserved.

- **RoTKT** – object

  Root of Trust keys Table, optional object (in case when only one RoTK is specified for device, RoTKT is not present, 1 to 4 RoTK can be specified for device), so if RoTKT is present in Certificate Block, then RoTKT table contains 2 to 4 RoTKH records.

  - **RoTKH[0-3]** – uint8_t[32] or uint8_t[48]

    Root of Trust Key Hash is SHA256 or SHA384 of RoTKpublic. Hash algorithm is selected based on RoTK EC type (secp256r1 -> SHA256 or secp384r1 -> SHA384). Same RoTKs and RoTKTH values are shared between debug authentication, SB3.1 firmware updates container and signed boot image.

- **RoTK$_{public}$** – uint8_t[64] or uint8_t[96]

  X and Y coordinates of public key of selected RoTK, which will be used for signing and verification of ISK certificate or SB3.1 Block 0, field size depends on EC type (secp256r1 or secp384r1).

- **iskCertificate** – object

  Optional object, If ISK is not used (driven by CA flag), the whole iskCertificate section is missing.

  - **signatureOffset** – uint32_t, little endian

    Offset in bytes to ISK certificate signature from the beginning of iskCertificate object.

  - **constraint** – uint32_t, little endian

    Constraint = certificate version, compared with monotonic counter in fuses.

  - **flags** – uint32_t, little endian

    - **flags[bit 31]**: User data flag, if set to 1, user data are included.
    - **flags[bits 30:4]**: Reserved for future use.
    - **flags[3:0]**: Type of ISK certificate, secp256r1 = 0x1u or secp384r1 = 0x2u, other values are reserved.

  - **ISK$_{public}$** – uint8_t[64] or uint8_t[96]

    ISK public key. If root certificate is secp256r1, ISK can be also only secp256r1, if root is secp384r1, then ISK can be secp256r1 or secp384r1. Public key stored in big-endian

  - **userData** – uint8_t[??]

    Optional, variable size. Can contain user data, e.g. EC public key, which is signed by ISK signature. Maximal size on this device is limited to 96 bytes (EC384 key pair). Data should be aligned to 4 bytes.

  - **iskCertificate Signature** – uint8_t[64] or uint8_t[96]

    ECDSA signature of SHA256 or SHA384 of green framed data on certificate block figure based on EC type of RoTK (secp256r1 or secp384r1). Secp256r1 -> SHA256 or secp384r1 -> SHA384. Signature stored in big-endian.

4.3.2.2.6 SB3.1 data chunk

The data area of all blocks following the header block is a contiguous sequence of bytes, called the payload, which is divided into equal-sized chunks. Each chunk is placed into its own block.

For a given block size, the data chunk size is:

chunkSize = blockSize – blockOverhead

blockSize = 256 bytes.

blockOverhead = 4bytes for block number (uint32_t) + SHA256 or SHA384 digest of next block (32 or 48 bytes)

Data stream can be divided into multiple sections. Each section is starting by section header:

```
struct section_header {
   uint32_t sectionUid; //0x1u
   uint32_t sectionType; //0x1u
   uint32_t length;
   uint32_t _pad; //zeores
};
```

The length field of the section header can be used to skip over the section when searching for a given section.

Data range section contains one or more ranges of data to be loaded and the target addresses. Only one data range section is supported on this device.

Data range section consists of one or more ranges, where each range starting with a header:

```
struct range_header {
uint32_t tag
     uint32_t startAddress;
     uint32_t length;
     uint32_t cmd;
};
```

The startAddress and length specify the target memory address and data range to be written to the memory address. Tag carries a magic number 0x55aaaa55 as an identifier of data range header. Cmd field is an enum of various actions (commands) to be performed with the data range. Some commands contain also command specific extended header of 16 bytes added right after range header. More details are available in description of each command.

List of supported commands:

```
enum CommandType {
   kSB3_COMMAND_erase = 0x1u,
   kSB3_COMMAND_load = 0x2u,
   kSB3_COMMAND_execute = 0x3u,
   kSB3_COMMAND_programFuses = 0x5u,
   kSB3_COMMAND_programIfr = 0x6u,
   kSB3_COMMAND_fillMemory = 0xCu,
   kSB3_COMMAND_fwVersionCheck = 0xDu,
};
```

- **1–Erase**

  Performs a flash erase of the given address range. The erase will be rounded up to the sector size.

  ```
  struct range_header {
        uint32_t tag 0x55aaaa55
        uint32_t startAddress;
        uint32_t length;
        uint32_t cmd; //0x1u
  };
  Followed by:
  struct range_header_memory_data {
        uint32_t memoryId;
        uint32_t _ pad0; //zeros
        uint32_t _ pad1; //zeros
        uint32_t _pad2; //zeros
  };
  ```

- **2–Load**

  If set, then the data to write immediately follows the range header. Padding must be appended after the data such that the start of the next range or section header is 16-byte aligned. The length field contains the actual data length, not the aligned length.

  ```
  struct range_header {
       uint32_t tag 0x55aaaa55
       uint32_t startAddress;
  ```

```
     uint32_t length;
     uint32_t cmd;//0x2u
};
Followed by:
struct range_header_memory_data {
     uint32_t memoryId; 782514
     uint32_t _ pad0; //zeros
     uint32_t _ pad1; //zeros
     uint32_t _pad2; //zeros
};
```

- 3–Execute

  Perform authentication and jump to the code immediately after receive-sb operation is complete. So, test code can be loaded and executed out of RAM or flash. Command requires startAddress where signed/plain image is loaded.

- 4–Call

  The startAddress will be the address to jump, however, the state machine expects a return to the next statement to continue processing the sb file.

- 5–programFuses

  The startAddress will be the address of fuse register, length will be number of fuse words to program. The data to write to the fuse registers will immediately follow the header.

- 6–programIFR

  The startAddress will be the address into the IFR region, length will be in number of bytes to write to IFR region. The data to write to IFR region at the given address will immediately follow the header.

- 12–fillMemory

  The startAddress specifies the address of memory region to be filled with pattern and size as parameters of a command.

```
struct range_header {
  uint32_t tag 0x55aaaa55
  uint32_t startAddress;
  uint32_t length; // number of repeats of pattern
  uint32_t cmd; //0xCu
};
Followed by:
struct fill_command_data {
  uint32_t pattern;
  uint32_t memoryId;
  uint32_t _pad0; //zeros
  uint32_t _pad1; //zeros
};
```

- 13–fwVersionCheck

  Check whether the FW version value specified in command for specific counterId is acceptable. FW version value in command must be greater than that programmed in fuses to be acceptable else rollback will be detected.

```
struct fw_version_check_range_header {
const uint32_t tag = NBOOT_RANGE_SECTION_TAG;
      uint32_t value; //value to compare with counter
      uint32_t counterId;
      uint32_t cmd; //0xDu
};
enum counterId {
             kNBOOT_CNT_none = 0x0u,
             kNBOOT_CNT_nonsecure = 0x1u,
             kNBOOT_CNT_secure = 0x2u,
             kNBOOT_CNT_radio = 0x3u,
             kNBOOT_CNT_snt = 0x4u,
             kNBOOT_CNT_bootloader = 0x5u,
  };
```

It is an error to have a cmd field with a value of 0.

Example of payload

- section_header
  - sectionUid = 0x0000_0001
  - sectionType = 0x0000_0001 (Data range section)
  - length = n
- range_header
  - tag = 0x55aaaa55
  - startAddress = 0x0000_0000
  - length = 8192
  - cmd = 0x1 (erase)
- range_header_memory_data
  - memory_id = 0x0
  - _pad0 = 0x0
  - _pad1 = 0x0
  - _pad2 = 0x0
- range_header
  - tag = 0x55aaaa55
  - startAddress = 0x0000_0000
  - length = 8192
  - cmd = 0x2 (load
- range_header_memory_data
  - memory_id = 0x0
  - _pad0 = 0x0
  - _pad1 = 0x0
  - _pad2 = 0x0
- (8192 bytes of data)
- range_header
  - tag = 0x55aaaa55
  - startAddress = 0x0200_0100
  - length = 16
  - cmd = 0x2 (load
- range_header_memory_data
  - memory_id = 0x0

  - _pad0 = 0x0
  - _pad1 = 0x0
  - _pad2 = 0x0
- (16 bytes of data)
- range_header
  - tag = 0x55aaaa55
  - startAddress = 0x0000_0000
  - length = 0
  - cmd = 0x3 (execute)
- range_header
  - tag = 0x55aaaa55
  - startAddress = 0x0300_0100
  - length = 16 (16 fuse registers of 32-bit each)
  - Cmd = 0x5 (programFuse)
- (16 * 4 bytes of data)
- range_header
  - Tag = 0x55aaaa55
  - startAddress = 0x0010_0100
  - length = 16 bytes
  - Cmd = 0x6 (programIFR)
- (16 * 4 bytes of data)
- range_header
  - Tag = 0x55aaaa55
  - startAddress = 0x25
  - length = 0xFF words
  - Cmd = 0xC (fillMemory)
- fill_command_data
  - pattern = 0x89ABCDEF
  - memoryId = 0x0
  - _pad0 = 0x0
  - _pad1 = 0x0
- fw_version_check_range_header
  - tag = 0x55aaaa55
  - value = 0x3 (FW version value to be checked)
  - counterId = 0x1 (nonsecure), 0x2 (secure), 0x3 (radio), 0x4 (s200), 0x5 and above (reserved)
  - cmd = 0xD (checkFwVersion)

Above example shows different kinds of range headers within a section.

4.3.2.2.7 SB3.1 processing

The following figure shows SB3.1 processing flow.

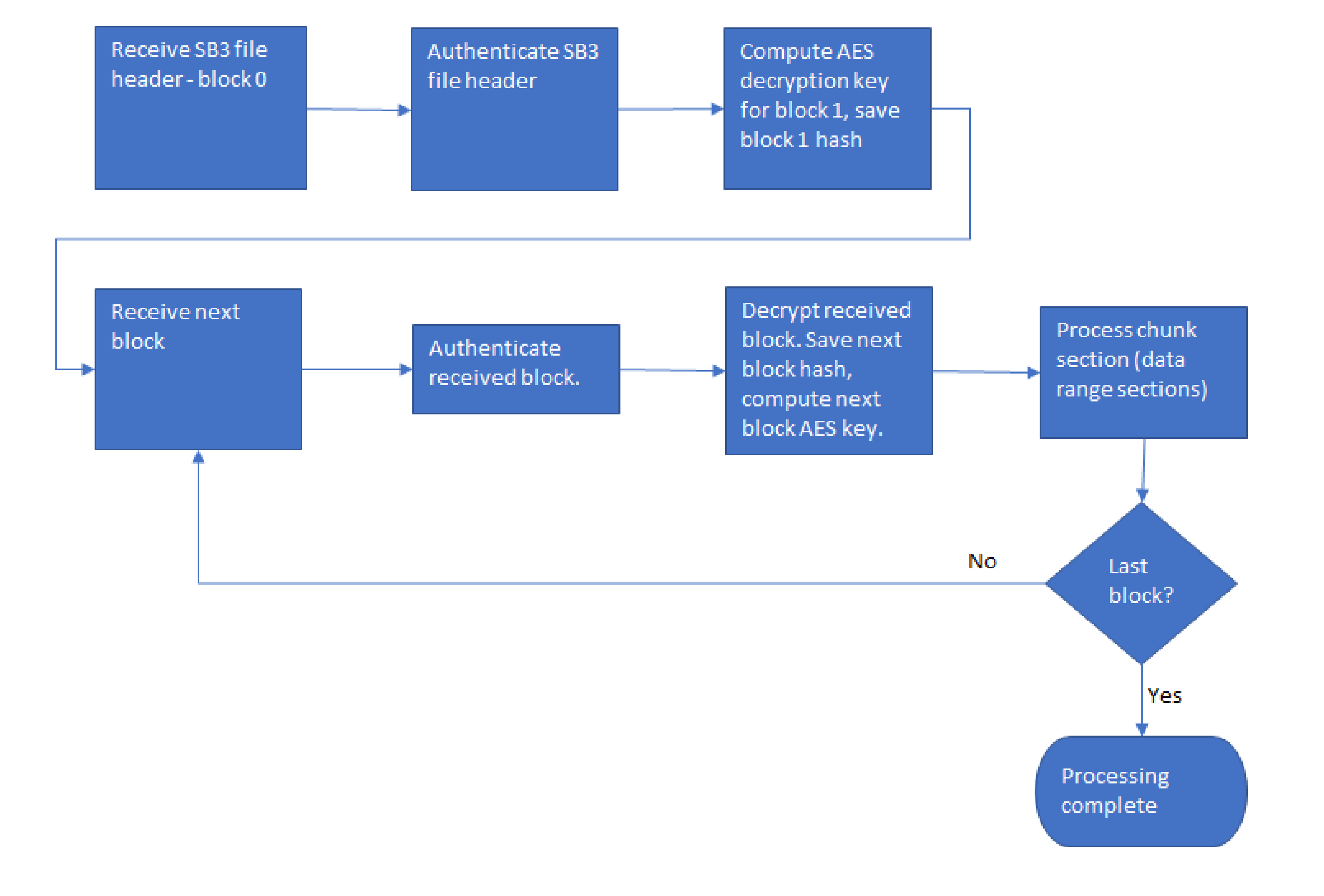

Receive SB3 file header - block 0
Authenticate SB3 file header
Compute AES decryption key for block 1, save block 1 hash
Receive next block
Authenticate received block.
Decrypt received block. Save next block hash, compute next block AES key.
Process chunk section (data range sections)
Last block?
No
Yes
Processing complete

Figure 19. SB3.1 processing flow

Refer to Image signature verification for more details about SB3.1 Block0 (header) authentication step.

4.3.3 NPX configuration

To protect and enhance security/confidentiality of application code in flash memory against semi-invasive attacks this device will deploy a mechanism to store all flash contents encrypted, that is transparent to the developer and the M33 Cortex platform.

4.3.3.1 Configure NPX

To enable the NPX feature, there are two parts in User IFR0 that will need to be configured. One is the options to enable NPX. The second is to configure the NPX regions definitions. Here is the NPX regions definition structure:

The NPX region descriptors simply consist of start and end addresses that are 512-byte aligned. The configuration structure closely mirrors the register settings.

```
#define NPX_COUNT_MAX        (4u)
#define NPX_VALID_ENTRY        (0x3824u)
struct npx_regions {
       uint32_t valid;
       uint32_t count;
       struct {
              uint32_t valid;
```

```
            uint32_t start;
            uint32_t end;
    } regions[4]; };
```

The maximum number of regions is 4. However, an NPX instantiation may only support 2 regions.

Bits [8:0] of the region start address must be 9'h0.

Bits [8:1] of the region end address must be 8'hFF. Bit 0 of the region end address is the Valid bit that enables the region.

For example, if user wants to restore the NPX for normal boot only, enable both encryption and decryption, but do not want to lock it, then here is the configuration for User Rom IFR boot option:

Table 15. Example configuration of user ROM IFR boot option

IFR0 address	Byte location (00 01 … 0F)
0x02000_0010	E6 FF FF FF FF FF FF FF 24 38 00 00 04 00 00 00
0x02000_0020	24 38 00 00 00 08 00 00 FF 09 00 00 24 38 00 00
0x02000_0030	00 0A 00 00 FF 0B 00 00 24 38 00 00 00 0C 00 00
0x02000_0040	FF 27 00 00 24 38 00 00 00 48 00 00 FF 67 00 00

It will configure and enable NPX for the following flash memory range:

0x0000_0800 – 0x0000_09FF

0x0000_0A00 – 0x0000_0BFF

0x0000_0C00 – 0x0000_27FF

0x0000_4800 – 0x0000_67FF

4.3.4 ROM Trustzone preset data support

4.3.4.1 TrustZone preset data

ROM provides support for TrustZone data configuration during boot process. The TrustZone preset data includes:

- VTOR, VTOR_NS, NVIC_ITNS0, NVIC_ITNS1, NSACR, CPPWR, CPACR core registers
- AIRCR.SYSRESETREQ, AIRCR.BFHFNMINS, AIRCR.PRIS, SCR.SLEEPDEEPS and SHCSR.SECUREFAULTENA bits
- Secure MPU
- Non-secure MPU
- SAU
- TRDC controller
- GPIO secure/non-secure assignment

If the TrustZone preset is enabled, the ROM, after image validation, configures all TrustZone related registers by data, provided at the end of the image. If corresponding lock bits are set, the registers are also locked, so any further registers modification is not possible until next reset.

This feature increases robustness of the user application since the user application jumps into pre-configured TrustZone environment and it doesn’t need to contain any TrustZone configuration code.

4.3.4.1.1 Master boot image with TrustZone preset data

The information whether image file contains TrustZone configuration data or not is defined in the vector section of the image header at offset 0x24, bit 13 (TZM_PRESET). The position of preset data block in image can be seen in Signed image structure.

Bit 13	TZ-M Preset	0: Trustzone preset data not present. 1: Trustzone preset data present.

4.3.4.1.2 TrustZone preset data structure

The TrustZone preset data structure is defined by following C structure:

```
typedef struct
{
    uint32_t tzm_magic; /*!< It contains four letters "TZ-M" to identify start of block */

    uint32_t tzm_control; /*!< It contains info, which data are initialized */

    uint32_t cm33_vtor_addr;    /*!< CM33 Secure vector table address */
    uint32_t cm33_vtor_ns_addr; /*!< CM33 Non-secure vector table address */
    uint32_t cm33_nvic_itns0;   /*!< CM33 Interrupt target non-secure register 0 */
    uint32_t cm33_nvic_itns1;   /*!< CM33 Interrupt target non-secure register 1 */
    uint32_t cm33_nvic_itns2;   /*!< CM33 Interrupt target non-secure register 2 */
    uint32_t cm33_misc_ctrl;    /*!< Miscellaneous CM33 settings:
                                     AIRCR.SYSRESETREQ
                                     AIRCR.BFHFNMINS
                                     AIRCR.PRIS
                                     SCR.SLEEPDEEPS
                                     SHCSR.SECUREFAULTENA */
    uint32_t cm33_nsacr;        /*!< CM33 Non-secure Access Control Register */
    uint32_t cm33_cppwr;        /*!< CM33 Coprocessor Power Control Register */
    uint32_t cm33_cpacr;        /*!< CM33 Coprocessor Access Control Register */

    /* SECTION 1 - START */
    uint32_t cm33_mpu_ctrl;  /*!< MPU Control Register.*/
    uint32_t cm33_mpu_mair0; /*!< MPU Memory Attribute Indirection Register 0 */
    uint32_t cm33_mpu_mair1; /*!< MPU Memory Attribute Indirection Register 1 */
    uint32_t cm33_mpu_rbar0; /*!< MPU Region 0 Base Address Register */
    uint32_t cm33_mpu_rlar0; /*!< MPU Region 0 Limit Address Register */
    uint32_t cm33_mpu_rbar1; /*!< MPU Region 1 Base Address Register */
    uint32_t cm33_mpu_rlar1; /*!< MPU Region 1 Limit Address Register */
    uint32_t cm33_mpu_rbar2; /*!< MPU Region 2 Base Address Register */
    uint32_t cm33_mpu_rlar2; /*!< MPU Region 2 Limit Address Register */
    uint32_t cm33_mpu_rbar3; /*!< MPU Region 3 Base Address Register */
    uint32_t cm33_mpu_rlar3; /*!< MPU Region 3 Limit Address Register */
    uint32_t cm33_mpu_rbar4; /*!< MPU Region 4 Base Address Register */
    uint32_t cm33_mpu_rlar4; /*!< MPU Region 4 Limit Address Register */
    uint32_t cm33_mpu_rbar5; /*!< MPU Region 5 Base Address Register */
    uint32_t cm33_mpu_rlar5; /*!< MPU Region 5 Limit Address Register */
    uint32_t cm33_mpu_rbar6; /*!< MPU Region 6 Base Address Register */
    uint32_t cm33_mpu_rlar6; /*!< MPU Region 6 Limit Address Register */
    uint32_t cm33_mpu_rbar7; /*!< MPU Region 7 Base Address Register */
    uint32_t cm33_mpu_rlar7; /*!< MPU Region 7 Limit Address Register */
    /* SECTION 1 - END */

    /* SECTION 2 - START */
    uint32_t cm33_mpu_ctrl_ns;  /*!< Non-secure MPU Control Register.*/
    uint32_t cm33_mpu_mair0_ns; /*!< Non-secure MPU Memory Attribute Indirection Register 0 */
    uint32_t cm33_mpu_mair1_ns; /*!< Non-secure MPU Memory Attribute Indirection Register 1 */
    uint32_t cm33_mpu_rbar0_ns; /*!< Non-secure MPU Region 0 Base Address Register */
    uint32_t cm33_mpu_rlar0_ns; /*!< Non-secure MPU Region 0 Limit Address Register */
    uint32_t cm33_mpu_rbar1_ns; /*!< Non-secure MPU Region 1 Base Address Register */
    uint32_t cm33_mpu_rlar1_ns; /*!< Non-secure MPU Region 1 Limit Address Register */
    uint32_t cm33_mpu_rbar2_ns; /*!< Non-secure MPU Region 2 Base Address Register */
    uint32_t cm33_mpu_rlar2_ns; /*!< Non-secure MPU Region 2 Limit Address Register */
    uint32_t cm33_mpu_rbar3_ns; /*!< Non-secure MPU Region 3 Base Address Register */
    uint32_t cm33_mpu_rlar3_ns; /*!< Non-secure MPU Region 3 Limit Address Register */
    uint32_t cm33_mpu_rbar4_ns; /*!< Non-secure MPU Region 4 Base Address Register */
    uint32_t cm33_mpu_rlar4_ns; /*!< Non-secure MPU Region 4 Limit Address Register */
    uint32_t cm33_mpu_rbar5_ns; /*!< Non-secure MPU Region 5 Base Address Register */
    uint32_t cm33_mpu_rlar5_ns; /*!< Non-secure MPU Region 5 Limit Address Register */
    uint32_t cm33_mpu_rbar6_ns; /*!< Non-secure MPU Region 6 Base Address Register */
    uint32_t cm33_mpu_rlar6_ns; /*!< Non-secure MPU Region 6 Limit Address Register */
    uint32_t cm33_mpu_rbar7_ns; /*!< Non-secure MPU Region 7 Base Address Register */
    uint32_t cm33_mpu_rlar7_ns; /*!< Non-secure MPU Region 7 Limit Address Register */
    /* SECTION 2 - END */

    /* SECTION 3 - START */
    uint32_t cm33_sau_ctrl;  /*!< SAU Control Register.*/
    uint32_t cm33_sau_rbar0; /*!< SAU Region 0 Base Address Register */
    uint32_t cm33_sau_rlar0; /*!< SAU Region 0 Limit Address Register */
    uint32_t cm33_sau_rbar1; /*!< SAU Region 1 Base Address Register */
    uint32_t cm33_sau_rlar1; /*!< SAU Region 1 Limit Address Register */
    uint32_t cm33_sau_rbar2; /*!< SAU Region 2 Base Address Register */
    uint32_t cm33_sau_rlar2; /*!< SAU Region 2 Limit Address Register */
    uint32_t cm33_sau_rbar3; /*!< SAU Region 3 Base Address Register */
    uint32_t cm33_sau_rlar3; /*!< SAU Region 3 Limit Address Register */
    uint32_t cm33_sau_rbar4; /*!< SAU Region 4 Base Address Register */
    uint32_t cm33_sau_rlar4; /*!< SAU Region 4 Limit Address Register */
    uint32_t cm33_sau_rbar5; /*!< SAU Region 5 Base Address Register */
    uint32_t cm33_sau_rlar5; /*!< SAU Region 5 Limit Address Register */
    uint32_t cm33_sau_rbar6; /*!< SAU Region 6 Base Address Register */
    uint32_t cm33_sau_rlar6; /*!< SAU Region 6 Limit Address Register */
    uint32_t cm33_sau_rbar7; /*!< SAU Region 7 Base Address Register */
    uint32_t cm33_sau_rlar7; /*!< SAU Region 7 Limit Address Register */
    /* SECTION 3 - END */

    uint32_t cr; /*!< TRDC Control Register */

    uint32_t idau_cr; /*!< TRDC IDAU Control Register */

    /* SHARED BY SECTION 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 - START */
    uint32_t mda_w0_0_dfmt0; /*!< Master 0 Domain Assignment Register*/
    uint32_t mda_w0_1_dfmt1; /*!< Master 1 Domain Assignment Register*/
    uint32_t mda_w0_2_dfmt1; /*!< Master 2 Domain Assignment Register*/
    uint32_t mda_w0_3_dfmt1; /*!< Master 3 Domain Assignment Register*/
    uint32_t mda_w0_4_dfmt1; /*!< Master 4 Domain Assignment Register*/
    /* SHARED BY SECTION 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 - END */

    /* SHARED BY SECTION 4, 5, 6 - START */
    uint32_t mbc0_memn_glbac0; /*!< Memory Block Checker 0, Global Access Control Register 0*/
    uint32_t mbc0_memn_glbac1; /*!< Memory Block Checker 0, Global Access Control Register 1*/
    uint32_t mbc0_memn_glbac2; /*!< Memory Block Checker 0, Global Access Control Register 2*/
    uint32_t mbc0_memn_glbac3; /*!< Memory Block Checker 0, Global Access Control Register 3*/
    uint32_t mbc0_memn_glbac4; /*!< Memory Block Checker 0, Global Access Control Register 4*/
    uint32_t mbc0_memn_glbac5; /*!< Memory Block Checker 0, Global Access Control Register 5*/
    uint32_t mbc0_memn_glbac6; /*!< Memory Block Checker 0, Global Access Control Register 6*/
    uint32_t mbc0_memn_glbac7; /*!< Memory Block Checker 0, Global Access Control Register 7*/
    /* SHARED BY SECTION 4, 5, 6 - START */

    /* SECTION 4 - START */
    uint32_t mbc0_0_mem0_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 0, Slave 0, Configuration
Word Register 0*/
    uint32_t mbc0_0_mem0_blk_cfg_w1; /*!< Memory Block Checker 0, Domain 0, Slave 0, Configuration
Word Register 1*/
    uint32_t mbc0_0_mem0_blk_cfg_w2; /*!< Memory Block Checker 0, Domain 0, Slave 0, Configuration
Word Register 2*/
    uint32_t mbc0_0_mem0_blk_cfg_w3; /*!< Memory Block Checker 0, Domain 0, Slave 0, Configuration
Word Register 3*/
    uint32_t mbc0_0_mem0_blk_cfg_w4; /*!< Memory Block Checker 0, Domain 0, Slave 0, Configuration
Word Register 4*/
    uint32_t mbc0_0_mem0_blk_cfg_w5; /*!< Memory Block Checker 0, Domain 0, Slave 0, Configuration
Word Register 5*/
    uint32_t mbc0_0_mem0_blk_cfg_w6; /*!< Memory Block Checker 0, Domain 0, Slave 0, Configuration
Word Register 6*/
    uint32_t mbc0_0_mem0_blk_cfg_w7; /*!< Memory Block Checker 0, Domain 0, Slave 0, Configuration
Word Register 7*/
    uint32_t mbc0_0_mem1_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 0, Slave 1, Configuration
Word Register 0*/
    uint32_t mbc0_0_mem2_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 0, Slave 2, Configuration
Word Register 0*/
    uint32_t mbc0_0_mem3_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 0, Slave 3, Configuration
Word Register 0*/
    uint32_t mbc0_0_mem3_blk_cfg_w1; /*!< Memory Block Checker 0, Domain 0, Slave 3, Configuration
Word Register 1*/
    /* SECTION 4 - END */

    /* SECTION 5 - START */
    uint32_t mbc0_1_mem0_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 1, Slave 0, Configuration
Word Register 0*/
    uint32_t mbc0_1_mem0_blk_cfg_w1; /*!< Memory Block Checker 0, Domain 1, Slave 0, Configuration
Word Register 1*/
    uint32_t mbc0_1_mem0_blk_cfg_w2; /*!< Memory Block Checker 0, Domain 1, Slave 0, Configuration
Word Register 2*/
    uint32_t mbc0_1_mem0_blk_cfg_w3; /*!< Memory Block Checker 0, Domain 1, Slave 0, Configuration
Word Register 3*/
    uint32_t mbc0_1_mem0_blk_cfg_w4; /*!< Memory Block Checker 0, Domain 1, Slave 0, Configuration
Word Register 4*/
    uint32_t mbc0_1_mem0_blk_cfg_w5; /*!< Memory Block Checker 0, Domain 1, Slave 0, Configuration
Word Register 5*/
    uint32_t mbc0_1_mem0_blk_cfg_w6; /*!< Memory Block Checker 0, Domain 1, Slave 0, Configuration
Word Register 6*/
    uint32_t mbc0_1_mem0_blk_cfg_w7; /*!< Memory Block Checker 0, Domain 1, Slave 0, Configuration
Word Register 7*/
    uint32_t mbc0_1_mem1_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 1, Slave 1, Configuration
Word Register 0*/
    uint32_t mbc0_1_mem2_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 1, Slave 2, Configuration
Word Register 0*/
    uint32_t mbc0_1_mem3_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 1, Slave 3, Configuration
Word Register 0*/
    uint32_t mbc0_1_mem3_blk_cfg_w1; /*!< Memory Block Checker 0, Domain 1, Slave 3, Configuration
Word Register 1*/
    /* SECTION 5 - END */

    /* SECTION 6 - START */
    uint32_t mbc0_2_mem0_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 2, Slave 0, Configuration
Word Register 0*/
```

```
    uint32_t mbc0_2_mem0_blk_cfg_w1; /*!< Memory Block Checker 0, Domain 2, Slave 0, Configuration
Word Register 1*/
    uint32_t mbc0_2_mem0_blk_cfg_w2; /*!< Memory Block Checker 0, Domain 2, Slave 0, Configuration
Word Register 2*/
    uint32_t mbc0_2_mem0_blk_cfg_w3; /*!< Memory Block Checker 0, Domain 2, Slave 0, Configuration
Word Register 3*/
    uint32_t mbc0_2_mem0_blk_cfg_w4; /*!< Memory Block Checker 0, Domain 2, Slave 0, Configuration
Word Register 4*/
    uint32_t mbc0_2_mem0_blk_cfg_w5; /*!< Memory Block Checker 0, Domain 2, Slave 0, Configuration
Word Register 5*/
    uint32_t mbc0_2_mem0_blk_cfg_w6; /*!< Memory Block Checker 0, Domain 2, Slave 0, Configuration
Word Register 6*/
    uint32_t mbc0_2_mem0_blk_cfg_w7; /*!< Memory Block Checker 0, Domain 2, Slave 0, Configuration
Word Register 7*/
    uint32_t mbc0_2_mem1_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 2, Slave 1, Configuration
Word Register 0*/
    uint32_t mbc0_2_mem2_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 2, Slave 2, Configuration
Word Register 0*/
    uint32_t mbc0_2_mem3_blk_cfg_w0; /*!< Memory Block Checker 0, Domain 2, Slave 3, Configuration
Word Register 0*/
    uint32_t mbc0_2_mem3_blk_cfg_w1; /*!< Memory Block Checker 0, Domain 2, Slave 3, Configuration
Word Register 1*/
    /* SECTION 6 - END */

    /* SHARED BY SECTION 7, 8, 9 - START */
    uint32_t mbc1_memn_glbac0; /*!< Memory Block Checker 1, Global Access Control Register 0*/
    uint32_t mbc1_memn_glbac1; /*!< Memory Block Checker 1, Global Access Control Register 1*/
    uint32_t mbc1_memn_glbac2; /*!< Memory Block Checker 1, Global Access Control Register 2*/
    uint32_t mbc1_memn_glbac3; /*!< Memory Block Checker 1, Global Access Control Register 3*/
    uint32_t mbc1_memn_glbac4; /*!< Memory Block Checker 1, Global Access Control Register 4*/
    uint32_t mbc1_memn_glbac5; /*!< Memory Block Checker 1, Global Access Control Register 5*/
    uint32_t mbc1_memn_glbac6; /*!< Memory Block Checker 1, Global Access Control Register 6*/
    uint32_t mbc1_memn_glbac7; /*!< Memory Block Checker 1, Global Access Control Register 7*/
    /* SHARED BY SECTION 7, 8, 9 - END */

    /* SECTION 7 - START */
    uint32_t mbc1_0_mem0_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 0, Slave 0, Configuration
Word Register 0*/
    uint32_t mbc1_0_mem1_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 0, Slave 1, Configuration
Word Register 0*/
    uint32_t mbc1_0_mem1_blk_cfg_w1; /*!< Memory Block Checker 1, Domain 0, Slave 1, Configuration
Word Register 1*/
    uint32_t mbc1_0_mem2_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 0, Slave 2, Configuration
Word Register 0*/
    uint32_t mbc1_0_mem2_blk_cfg_w1; /*!< Memory Block Checker 1, Domain 0, Slave 2, Configuration
Word Register 1*/
    uint32_t mbc1_0_mem3_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 0, Slave 3, Configuration
Word Register 0*/
    /* SECTION 7 - END */

    /* SECTION 8 - START */
    uint32_t mbc1_1_mem0_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 1, Slave 0, Configuration
Word Register 0*/
    uint32_t mbc1_1_mem1_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 1, Slave 1, Configuration
Word Register 0*/
    uint32_t mbc1_1_mem1_blk_cfg_w1; /*!< Memory Block Checker 1, Domain 1, Slave 1, Configuration
Word Register 1*/
    uint32_t mbc1_1_mem2_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 1, Slave 2, Configuration
Word Register 0*/
    uint32_t mbc1_1_mem2_blk_cfg_w1; /*!< Memory Block Checker 1, Domain 1, Slave 2, Configuration
```

```
Word Register 1*/
    uint32_t mbc1_1_mem3_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 1, Slave 3, Configuration
Word Register 0*/
    /* SECTION 8 - END */

    /* SECTION 9 - START */
    uint32_t mbc1_2_mem0_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 0, Slave 0, Configuration
Word Register 0*/
    uint32_t mbc1_2_mem1_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 0, Slave 1, Configuration
Word Register 0*/
    uint32_t mbc1_2_mem1_blk_cfg_w1; /*!< Memory Block Checker 1, Domain 0, Slave 1, Configuration
Word Register 1*/
    uint32_t mbc1_2_mem2_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 0, Slave 2, Configuration
Word Register 0*/
    uint32_t mbc1_2_mem2_blk_cfg_w1; /*!< Memory Block Checker 1, Domain 0, Slave 2, Configuration
Word Register 1*/
    uint32_t mbc1_2_mem3_blk_cfg_w0; /*!< Memory Block Checker 1, Domain 0, Slave 3, Configuration
Word Register 0*/
    /* SECTION 9 - END */

    /* SHARED BY SECTION 10, 11, 12 - START */
    uint32_t mbc2_memn_glbac0; /*!< Memory Block Checker 2, Global Access Control Register 0*/
    uint32_t mbc2_memn_glbac1; /*!< Memory Block Checker 2, Global Access Control Register 1*/
    uint32_t mbc2_memn_glbac2; /*!< Memory Block Checker 2, Global Access Control Register 2*/
    uint32_t mbc2_memn_glbac3; /*!< Memory Block Checker 2, Global Access Control Register 3*/
    uint32_t mbc2_memn_glbac4; /*!< Memory Block Checker 2, Global Access Control Register 4*/
    uint32_t mbc2_memn_glbac5; /*!< Memory Block Checker 2, Global Access Control Register 5*/
    uint32_t mbc2_memn_glbac6; /*!< Memory Block Checker 2, Global Access Control Register 6*/
    uint32_t mbc2_memn_glbac7; /*!< Memory Block Checker 2, Global Access Control Register 7*/
    /* SHARED BY SECTION 10, 11, 12 - END */

    /* SECTION 10 - START */
    uint32_t mbc2_0_mem0_blk_cfg_w0;  /*!< Memory Block Checker 2, Domain 0, Slave 0, Configuration
Word Register 0*/
    uint32_t mbc2_0_mem0_blk_cfg_w1;  /*!< Memory Block Checker 2, Domain 0, Slave 0, Configuration
Word Register 1*/
    uint32_t mbc2_0_mem0_blk_cfg_w2;  /*!< Memory Block Checker 2, Domain 0, Slave 0, Configuration
Word Register 2*/
    uint32_t mbc2_0_mem0_blk_cfg_w3;  /*!< Memory Block Checker 2, Domain 0, Slave 0, Configuration
Word Register 3*/
    uint32_t mbc2_0_mem0_blk_cfg_w4;  /*!< Memory Block Checker 2, Domain 0, Slave 0, Configuration
Word Register 4*/
    uint32_t mbc2_0_mem0_blk_cfg_w5;  /*!< Memory Block Checker 2, Domain 0, Slave 0, Configuration
Word Register 5*/
    uint32_t mbc2_0_mem0_blk_cfg_w6;  /*!< Memory Block Checker 2, Domain 0, Slave 0, Configuration
Word Register 6*/
    uint32_t mbc2_0_mem0_blk_cfg_w7;  /*!< Memory Block Checker 2, Domain 0, Slave 0, Configuration
Word Register 7*/
    uint32_t mbc2_0_mem0_blk_cfg_w8;  /*!< Memory Block Checker 2, Domain 0, Slave 0, Configuration
Word Register 8*/
    uint32_t mbc2_0_mem0_blk_cfg_w9;  /*!< Memory Block Checker 2, Domain 0, Slave 0, Configuration
Word Register 9*/
    uint32_t mbc2_0_mem0_blk_cfg_w10; /*!< Memory Block Checker 2, Domain 0, Slave 0, Configuration
Word Register 10*/
    uint32_t mbc2_0_mem1_blk_cfg_w0;  /*!< Memory Block Checker 2, Domain 0, Slave 1, Configuration
Word Register 0*/
    uint32_t mbc2_0_mem2_blk_cfg_w0;  /*!< Memory Block Checker 2, Domain 0, Slave 2, Configuration
Word Register 0*/
    uint32_t mbc2_0_mem2_blk_cfg_w1;  /*!< Memory Block Checker 2, Domain 0, Slave 2, Configuration
Word Register 1*/
```

```
    /* SECTION 10 - END */

    /* SECTION 11 - START */
    uint32_t mbc2_1_mem0_blk_cfg_w0;  /*!< Memory Block Checker 2, Domain 1, Slave 0, Configuration 
Word Register 0*/
    uint32_t mbc2_1_mem0_blk_cfg_w1;  /*!< Memory Block Checker 2, Domain 1, Slave 0, Configuration 
Word Register 1*/
    uint32_t mbc2_1_mem0_blk_cfg_w2;  /*!< Memory Block Checker 2, Domain 1, Slave 0, Configuration 
Word Register 2*/
    uint32_t mbc2_1_mem0_blk_cfg_w3;  /*!< Memory Block Checker 2, Domain 1, Slave 0, Configuration 
Word Register 3*/
    uint32_t mbc2_1_mem0_blk_cfg_w4;  /*!< Memory Block Checker 2, Domain 1, Slave 0, Configuration 
Word Register 4*/
    uint32_t mbc2_1_mem0_blk_cfg_w5;  /*!< Memory Block Checker 2, Domain 1, Slave 0, Configuration 
Word Register 5*/
    uint32_t mbc2_1_mem0_blk_cfg_w6;  /*!< Memory Block Checker 2, Domain 1, Slave 0, Configuration 
Word Register 6*/
    uint32_t mbc2_1_mem0_blk_cfg_w7;  /*!< Memory Block Checker 2, Domain 1, Slave 0, Configuration 
Word Register 7*/
    uint32_t mbc2_1_mem0_blk_cfg_w8;  /*!< Memory Block Checker 2, Domain 1, Slave 0, Configuration 
Word Register 8*/
    uint32_t mbc2_1_mem0_blk_cfg_w9;  /*!< Memory Block Checker 2, Domain 1, Slave 0, Configuration 
Word Register 9*/
    uint32_t mbc2_1_mem0_blk_cfg_w10; /*!< Memory Block Checker 2, Domain 1, Slave 0, Configuration 
Word Register 10*/
    uint32_t mbc2_1_mem1_blk_cfg_w0;  /*!< Memory Block Checker 2, Domain 1, Slave 1, Configuration 
Word Register 0*/
    uint32_t mbc2_1_mem2_blk_cfg_w0;  /*!< Memory Block Checker 2, Domain 1, Slave 2, Configuration 
Word Register 0*/
    uint32_t mbc2_1_mem2_blk_cfg_w1;  /*!< Memory Block Checker 2, Domain 1, Slave 2, Configuration 
Word Register 1*/
    /* SECTION 11 - END */

    /* SECTION 12 - START */
    uint32_t mbc2_2_mem0_blk_cfg_w0;  /*!< Memory Block Checker 2, Domain 2, Slave 0, Configuration 
Word Register 0*/
    uint32_t mbc2_2_mem0_blk_cfg_w1;  /*!< Memory Block Checker 2, Domain 2, Slave 0, Configuration 
Word Register 1*/
    uint32_t mbc2_2_mem0_blk_cfg_w2;  /*!< Memory Block Checker 2, Domain 2, Slave 0, Configuration 
Word Register 2*/
    uint32_t mbc2_2_mem0_blk_cfg_w3;  /*!< Memory Block Checker 2, Domain 2, Slave 0, Configuration 
Word Register 3*/
    uint32_t mbc2_2_mem0_blk_cfg_w4;  /*!< Memory Block Checker 2, Domain 2, Slave 0, Configuration 
Word Register 4*/
    uint32_t mbc2_2_mem0_blk_cfg_w5;  /*!< Memory Block Checker 2, Domain 2, Slave 0, Configuration 
Word Register 5*/
    uint32_t mbc2_2_mem0_blk_cfg_w6;  /*!< Memory Block Checker 2, Domain 2, Slave 0, Configuration 
Word Register 6*/
    uint32_t mbc2_2_mem0_blk_cfg_w7;  /*!< Memory Block Checker 2, Domain 2, Slave 0, Configuration 
Word Register 7*/
    uint32_t mbc2_2_mem0_blk_cfg_w8;  /*!< Memory Block Checker 2, Domain 2, Slave 0, Configuration 
Word Register 8*/
    uint32_t mbc2_2_mem0_blk_cfg_w9;  /*!< Memory Block Checker 2, Domain 2, Slave 0, Configuration 
Word Register 9*/
    uint32_t mbc2_2_mem0_blk_cfg_w10; /*!< Memory Block Checker 2, Domain 2, Slave 0, Configuration 
Word Register 10*/
    uint32_t mbc2_2_mem1_blk_cfg_w0;  /*!< Memory Block Checker 2, Domain 2, Slave 1, Configuration 
Word Register 0*/
    uint32_t mbc2_2_mem2_blk_cfg_w0;  /*!< Memory Block Checker 2, Domain 2, Slave 2, Configuration 
Word Register 0*/
```

```
    uint32_t mbc2_2_mem2_blk_cfg_w1;  /*!< Memory Block Checker 2, Domain 2, Slave 2, Configuration
Word Register 1*/
    /* SECTION 12 - END */

    /* SHARED BY SECTION 13, 14, 15 - START */
    uint32_t mrc0_memn_glbac0; /*!< Memory Region Controller 0, Global Access Control Register 0*/
    uint32_t mrc0_memn_glbac1; /*!< Memory Region Controller 0, Global Access Control Register 1*/
    uint32_t mrc0_memn_glbac2; /*!< Memory Region Controller 0, Global Access Control Register 2*/
    uint32_t mrc0_memn_glbac3; /*!< Memory Region Controller 0, Global Access Control Register 3*/
    uint32_t mrc0_memn_glbac4; /*!< Memory Region Controller 0, Global Access Control Register 4*/
    uint32_t mrc0_memn_glbac5; /*!< Memory Region Controller 0, Global Access Control Register 5*/
    uint32_t mrc0_memn_glbac6; /*!< Memory Region Controller 0, Global Access Control Register 6*/
    uint32_t mrc0_memn_glbac7; /*!< Memory Region Controller 0, Global Access Control Register 7*/
    /* SHARED BY SECTION 13, 14, 15 - END */

    /* SECTION 13 - START */
    uint32_t mrc0_0_rgd0_w0; /*!< Memory Region Controller 0, Domain 0, Region 0, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_0_rgd0_w1; /*!< Memory Region Controller 0, Domain 0, Region 0, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_0_rgd1_w0; /*!< Memory Region Controller 0, Domain 0, Region 1, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_0_rgd1_w1; /*!< Memory Region Controller 0, Domain 0, Region 1, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_0_rgd2_w0; /*!< Memory Region Controller 0, Domain 0, Region 2, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_0_rgd2_w1; /*!< Memory Region Controller 0, Domain 0, Region 2, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_0_rgd3_w0; /*!< Memory Region Controller 0, Domain 0, Region 3, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_0_rgd3_w1; /*!< Memory Region Controller 0, Domain 0, Region 3, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_0_rgd4_w0; /*!< Memory Region Controller 0, Domain 0, Region 4, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_0_rgd4_w1; /*!< Memory Region Controller 0, Domain 0, Region 4, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_0_rgd5_w0; /*!< Memory Region Controller 0, Domain 0, Region 5, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_0_rgd5_w1; /*!< Memory Region Controller 0, Domain 0, Region 5, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_0_rgd6_w0; /*!< Memory Region Controller 0, Domain 0, Region 6, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_0_rgd6_w1; /*!< Memory Region Controller 0, Domain 0, Region 6, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_0_rgd7_w0; /*!< Memory Region Controller 0, Domain 0, Region 7, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_0_rgd7_w1; /*!< Memory Region Controller 0, Domain 0, Region 7, Memory Region
Descriptor Register 1*/
    /* SECTION 13 - END */

    /* SECTION 14 - START */
    uint32_t mrc0_1_rgd0_w0; /*!< Memory Region Controller 0, Domain 1, Region 0, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_1_rgd0_w1; /*!< Memory Region Controller 0, Domain 1, Region 0, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_1_rgd1_w0; /*!< Memory Region Controller 0, Domain 1, Region 1, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_1_rgd1_w1; /*!< Memory Region Controller 0, Domain 1, Region 1, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_1_rgd2_w0; /*!< Memory Region Controller 0, Domain 1, Region 2, Memory Region
```

```
Descriptor Register 0*/
    uint32_t mrc0_1_rgd2_w1; /*!< Memory Region Controller 0, Domain 1, Region 2, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_1_rgd3_w0; /*!< Memory Region Controller 0, Domain 1, Region 3, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_1_rgd3_w1; /*!< Memory Region Controller 0, Domain 1, Region 3, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_1_rgd4_w0; /*!< Memory Region Controller 0, Domain 1, Region 4, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_1_rgd4_w1; /*!< Memory Region Controller 0, Domain 1, Region 4, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_1_rgd5_w0; /*!< Memory Region Controller 0, Domain 1, Region 5, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_1_rgd5_w1; /*!< Memory Region Controller 0, Domain 1, Region 5, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_1_rgd6_w0; /*!< Memory Region Controller 0, Domain 1, Region 6, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_1_rgd6_w1; /*!< Memory Region Controller 0, Domain 1, Region 6, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_1_rgd7_w0; /*!< Memory Region Controller 0, Domain 1, Region 7, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_1_rgd7_w1; /*!< Memory Region Controller 0, Domain 1, Region 7, Memory Region
Descriptor Register 1*/
    /* SECTION 14 - END */

    /* SECTION 15 - START */
    uint32_t mrc0_2_rgd0_w0; /*!< Memory Region Controller 0, Domain 2, Region 0, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_2_rgd0_w1; /*!< Memory Region Controller 0, Domain 2, Region 0, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_2_rgd1_w0; /*!< Memory Region Controller 0, Domain 2, Region 1, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_2_rgd1_w1; /*!< Memory Region Controller 0, Domain 2, Region 1, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_2_rgd2_w0; /*!< Memory Region Controller 0, Domain 2, Region 2, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_2_rgd2_w1; /*!< Memory Region Controller 0, Domain 2, Region 2, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_2_rgd3_w0; /*!< Memory Region Controller 0, Domain 2, Region 3, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_2_rgd3_w1; /*!< Memory Region Controller 0, Domain 2, Region 3, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_2_rgd4_w0; /*!< Memory Region Controller 0, Domain 2, Region 4, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_2_rgd4_w1; /*!< Memory Region Controller 0, Domain 2, Region 4, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_2_rgd5_w0; /*!< Memory Region Controller 0, Domain 2, Region 5, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_2_rgd5_w1; /*!< Memory Region Controller 0, Domain 2, Region 5, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_2_rgd6_w0; /*!< Memory Region Controller 0, Domain 2, Region 6, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_2_rgd6_w1; /*!< Memory Region Controller 0, Domain 2, Region 6, Memory Region
Descriptor Register 1*/
    uint32_t mrc0_2_rgd7_w0; /*!< Memory Region Controller 0, Domain 2, Region 7, Memory Region
Descriptor Register 0*/
    uint32_t mrc0_2_rgd7_w1; /*!< Memory Region Controller 0, Domain 2, Region 7, Memory Region
Descriptor Register 1*/
    /* SECTION 15 - END */
    /* SECTION 16 - START */
    uint32_t gpioa_lock;  /*!< GPIO A, Lock Register */
    uint32_t gpioa_pcns;  /*!< GPIO A, Pin Control Non-Secure Register */
    uint32_t gpioa_icns;  /*!< GPIO A, Interrupt Control Non-Secure Register */
    uint32_t gpioa_pcnp;  /*!< GPIO A, Pin Control Non-Privilege Register */
    uint32_t gpioa_icnp;  /*!< GPIO A, Interrupt Control Non-Privilege Register */
    uint32_t gpioa_icr0;  /*!< GPIO A, Interrupt Control Pin 0 */
    uint32_t gpioa_icr1;  /*!< GPIO A, Interrupt Control Pin 1 */
    uint32_t gpioa_icr2;  /*!< GPIO A, Interrupt Control Pin 2 */
    uint32_t gpioa_icr3;  /*!< GPIO A, Interrupt Control Pin 3 */
    uint32_t gpioa_icr4;  /*!< GPIO A, Interrupt Control Pin 4 */
    uint32_t gpioa_icr5;  /*!< GPIO A, Interrupt Control Pin 5 */
    uint32_t gpioa_icr6;  /*!< GPIO A, Interrupt Control Pin 6 */
    uint32_t gpioa_icr7;  /*!< GPIO A, Interrupt Control Pin 7 */
    uint32_t gpioa_icr8;  /*!< GPIO A, Interrupt Control Pin 8 */
    uint32_t gpioa_icr9;  /*!< GPIO A, Interrupt Control Pin 9 */
    uint32_t gpioa_icr10; /*!< GPIO A, Interrupt Control Pin 10 */
    uint32_t gpioa_icr11; /*!< GPIO A, Interrupt Control Pin 11 */

    uint32_t gpioa_icr16; /*!< GPIO A, Interrupt Control Pin 16 */
    uint32_t gpioa_icr17; /*!< GPIO A, Interrupt Control Pin 17 */
    uint32_t gpioa_icr18; /*!< GPIO A, Interrupt Control Pin 18 */
    uint32_t gpioa_icr19; /*!< GPIO A, Interrupt Control Pin 19 */
    uint32_t gpioa_icr20; /*!< GPIO A, Interrupt Control Pin 20 */
    uint32_t gpioa_icr21; /*!< GPIO A, Interrupt Control Pin 21 */
    uint32_t gpioa_icr22; /*!< GPIO A, Interrupt Control Pin 22 */
    uint32_t gpioa_icr23; /*!< GPIO A, Interrupt Control Pin 23 */
    uint32_t gpioa_icr24; /*!< GPIO A, Interrupt Control Pin 24 */
    uint32_t gpioa_icr25; /*!< GPIO A, Interrupt Control Pin 25 */
    uint32_t gpioa_icr26; /*!< GPIO A, Interrupt Control Pin 26 */
    uint32_t gpioa_icr27; /*!< GPIO A, Interrupt Control Pin 27 */

    uint32_t gpioa_icr30; /*!< GPIO A, Interrupt Control Pin 30 */
    uint32_t gpioa_icr31; /*!< GPIO A, Interrupt Control Pin 31 */

    uint32_t gpiob_lock;  /*!< GPIO B, Lock Register */
    uint32_t gpiob_pcns;  /*!< GPIO B, Pin Control Non-Secure Register */
    uint32_t gpiob_icns;  /*!< GPIO B, Interrupt Control Non-Secure Register */
    uint32_t gpiob_pcnp;  /*!< GPIO B, Pin Control Non-Privilege Register */
    uint32_t gpiob_icnp;  /*!< GPIO B, Interrupt Control Non-Privilege Register */
    uint32_t gpiob_icr0;  /*!< GPIO B, Interrupt Control Pin 0 */
    uint32_t gpiob_icr1;  /*!< GPIO B, Interrupt Control Pin 1 */
    uint32_t gpiob_icr2;  /*!< GPIO B, Interrupt Control Pin 2 */
    uint32_t gpiob_icr3;  /*!< GPIO B, Interrupt Control Pin 3 */
    uint32_t gpiob_icr4;  /*!< GPIO B, Interrupt Control Pin 4 */
    uint32_t gpiob_icr5;  /*!< GPIO B, Interrupt Control Pin 5 */
    uint32_t gpiob_icr6;  /*!< GPIO B, Interrupt Control Pin 6 */
    uint32_t gpiob_icr7;  /*!< GPIO B, Interrupt Control Pin 7 */
    uint32_t gpiob_icr8;  /*!< GPIO B, Interrupt Control Pin 8 */
    uint32_t gpiob_icr9;  /*!< GPIO B, Interrupt Control Pin 9 */
    uint32_t gpiob_icr10; /*!< GPIO B, Interrupt Control Pin 10 */
    uint32_t gpiob_icr11; /*!< GPIO B, Interrupt Control Pin 11 */

    uint32_t gpioc_lock;  /*!< GPIO C, Lock Register */
    uint32_t gpioc_pcns;  /*!< GPIO C, Pin Control Non-Secure Register */
    uint32_t gpioc_icns;  /*!< GPIO C, Interrupt Control Non-Secure Register */
    uint32_t gpioc_pcnp;  /*!< GPIO C, Pin Control Non-Privilege Register */
    uint32_t gpioc_icnp;  /*!< GPIO C, Interrupt Control Non-Privilege Register */
    uint32_t gpioc_icr0;  /*!< GPIO C, Interrupt Control Pin 0 */
    uint32_t gpioc_icr1;  /*!< GPIO C, Interrupt Control Pin 1 */
    uint32_t gpioc_icr2;  /*!< GPIO C, Interrupt Control Pin 2 */
    uint32_t gpioc_icr3;  /*!< GPIO C, Interrupt Control Pin 3 */
    uint32_t gpioc_icr4;  /*!< GPIO C, Interrupt Control Pin 4 */
    uint32_t gpioc_icr5;  /*!< GPIO C, Interrupt Control Pin 5 */
    uint32_t gpioc_icr6;  /*!< GPIO C, Interrupt Control Pin 6 */
    uint32_t gpioc_icr7;  /*!< GPIO C, Interrupt Control Pin 7 */
    uint32_t gpioc_icr8;  /*!< GPIO C, Interrupt Control Pin 8 */
    uint32_t gpioc_icr9;  /*!< GPIO C, Interrupt Control Pin 9 */
    uint32_t gpioc_icr10; /*!< GPIO C, Interrupt Control Pin 10 */
    uint32_t gpioc_icr11; /*!< GPIO C, Interrupt Control Pin 11 */
    uint32_t gpioc_icr12; /*!< GPIO C, Interrupt Control Pin 12 */
    uint32_t gpioc_icr13; /*!< GPIO C, Interrupt Control Pin 13 */
    uint32_t gpioc_icr14; /*!< GPIO C, Interrupt Control Pin 14 */
    uint32_t gpioc_icr15; /*!< GPIO C, Interrupt Control Pin 15 */
    uint32_t gpioc_icr16; /*!< GPIO C, Interrupt Control Pin 16 */
    uint32_t gpioc_icr17; /*!< GPIO C, Interrupt Control Pin 17 */
    uint32_t gpioc_icr18; /*!< GPIO C, Interrupt Control Pin 18 */
    uint32_t gpioc_icr19; /*!< GPIO C, Interrupt Control Pin 19 */

    uint32_t gpiod_lock;  /*!< GPIO D, Lock Register */
    uint32_t gpiod_pcns;  /*!< GPIO D, Pin Control Non-Secure Register */
    uint32_t gpiod_icns;  /*!< GPIO D, Interrupt Control Non-Secure Register */
    uint32_t gpiod_pcnp;  /*!< GPIO D, Pin Control Non-Privilege Register */
    uint32_t gpiod_icnp;  /*!< GPIO D, Interrupt Control Non-Privilege Register */
    uint32_t gpiod_icr0;  /*!< GPIO D, Interrupt Control Pin 0 */
    uint32_t gpiod_icr1;  /*!< GPIO D, Interrupt Control Pin 1 */
    uint32_t gpiod_icr2;  /*!< GPIO D, Interrupt Control Pin 2 */
    uint32_t gpiod_icr3;  /*!< GPIO D, Interrupt Control Pin 3 */
    uint32_t gpiod_icr4;  /*!< GPIO D, Interrupt Control Pin 4 */
    uint32_t gpiod_icr5;  /*!< GPIO D, Interrupt Control Pin 5 */
    uint32_t gpiod_icr6;  /*!< GPIO D, Interrupt Control Pin 6 */
    uint32_t gpiod_icr7;  /*!< GPIO D, Interrupt Control Pin 7 */
    uint32_t gpiod_icr8;  /*!< GPIO D, Interrupt Control Pin 8 */
    uint32_t gpiod_icr9;  /*!< GPIO D, Interrupt Control Pin 9 */
    uint32_t gpiod_icr10; /*!< GPIO D, Interrupt Control Pin 10 */
    uint32_t gpiod_icr11; /*!< GPIO D, Interrupt Control Pin 11 */
    /* SECTION 16 - END */
} tzm_secure_config_t;
```

Almost all data in TZ-M preset data structure is one to one copy to corresponding registers. The data with specific function is described in the following sections.

4.3.4.2 TZ-M data with specific function

4.3.4.2.1 uint32_t tzm_magic

tzm_magic is used to identify correct start of TZ-M preset data block. This value must be set to four letters “TZ-M”. In little endian format it corresponds to value 0x4d2d5a54. Every TZ-M preset data block must start with this value otherwise boot process fails.

4.3.4.2.2 uint32_t tzm_control

The TZ-M preset data initialization is configurable to speed up boot time. The data is split into several sections (see comments in data structure definition) and data not used by application can be skipped during TZ-M preset data initialization. The variable tzm_control controls, which section is skipped. If tzm_control = 0, all data is initialized. Setting of corresponding bits in tzm_control will skip selected section. Note that TZ-M preset data must always contain all data (all sections) defined in tzm_secure_config_t structure even if data is skipped during initialization. But if initialization of some section is disabled, the data is ignored.

Table 16. tzm_control variable definition

Bit	Description
31-21	Reserved
20	Disable GPIO initialization (SECTION 16)
19	Reserved
18	Disable TRDC MRC0, domain 2 initialization (SECTION 15)
17	Disable TRDC MRC0, domain 1 initialization (SECTION 14)
16	Disable TRDC MRC0, domain 0 initialization (SECTION 13)
15	Reserved
14	Disable TRDC MBC2, domain 4 initialization (SECTION 12)
13	Disable TRDC MBC2, domain 4 initialization (SECTION 11)
12	Disable TRDC MBC2, domain 0 initialization (SECTION 10)
11	Reserved
10	Disable TRDC MBC1, domain 2 initialization (SECTION 9)
9	Disable TRDC MBC1, domain 1 initialization (SECTION 8)
8	Disable TRDC MBC1, domain 0 initialization (SECTION 7)
7	Reserved
6	Disable TRDC MBC0, domain 2 initialization (SECTION 6)
5	Disable TRDC MBC0, domain 1 initialization (SECTION 5)
4	Disable TRDC MBC0, domain 0 initialization (SECTION 4)
3	Reserved
2	Disable secure SAU initialization (SECTION 3)
1	Disable non-secure MPU initialization (SECTION 2)
0	Disable secure MPU initialization (SECTION 1)

The data shared by more sections means that data is initialized if at least one of the sections is also initialized. If no one section is initialized, this data initialization is also skipped.

The data, which doesn’t belong to any section, is always initialized.

4.3.4.2.3 uint32_t cm33_misc_ctrl

The cm33_misc_ctrl variable controls configuration of several core TrustZone related features spread among different SCB core registers. For better data size efficiency, the control of these features was merged into single 32-bit variable. The list of core features configured by this variable is listed below.

Table 17. cm33_misc_ctrl variable definition

Bit	Description
31-5	Reserved
4	Defines value of SYSRESETREQ bit in AIRCR register

Table continues on the next page...

Table 17. cm33_misc_ctrl variable definition (continued)

Bit	Description
3	Defines value of BFHFNMINS bit in AIRCR register
2	Defines value of PRIS bit in AIRCR register
1	Defines value of SLEEPDEEPS bit in SCR register
0	Defines value of SECUREFAULTENA bit in SHCSR register

NOTE

The TrustZone configuration takes effect already in ROM code execution before a jump to the user application. Therefore, the user's TrustZone configuration data must allow ROM code to successfully jump to user application. In order to avoid boot process failure, the user's TrustZone settings must configure:

1. Whole ROM space (0x14800000-0x1481FFFF) as secure privilege,
2. If the secure MPU is used, whole ROM space (0x14800000-0x1481FFFF) must be configured for code execution.

4.3.5 Secure boot usage

The device allows booting of public-key signed images. The device boot ROM supports following types of security protected boot modes:

- Secure boot with signed image
- Secure boot with signed image from encrypted internal flash regions

Each of these options has attributes related to manufacturability, the firmware update scheme and level of protection against attacks.

The ROM further supports public keys and image revocation i.e. the method of not allowing new updates to be applied unless they are of a specific version. This is the basis for roll back protection.

The following section describes the main steps for key provisioning, creating signed images and loading the signed images into the target. The elftosb tool is the NXP image signing and SB file creating tool for Windows/Linux/MAC. Refer to elftosb User's guide for detailed step-by-step guide describing use case of a tool to generate signed and encrypted (SB) image.

The Secure Binary (SB) image format is a command-based firmware update image. ROM bootloader's RecieveSBFile command can verify digital signature to process SB file for secure provisioning. The layout of an SB 3.0 file is shown in elftosb tool User's guide. Also one can refer to Signed image structure and Loadable firmware container format to understand supported image formats.

4.3.5.1 Keys and certificates

The required inputs for image signing process are:

- JSON configuration text file
  - Index of the root key to be used to sign Intermediate Certificate
  - Intermediate Certificate's 32-bit revocation word
- Application image to be signed
  - raw bin
- 1 to 4 root certificates (PEM or DER) with 1 to 4 ECC public keys (RoTK)
- 1 ECC private key (which creates the pair for one of the $\text{RoTK}_{\text{private}}$ root keys)
  - This is used by the tool to sign the Intermediate Certificate.

- Intermediate Certificate (PEM or DER) with ECC public key
- ECC private key – creating a pair with the public key from Intermediate Certificate
  - This is used by the tool to sign the application image.

4.3.5.2 Fuses preparation

Following fuse values are checked for firmware update process:

- SB3KDK (CUST_PROD_OEMFW_ENC_SK)
- RoTKTH (CUST_PROD_OEMFW_AUTH_PUK for CM33 and OEM NBU FW authentication)
- NXP_PROD_nPrivFW_AUTH_PUK (NXP owned NBU FW authentication)

Following are some of the cases when authentication and firmware update can fail:

- If any of the root key is revoked in fuse, CUST_PROD_OEMFW_AUTH_PUK_REVOKE[3:0])
- If sb3 file is signed with the signingCertificateConstraint less than the IMG_KEY_REVOKE[15:0]
- $\text{RoTK}_{\text{public}}$ does not match with RoTKTH in fuses.
- KeyBlobEncryptionKey does not match with SB3KDK in fuses

4.3.5.3 Signed image preparation

NXP provides elftosb tool which prepares signed binary, which can be loaded to target device. The input for elftosb program is plain application image in binary format, image signing certificate, associated private key and JSON format configuration structure. For detailed step-by-step guide refer to elftosb User's guide or application note.

SB3 image generation process supports loader commands such as ERASE, LOAD, programIFR, etc.

It is 2-step process to generate encrypted sb3 image. First, user needs to generate signed image and required inputs are mentioned in Keys and certificates. Signed image is considered as an input to generate SB3 image whereas ECC private key is used by the elftosb tool to sign SB3 header.

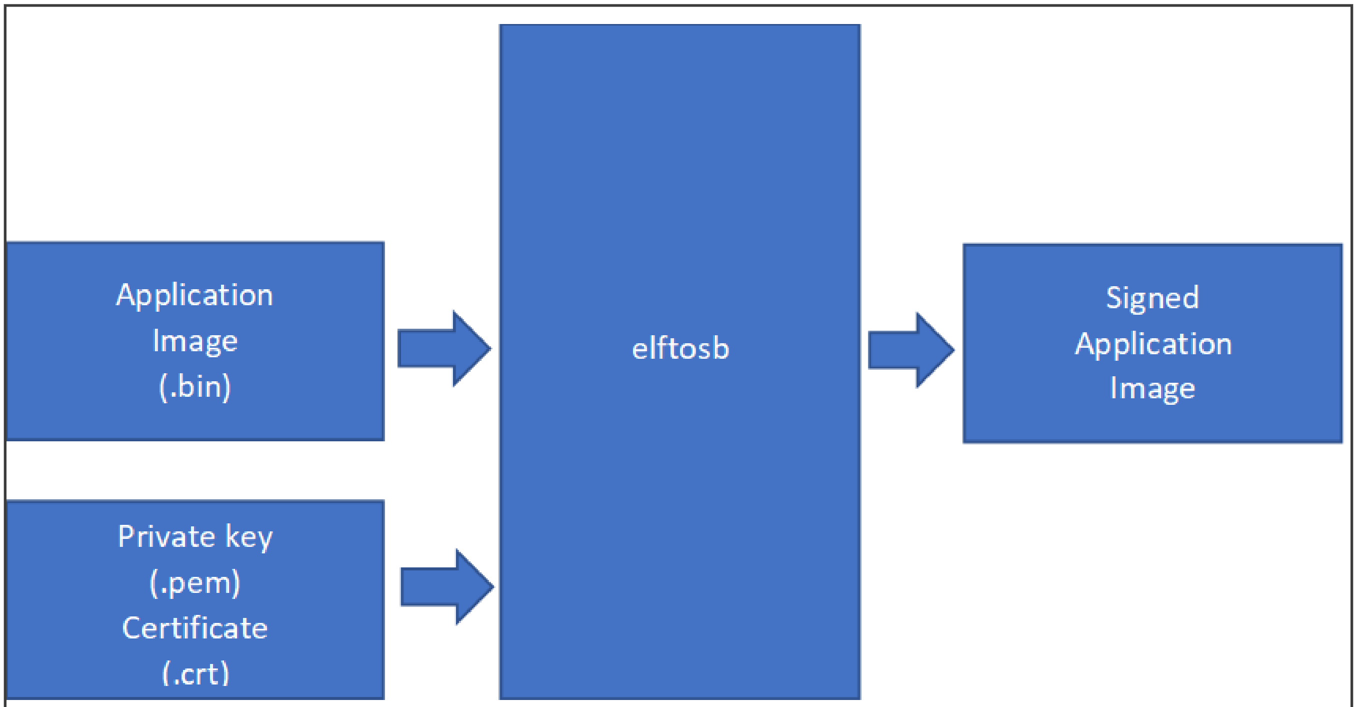


Figure 20. Signed image preparation

Following information are needed by elftosb tool to produce Internal flash (XIP) signed image:

- Plain application binary
- Start address of application binary
- TZ related settings

4.3.5.4 Loading signed image

The signed image could be programmed directly into the device using various methods:

- ROM In System Programming (ISP) using write-memory blhost command
- ROM ISP using Secure FW update container (receive-sb command)
- Programming signed image directly from target application using ROM API
- Flashing signed image through debugger

4.3.5.5 Secure boot status code

Image authentication (based on ECDSA/CMAC verification) status can be checked by CMC0_BSR register. It can also be checked using one of the ISP properties, ?get-property 0x14?.

Table 18. Secure boot status code

Status code	Description
(CMC0_BSR [0]) = 1b	NBU FW CMAC authentication was success.
(CMC0_BSR[1]) = 1b	Main flash image CMAC authentication was success.
(CMC0_BSR[2]) = 1b	Both (Main flash image, NBU FW) CMAC authentication was success.
(CMC0_BSR[31:0]) = 0x0b38f000	NBU FW ECDSA authentication was success.
(CMC0_BSR[31:0]) = 0x0b38f300	NBU FW ECDSA authentication failed.

Chapter 5 ROM API

5.1 Overview

ROM bootloader provides several APIs for users. Disabling the interrupts before making any ROM API call is suggested, since API code does not deal with interrupts. Flash APIs are mainly used to manage flash including read, write and erase. Similarly, SPI NOR APIs are mainly used to manage SPI NOR including read, write and erase. nboot APIs are mainly used to do security related operations including image authentication and fuse programming. kb APIs are mainly used to process SB file.

The ROM API table locates at address 0x14816800. See Figure below for the ROM API layout.

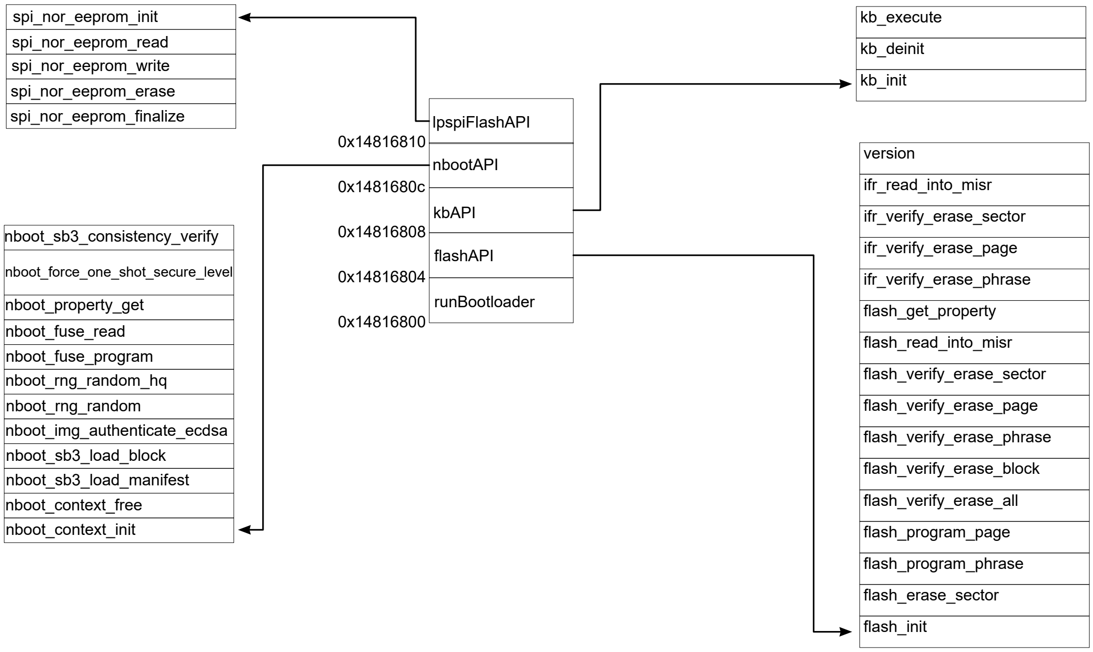


Figure 21. ROM API layout

5.2 SPI Flash API

5.2.1 spi_eeprom_init

This API is used to initialize SPI Flash with requested baud rate and internally reads JEDEC ID as sanity check.

Prototype:

```
status_t spi_nor_eeprom_init(uint32_t baudRate);
```

Parameters

Table 19. spi_eeprom_init parameters

Parameter	Description
baudRate	value used to configure SPI NOR flash for subsequent operations

5.2.2 spi_eeprom_read

This API is used to read SPI Flash memory contents.

Prototype:

```
status_t spi_eeprom_read(uint8_t *dest, uint32_t NoOfBytes, uint32_t address, bool
requestFastRead);
```

Parameters

Table 20. spi_eeprom_read parameters

Parameter	Description
dest	pointer to destination buffer to store memory contents read from SPI flash
NoOfBytes	number of bytes to read
address	absolute start address of SPI Flash from which read operation should be prformed

requestFastRead	true = Uses FAST READ (0Bh)
	false = Uses simple read READ (03h)
	• User must check AC chacracteristics of Flash memory model and should know SPI functional clock before requesting requestFastRead. Otherwise requestFastRead option can cause a bootleneck in data transaction speed as command buffer involves dummy byte.

5.2.3 spi_eeprom_write

This API is used to program data to SPI flash.

Prototype:

```
status_t spi_eeprom_write(uint8_t *data, uint32_t NoOfBytes, uint32_t address);
```

Parameters

Table 21. spi_eeprom_write parameter

Parameter	Description
data	Pointer to the buffer of data that is to be programmed into SPI flash.
NoOfBytes	Number of bytes to be programmed.
Address	Start address of the SPI Flash memory to be programmed.

5.2.4 spi_eeprom_erase

This API is used to erase SPI flash contents with specific erase size.

Prototype:

```
status_t spi_eeprom_erase(uint32_t address, eraseOptions_t option);
```

Parameters

Table 22. spi_eeprom_erase parameters

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>address</td><td>Start address of the SPI Flash memory to be erased.</td></tr>
<tr><td>option</td><td>Following are the options supported for erase operation:
<pre>
typedef enum
{
    kSize_ErasePage = 0x1,
    kSize_Erase4K = 0x2,
    kSize_Erase32K = 0x3,
    kSize_Erase64K = 0x4,
    kSize_EraseAll = 0x5,
} eraseOptions_t;
</pre>
<p>NOTE</p>
<p>kSize_ErasePage option is not supported for all memory models. Example: Adesto flash can support page erase commands but not Micron.</p></td></tr>
</table>

5.2.5 spi_eeprom_finalize

This API is used de-initialize IOMUX and clock settings used for SPI Flash operations and resets all LPSPI logic and control registers.

Prototype:

```
void spi_eeprom_finalize(void);
```

5.2.6 SPI Flash API status code

The following table lists the SPI Flash API status code.

Table 23. SPI Flash API status code

Status	Code
kStatus_Success	0
kStatus_Fail	1

5.3 nboot API

The ROM API table is located at address 0x1481680c and contains absolute ROM API function addresses which can be called using function pointers. Only secure boot related functions are described in this chapter.

The main purpose of these APIs is to provide access to functions used and implemented in ROM to authenticate the application image and processing sb3 files.

5.3.1 nboot_context_init

This API is used for initializing the nboot context data structure. It must be called before calling other nboot APIs which performs authentication and sb3 processing.

Prototype:

```
#define NBOOT_CONTEXT_SIZE_IN_BYTES (156u)
typedef struct
{
    uint32_t blockSize;
    uint32_t totalBlocks;
    uint32_t processData;
    uint8_t
context[NBOOT_CONTEXT_SIZE_IN_BYTES];
} nboot_context_t;
nboot_status_t nboot_context_init(nboot_context_t *context);
```

Parameters

Table 24. nboot_context_init parameters

Parameter	Description
context	Pointer to nboot_context_t data structure in memory to store runtime state.

5.3.2 nboot_context_free

This API is used to release context used by all nboot functions.

Prototype:

```
nboot_status_t nboot_context_free(nboot_context_t *context);
```

Parameters

Parameter	Description
context	Pointer to nboot_context_t data structure in memory used to store runtime state.

5.3.3 nboot_sb3_load_manifest

This API is used to verify nboot sb3.1 manifest. It loads keys into the key store so that they can be used for subsequent operations such as nboot_sb3_load_block. NBOOT context must be initialized by the API nboot_context_init before calling this API.

Prototype:

```
nboot_status_t nboot_sb3_load_manifest(nboot_context_t *context, uint32_t *manifest);
```

Parameters

Table 25. nboot_sb3_load_manifest parameters

Parameter	Description
context	Pointer to nboot_context_t data structure in memory to store runtime state.
manifest	Pointer to the input manifest buffer.

5.3.4 nboot_sb3_load_block

This API is used to verify and decrypt NBOOT SB3.1 block. Decryption is performed in-place. NBOOT context must be initialized by the API nboot_context_init before calling this API.

Prototype:

```
nboot_status_t nboot_sb3_load_block(nboot_context_t *context, uint32_t *block);
```

Parameters

Table 26. nboot_sb3_load_block parameters

Parameter	Description
context	Pointer to nboot_context_t data structure in memory to store runtime state.
block	Pointer to the input block

5.3.5 nboot_img_authenticate_ecdsa

This API is used to authenticate signed image with asymmetric cryptography.

Prototype:

```
nboot_status_t nboot_img_authenticate_ecdsa(nboot_context_t *context, uint8_t imageStartAddress[],
nboot_bool_t *isSignatureVerified);
```

Parameters

Table 27. nboot_img_authenticate_ecdsa

Parameter	Description
context	Pointer to nboot_context_t data structure in memory to store runtime state.
imageStartAddress	Pointer to start of the image in memory (32-bit word aligned)

isSignatureVerified	Pointer to memory holding function call result.
	typedef enum
	{
	kNBOOT_TRUE = 0x3C5AC33Cu,
	kNBOOT_TRUE256 = 0x3C5AC35Au,
	kNBOOT_TRUE384 = 0x3C5AC3A5u,
	kNBOOT_FALSE = 0x5AA55AA5u,
	kNBOOT_OperationAllowed = 0x3c5a33ccU,
	kNBOOT_OperationDisallowed = 0x5aa5cc33U,
	} nboot_bool_t;

5.3.6 nboot_rng_random

This API is used to get random number(s).

Prototype:

```
nboot_status_t nboot_rng_random(nboot_context_t *context, void *buf, size_t bufLen);
```

Parameters

Table 28. nboot_rng_random

Parameter	Description
context	Pointer to nboot_context_t data structure in memory to store runtime state.
Buf	Pointer to buffer in memory to store random number
bufLen	Buffer length in number of bytes.

5.3.7 nboot_rng_random_hq

This API is used to get high quality random number(s).

Prototype:

```
nboot_status_t nboot_rng_random_hq(nboot_context_t *context, void *buf, size_t bufLen);
```

Parameters

Table 29. nboot_rng_random_hq

Parameter	Description
context	Pointer to nboot_context_t data structure in memory to store runtime state.
Buf	Pointer to buffer in memory to store random number
bufLen	Buffer length in number of bytes.

5.3.8 nboot_fuse_program

This API is used to program a fuse word at a given address with new data. Before using this API, voltage must be regulated for over-drive and normalize voltage after operation is completed.

Prototype:

```
nboot_status_t nboot_fuse_program(nboot_context_t *context, uint32_t addr, uint32_t *data, uint32_t
systemClockFrequencyMHz);
```

Parameters

Table 30. nboot_fuse_program

Parameter	Description
context	Pointer to nboot_context_t data structure in memory to store runtime state.
Addr	Fuse index
Data	Pointer to data expected to be programmed in fuse
systemClockFrequencyMHz	Boot frequency

5.3.9 nboot_fuse_read

This API is used to read a fuse word.

Prototype:

```
nboot_status_t nboot_fuse_read(nboot_context_t *context,
                                       uint32_t addr,
                                       uint32_t *data,
                                       uint32_t systemClockFrequencyMHz);
```

Parameters

Table 31. nboot_fuse_read

Parameter	Description
context	Pointer to nboot_context_t data structure in memory to store runtime state.
Addr	Fuse index
Data	Pointer to data buffer expecting fuse contents after successful read
systemClockFrequencyMHz	Boot frequency

5.3.10 nboot_property_get

This API is used to read property. One of the important properties that can be read is the property that last authentication of signed image container has succeeded.

Prototype:

```
nboot_status_t nboot_property_get(nboot_context_t *context,
                                 uint32_t propertyId,
                                 uint8_t *destData,
                                 size_t *dataLen);
```

Parameters

Table 32. nboot_property_get parameters

Parameter	Description
context	Pointer to nboot_context_t data structure in memory to store runtime state.
propertyId	Property ID must be supported by nboot
destData	Pointer to data buffer for storing returned contents
dataLen	Data buffer length

Property IDs supported are as follows:

Table 33. Supported property IDs

Property	ID	Description
DICE_CDI	0x10	Returns 32 bytes of CDI value.
IMAGE_HASH	0x20	Return 64 bytes (SHA-512) value containing hash of the last authenticated image.
PSA_BOOT_SEED	0x30	Returns 32 bytes boot seed.
LAST_AUTH_STATE	0x40	Returns 4 bytes status of last authentication.

Table continues on the next page...

Table 33. Supported property IDs (continued)

Property	ID	Description
ELE_ROM_VERSION	0x50	Returns 8 bytes of ELE ROM version.
ELE_FW_VERSION	0x51	If ELE FW not loaded successfully, returns 0xFFFFFFFF as 1st word. Else returns 2 words, 1st word: version, 2nd word – commit id.
DTRK_ATTEST_PUBK	0x60	Returns 64 bytes value – public part of the DTRK_ATTEST private key
UUID	0x90	Returns UUID to serve as device unique identifier.

5.3.11 nboot_force_one_shot_secure_level

This API is used to switch ELE to specified secure level access for next command.

Prototype:

```
nboot_status_t nboot_force_one_shot_secure_level (nboot_context_t *context,

nboot_security_level_t secLvl);
```

Parameters:

Table 34. nboot_force_one_shot_secure_level parameters

Parameter	Description
context	Pointer to nboot_context_t data structure in memory to store runtime state.
secLvl	Required security level, must be either equal or lower as security level of the host sending this command.

The following security levels are supported.

Table 35. Supported security levels

Security level	Value
Non-Secure	00h
Non-Secure Privileged	01h
Secure	02h
Secure Privileged	03h

5.3.12 nboot_sb3_consistency_verify

This API is used to authenticate SB3.1 file header and verifies consistency of all data blocks to avoid processing of corrupted data.

Prototype:

```
nboot_status_t nboot_sb3_consistency_verify(nboot_context_t *context,
                                            uint8_t sb3Data[],
                                            nboot_bool_t *isConsistent);
```

Parameters:

Table 36. nboot_sb3_consistency_verify parameters

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>context</td><td>Pointer to nboot_context_t data structure in memory to store runtime state</td></tr>
<tr><td>sb3Data</td><td>Pointer to start of SB3.1 file to be verified</td></tr>
<tr><td>isConsistent</td><td>Pointer to secure bool indicating verification result:
<pre>
typedef enum
{
    kNBOOT_TRUE = 0x3C5AC33Cu,
    kNBOOT_TRUE256 = 0x3C5AC35Au,
    kNBOOT_TRUE384 = 0x3C5AC3A5u,
    kNBOOT_FALSE = 0x5AA55AA5u,
    kNBOOT_OperationAllowed = 0x3c5a33ccu,
    kNBOOT_OperationDisallowed = 0x5aa5cc33u,
} nboot_bool_t;
</pre></td></tr>
</table>

5.3.13 nboot API status code

The following table lists the nboot API status code.

Table 37. nboot API status code

Status	Code
kStatus_NBOOT_Success	0x5a5a5a5a
kStatus_NBOOT_Fail	0x5a5aa5a5
kStatus_NBOOT_InvalidArgument	0x5a5aa501
kStatus_NBOOT_RequestTimeout	0x5a5aa502
kStatus_NBOOT_ResourceBusy	0x5a5aa503
kStatus_NBOOT_OperationNotAvaialable	0x5a5aa5e5
kStatus_NBOOT_MemcpyFail	0x5a5a845a

5.4 kb API

This set of APIs provides interface to bootloader API functions. The main purpose of this APIs is to provide ROM functions that can be used to process SB file.

5.4.1 kb_init

This API is used to initialize an internal nboot context necessary to process sb3 file format.

Prototype:

```
#define kRomApiVersion 1u
typedef enum kb_operation_tag
{
    kRomAuthenticateImage = 1, //!< Authenticate a signed image. Not supported.
    kRomLoadImage = 2,         //!< Load SB file.
    kRomOperationCount,
} kb_operation_t;
typedef struct kb_options_tag
```

```
{
    uint32_t version; //!< Should be set to #kRomApiVersion.
    uint8_t *buffer;  //!< Caller-provided buffer used by Kboot.
    uint32_t bufferLength; //!< Size of the caller-provided buffer
    kb_operation_t op;     //!< Should be set to kRomLoadImage
    uint32_t reserved[6];  //!< Should be set to 0s
} kb_options_t;
typedef void kb_session_ref_t;
status_t kb_init(kb_session_ref_t **session_out, const kb_options_t *options);
```

The function returns session_out; a pointer to the internal context initialized within the buffer provided by the options argument. The provided buffer shall not be modified until it is released by the kb_deinit call. Size of the provided buffer shall be greater than 728 bytes - the function returns kStatus_RomApiBufferSizeNotEnough (10802) when the buffer is too small.

5.4.2 kb_deinit

This API is used to release the session pointer and finalize sb3 file processing.

Prototype:

```
status_t kb_deinit(kb_session_ref_t *session);
```

Parameters

Table 38. kb_deinit parameters

Parameter	Description
session	pointer obtained from kb_init function

5.4.3 kb_execute

This API is used to decrypt sb3 file and store signed image contents specified by loader command supported while generating sb3 image through Json configuration. If sb3 file to be processed includes sbloader command “programFuses” then voltage must be regulated for over-drive before sb3 file processing start it shall be and normalized once operation is completed.

Prototype:

```
status_t kb_execute(kb_session_ref_t *session, const uint8_t *data, uint32_t dataLength);
```

Parameters

Table 39. kb_execute parameters

Parameter	Description
session	pointer obtained from kb_init function
data	pointer to start of sb file data in memory
dataLength	sb file data length in bytes

5.4.4 kb API status code

Table 40. kb API status code

Status	Code	Description
kStatus_Success	0	API succesfully finished its operation

Table continues on the next page...

Table 40. kb API status code (continued)

Status	Code	Description
kStatus_Fail	1	API failed and it could not finish its operation
kStatus_InvalidArgument	4	API cannot proceed because the input argument value is invalid
kStatus_RomLdrDataUnderrun	10109	API needs to be called again with more data (next data chunk)
kStatus_RomLdrJumpReturned	10110	API finished its execution but the jump to user code returned
kStatus_RomLdrRollbackBlocked	10115	New firmware version is lesser than the present one
kStatus_RomLdrPendingJumpCommand	10119	Returned by kb_execute; Call of kb_finish is needed to do the requested jump
kStatus_RomApiBufferSizeNotEnough	10802	API cannot proceed because the provided buffer (via call of kb_init in options argument) is not large enough
kStatus_RomApiInvalidBuffer	10803	API cannot proceed because the provided buffer pointer is NULL or buffer size is 0

5.5 Flash API

Flash APIs implement all the flash operation supported by CM33 FMU and RF-FMU1. Details of each API are described in this section.

5.5.1 flash_init

This API is used for initializing the flash controller and the flash_config context. It must be called before calling other flash APIs.

Prototype

```
status_t flash_init (flash_config_t *config);
```

Parameters

Table 41. flash_init parameters

Parameter	Description
config	Pointer to flash_config_t data structure in memory to store driver runtime state.

flash_config_t is defined as following:

```
typedef struct _flash_mem_descriptor {
    uint32_t blockBase;
    uint32_t totalSize;
    uint32_t blockCount;
} flash_mem_desc_t;

typedef struct _flash_ifr_desc {
    uint32_t pflashIfr0Start;
    uint32_t pflashIfr0MemSize;
} flash_ifr_desc_t;

typedef struct _msf1_config {
    flash_mem_desc_t flashDesc;
    flash_ifr_desc_t ifrDesc;
} msf1_config_t;

typedef struct _flash_config {
    msf1_config_t msf1Config[2];
} flash_config_t;
```

Example:

```
typedef struct
{
    __I void (*runBootloader)(void *arg);           //!< Function to start the bootloader executing.
    __I FLASH_API_Type *flashApiBase;         //!< Internal Flash driver API.
    __I KB_API_Type *kbApiBase;               //!< Bootloader API.
__I NBOOT_API_Type *nbootApiBase;         //!< Image authentication API.
    __I SPI_FLASH_API_Type *spiflashApiBase;             //!< SPI Flash API.

} ROM_API_TYPE;
/** ROM API base address */
#define ROM_API_BASE                         (0x14816800UL)
/** ROM API base pointer */
#define ROM_API                              ((ROM_API_TYPE*) ROM_API_BASE)
/** FLASH API base pointer */
#define FLASH_API                            (ROM_API->flashApiBase)
static flash_config_t flashConfig;
FLASH_API->flash_init(&flashConfig);
```

5.5.2 flash_erase_sector

This API is used for erasing specified flash sector (8 KB) in flash or User IFR(IFR0). This API will erase sectors which include the start address and (start + lengthInBytes - 1) address.

Prototype

```
status_t flash_erase_sector (flash_config_t *config, FMU_Type *base, uint32_t start, uint32_t
lengthInBytes, uint32_t key);
```

Parameters

Table 42. flash_erase_sector parameters

Parameter	Description
Config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1
Start	The starting address of the required flash memory to be erased. The starting address must be phrase-aligned (4-word or 16-byte).
lengthInBytes	The length, given in bytes (not words or long words) to be erased.
Key	Key is used to validate erase operation. Must be set to "kFLASH_ApiEraseKey"

Example:

```
#define FOUR_CHAR_CODE(a, b, c, d) (((d) << 24) | ((c) << 16) | ((b) << 8) | ((a)))
#define CM33_FMU                                    ((FMU_Type *)0x40020000u)
enum _flash_driver_api_keys
{
           kFLASH_ApiEraseKey = FOUR_CHAR_CODE('l', 'f', 'e', 'k')
};
result = FLASH_API->flash_erase_sector(&flashConfig, CM33_FMU, 0x0, 4, kFLASH_ApiEraseKey);
```

5.5.3 flash_program_phrase

This API is used for programming phrase-aligned data to a previously erased flash or User IFR phrase.

Prototype

```
status_t flash_program_phrase (flash_config_t *config, FMU_Type *base, uint32_t start,
                          uint32_t *src, uint32_t lengthInBytes);
```

Parameters

Table 43. flash_program_phrase parameters

Parameter	Description
config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1
Start	The starting address of the required flash memory to be programmed. The starting address must be phrase-aligned (4-word or 16-byte).
Src	Pointer to the source buffer of data that is to be programmed into flash.
lengthInBytes	The length, given in bytes (not words or long words) to be programmed. The lengthInBytes must be phrase-aligned (4-word or 16-byte).

5.5.4 flash_program_page

This API is used for programming page aligned data to a previously erased flash or User IFR page.

Prototype

```
status_t flash_program_page (flash_config_t *config, FMU_Type *base, uint32_t start,
                       uint32_t *src, uint32_t lengthInBytes);
```

Parameters

Table 44. flash_program_page parameters

Parameter	Description
config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1
Start	The starting address of the required flash memory to be programmed. The starting address must be page-aligned (32-word or 128-byte).
Src	Pointer to the source buffer of data that is to be programmed into flash.

Table continues on the next page...

Table 44. flash_program_page parameters (continued)

Parameter	Description
lengthInBytes	The length, given in bytes (not words or long words) to be programmed. The lengthInBytes must be page-aligned (32-word or 128-byte).

5.5.5 flash_verify_erase_all

This API is used for checking if all flash and IFR space are in the erased state..

Prototype

```
status_t flash_verify_erase_all (FMU_Type *base);
```

Parameters:

Table 45. flash_verify_erase_all parameters

Parameter	Description
FMU base	Base pointer to FMU or RF-FMU1

5.5.6 flash_verify_erase_block

This API is used for checking if a flash block is in the erased state.

Prototype

```
status_t flash_verify_erase_block (flash_config_t *config, FMU_Type *base, uint32_t blockaddr);
```

Parameters

Table 46. flash_verify_erase_block parameters

Parameter	Description
config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1
blockaddr	Starting address of a flash block, needs to be block-aligned

5.5.7 flash_verify_erase_phrase

This API is used for checking if specified flash phrases are in the erased state.

Prototype

```
status_t flash_verify_erase_phrase (flash_config_t *config, FMU_Type *base, uint32_t start, uint32_t
lengthInBytes);
```

Parameters

Table 47. flash_verify_erase_phrase parameters

Parameter	Description
config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1

Table continues on the next page...

Table 47. flash_verify_erase_phrase parameters (continued)

Parameter	Description
Start	The starting address of required flash memory to be checked. The starting address must be phrase-aligned (4-word or 16-byte).
lengthInBytes	The length, given in bytes (not words or long words) to be checked. The lengthInBytes must be phrase-aligned (4-word or 16-byte).

5.5.8 flash_verify_erase_page

This API is used for checking if specified flash pages are in the erased state.

Prototype

```
status_t flash_verify_erase_page (flash_config_t *config, FMU_Type *base, uint32_t start, uint32_t
lengthInBytes);
```

Parameters

Table 48. flash_verify_erase_page parameters

Parameter	Description
config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1
Start	The starting address of required flash memory to be checked. The starting address must be page-aligned (32-word or 128-byte).
lengthInBytes	The length, given in bytes (not words or long words) to be checked. The lengthInBytes must be page-aligned (32-word or 128-byte).

5.5.9 flash_verify_erase_sector

This API is used for checking if specified IFR0 sectors are in the erased state.

Prototype

```
status_t flash_verify_erase_sector (flash_config_t *config, FMU_Type *base, uint32_t start, uint32_t
lengthInBytes);
```

Parameters

Table 49. flash_verify_erase_sector parameters

Parameter	Description
Config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1
Start	The starting address of required flash memory to be checked. The starting address must be sector-aligned (8192-byte).
lengthInBytes	The length, given in bytes (not words or long words) to be checked. The lengthInBytes must be sector-aligned (8192-byte).

5.5.10 flash_read_into_misr

This API is used for generating a signature based on the contents of the selected flash memory using an embedded MISR.

Prototype

```
status_t flash_read_into_misr (flash_config_t *config, FMU_Type *base, uint32_t start, uint32_t end,
uint32_t *seed, uint32_t *signature);
```

Parameters

Table 50. flash_read_into_misr parameters

Parameter	Description
Config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1
Start	The starting address of the selected flash region . The start address must be page-aligned (32-word or 128-byte).
End	The ending address of the selected flash region. Ending address must be flash phrase aligned (4-word or 16-byte) and the last phrase in a page.
Seed	MISR seed
Signature	Pointer to buffer expecting signature after successful generation

5.5.11 ifr_verify_erase_phrase

This API is used for checking if specified IFR0 phrases are in the erased state.

Prototype

```
status_t ifr_verify_erase_phrase (flash_config_t *config, FMU_Type *base, uint32_t start, uint32_t
lengthInBytes);
```

Parameters

Table 51. ifr_verify_erase_phrase parameters

Parameter	Description
config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1
Start	The starting address of the required IFR0 memory to be checked. The starting address must be phrase-aligned (4-word or 16-byte).
lengthInBytes	The length, given in bytes (not words or long words) to be checked. The lengthInBytes must be phrase-aligned (4-word or 16-byte).

5.5.12 ifr_verify_erase_page

This API is used for checking if specified IFR0 pages are in the erased state.

Prototype

```
status_t ifr_verify_erase_page (flash_config_t *config, FMU_Type *base, uint32_t start, uint32_t
lengthInBytes);
```

Parameters

Table 52. ifr_verify_erase_page parameters

Parameter	Description
config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1
Start	The starting address of the required flash memory to be checked. The start address must be page-aligned (32-word or 128-byte).
lengthInBytes	The length, given in bytes (not words or long words) to be checked. The lengthInBytes must be page-aligned (32-word or 128-byte).

5.5.13 ifr_verify_erase_sector

This API is used for checking if specified IFR0 sectors are in the erased state.

Prototype

```
status_t ifr_verify_erase_sector (flash_config_t *config, FMU_Type *base, uint32_t start, uint32_t
lengthInBytes);
```

Parameters

Table 53. ifr_verify_erase_sector parameters

Parameter	Description
config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1
Start	The starting address of the required flash memory to be checked. The start address must be sector-aligned(8192-byte).
lengthInBytes	The length, given in bytes (not words or long words) to be checked. The lengthInBytes must be sector-aligned(8192-byte).

5.5.14 ifr_read_into_misr

This API is used for generating a signature based on the contents of the selected IFR0 space using an embedded MISR.

Prototype

```
status_t ifr_read_into_misr (flash_config_t *config, FMU_Type *base, uint32_t start, uint32_t end,
uint32_t *seed, uint32_t *signature);
```

Parameters

Table 54. ifr_read_into_misr parameters

Parameter	Description
Config	Pointer to flash_config_t data structure in memory to store driver runtime state.
FMU base	Base pointer to FMU or RF-FMU1
Start	The starting address of the selected flash region . The start address must be page-aligned (32-word or 128-byte).

Table continues on the next page...

Table 54. ifr_read_into_misr parameters (continued)

Parameter	Description
End	The ending address of the selected flash region. Ending address must be flash phrase aligned (4-word or 16-byte) and the last phrase in a page.
Seed	MISR seed
Signature	Pointer to buffer expecting signature after successful generation

5.5.15 flash_get_property

This API returns the required flash property, which includes base address, block size, and other options.

NOTE

The API returns the status kStatus_FLASH_Unknown, which may appear to indicate a failure. However, this status does not mean the API has failed. The result of the operation is provided through the value parameter.

Prototype

```
status_t flash_get_property (flash_config_t *config, flash_property_tag_t whichProperty, uint32_t
*value);
```

Parameters

Table 55. flash_get_property parameters

Parameter	Description
config	Pointer to flash_config_t data structure in memory to store driver runtime state.
whichProperty	The required property from the list of properties.
Value	Property value return to value pointer

Table 56. Property definition

Property Definition	Value
FMU_SectorSize	0x00
FMU_TotalSize	0x01
FMU_BlockSize	0x02
FMU_BlockCount	0x03
FMU_BlockBaseAddr	0x04
RF-FMU1_TotalSize	0x11
RF-FMU1_BlockSize	0x12
RF-FMU1_BlockCount	0x13
RF-FMU1_BlockBaseAddr	0x14

5.5.16 Flash API status code

Table 57. Flash API status code

Status	Code	Description
kStatus_FLASH_Success	0	The flash operation is successful
kStatus_FLASH_InvalidArgument	4	Invalid argument detected during executing a FLASH API.
kStatus_FLASH_SizeError	100	Invalid size detected during executing a FLASH API.
kStatus_FLASH_AlignmentError	101	Alignment error detected during executing a FLASH API.
kStatus_FLASH_AddressError	102	Address error detected during executing a FLASH API.
kStatus_FLASH_AccessError	103	Access error detected during executing a FLASH API.
kStatus_FLASH_ProtectionViolation	104	Command Protection Violation Flag which indicates an attempt was made to modify a protected area of flash memory during a command operation.
kStatus_FLASH_CommandFailure	105	Command failure detected during executing a FLASH API.
kStatus_FLASH_UnknownProperty	106	Unknown property for flash_get_property API.
kStatus_FLASH_EraseKeyError	107	Incorrect EraseKey for flash_erase API.
kStatus_FLASH_CommandAborOption	121	Operation is aborted

5.6 runBootloader API

This API allows to enter the bootloader from the application within the ROM API.

Prototype

```
void (*runBootloader)(uint32_t *arg);
```

Table 58. runBootloader parameter

Parameter	Description
arg	arg Pointer to structure user_app_boot_invoke_option_t

The input argument to the function is a pointer to a 32-bit value which controls how to enter the bootloader:

Table 59.

Bit	Meaning
[0:7]	Tag Must be 0xEB
[8:11]	Mode • 0 = MasterBoot • 1 = ISP Boot
[12:16]	Boot Interface

Table continues on the next page...

Table 59. (continued)

Bit	Meaning
	In ISP Boot mode selects Instance of the peripheral to be used • 0: UART • 1: I2C • 2: SPI • 3: CAN • 15: Autodetect
[17:20]	Instance of the peripheral to be used. Not supported on this device, as peripheral instances are hardwired.
[21:23]	Image index for the master boot mode • 0xE: OEM image • 0xA: ABASE image
[24:31]	Reserved Must be 0

The function stores the value of the input argument into a register surviving a reset and resets the device, which causes bootloader to run. Bootloader examines the argument value stored in the register and decides what to do:

- ISP Boot mode causes ISP path to be entered like the ISP pin was asserted.
- Master Boot mode causes booting the selected image.

Chapter 6
ROM ISP

6.1 Overview

ROM bootloader provides in-system programming utility that operates over a serial connection on the MCUs. It enables quick and easy programming of MCUs through the entire product lifecycle, including application development, final product manufacturing, and beyond. Host-side command line tool is available to communicate with the bootloader. Users can utilize host tools to upload/download application code and do manufacturing via the bootloader.

When ROM bootloader enters ISP mode it auto-detects activity on the LPI2C/LPSPI/LPUART or CAN interface, and selects the appropriate interface once a properly formed frame is received. If an invalid frame is received, the data is discarded and scanning resumes. See LPUART ISP, LPI2C ISP, LPSPI ISP, and CAN ISP for details.

One method to make ROM bootloader go to ISP path is by asserting the BOOT_CONFIG pin (PTA4) since BOOT_CONFIG pin is enabled by default in User IFR (IFR0). See ROM bootloader configuration for details of config BOOT_CONFIG pin. In addition, ROM bootloader will go to ISP path when no boot image is found, PC and SP are not valid and some other boot failure conditions. The runBootloader API can also be used to trigger the ISP mode.

6.2 Available peripherals

Table below shows the peripheral instances and pin assignments used by ISP.

Table 60. Peripheral instances and pin assignments used by ISP

Peripheral instances	Pin name	Pin assignment
LPUART1	LPUART1_TX	PTC3
	LPUART1_RX	PTC2
LPSPI1	LPSPI1_PCS0	PTB0
	LPSPI1_SIN	PTB1
	LPSPI1_SCK	PTB2
	LPSPI1_SOUT	PTB3
CAN0	CAN0_TX	PTC4
	CAN0_RX	PTC5
LPI2C1	LPI2C1_SCL	PTB5
	LPI2C1_SDA	PTB4
Boot pin	BOOT_CONFIG	PTA4

6.3 Available ISP commands

ISP commands availability is controlled by lifecycle. Table below shows the ISP commands available for each lifecycle. Details of several commands can be found in sections below.

Table 61. Available ISP commands for different lifecycle

Command	Description	At OEM Open	After OEM Open
Reset	Reset the device	Available	Available
get-property <tag>	Query about various properties and settings	Available	Available
set-property <tag> <value>	Change properties or options in ROM Bootloader	Available	Available
receive-sb-file <file>	Receive a file in Secure Binary (SB) format	Available	Available
flash-erase-region	Erase one or more sectors of the flash memory	Available	NA
flash-erase-all [<memoryID>]	Erase the entire flash memory specified by memoryID	Available	NA
read-memory <addr> <byte_count> [<file>]	Read memory at specified address	Available	NA
write-memory <addr> [<file> | {{<hex-data>}}]	Write memory at specified address from file or string of hex values Note: When write to SRAM, make sure the length of file or hex-data is 4-byte aligned	Available	NA
fill-memory <addr> <byte_count> <pattern> [word | short | byte]	Fill memory with pattern; pattern size can be word(default), short or byte	Available	NA
fuse-program <index> [<file> | {{<hex-data>}}]	Program fuse at the specified index from file or string of hex values	Available	NA
fuse-read <index> <byte_count> [<file>]	Read fuse from the specified index	Available	NA
Execute <address> <arg> <stackpointer>	Jumps to code at the provided address and does not return to the ROM bootloader	Available	NA

6.3.1 Available properties

Table below shows the available properties (tag) for this device.

Table 62. Supported properties in GetProperty and SetProperty

Name	Writable	Tag value	Size in Bytes	Description
CurrentVersion	no	1	4	The current bootloader version.
AvailablePeripherals	no	2	4	The set of peripherals supported on this chip.
FlashStartAddress	no	3	4	Start address of program flash.
FlashSizeInBytes	no	4	4	Program flash size in bytes.
FlashSectorSize	no	5	4	The size of one sector of program flash in bytes.
FlashBlockCount	no	6	4	The number of blocks of the on-chip flash.
AvailableCommands	no	7	4	The set of commands supported by the bootloader.

Table continues on the next page...

Table 62. Supported properties in GetProperty and SetProperty (continued)

Name	Writable	Tag value	Size in Bytes	Description
CRCCheckStatus	no	8	4	The status of the application CRC check.
VerifyErase	yes	10	4	Controls whether the bootloader verifies erase to flash. The VerifyErase feature is enabled by default: • 1 = Enable • 0 = Disable
MaxPacketSize	no	11	4	Maximum supported packet size for the currently active peripheral interface.
ReservedRegions	no	12	4	List of memory regions reserved by the bootloader. Returned as value pairs (<startaddress-ofregion>, <end-addressof-region>).
RAMStartAddress	no	14	4	Start address of RAM
RAMSizeInBytes	no	15	4	RAM size in bytes.
SystemDeviceId	no	16	4	System Device ID
SecurityState	no	17	4	Security status
UniqueDeviceId	no	18	16	Unique device identification
TargetVersion	no	24	4	Target version
IrqNotifierPin	yes	28	4	Enables/disables and configures the IRQ notifier pin feature. Bits 0-7: pin Bits 8-15: port Bit 16: active polarity Bits 17-30: reserved Bits 31: feature enable/disable
BootStatus	no	32	4	Value of Boot Status Register
LoadableFWVersion	no	33	4	ELE loadable firmware version
FuseProgramVoltage	yes	34	4	Control the System LDO VDD Regulator Voltage Level. To program fuse, System LDO VDD regulator level needs to be regulated to Over Drive Voltage (2.5 V). The default System LDD VDO Regulator Voltage Level is regulated to Normal Voltage (1.8 V). 0 = System LDO VDO Regulator Voltage Level is related to Normal Voltage (1.8 V) 1 = System LDO VDD regulator level is regulated to Over Drive Voltage (2.5 V)

Table continues on the next page...

Name	Writable	Tag value	Size in Bytes	Description

6.3.2 Commands with limitation

Command flash-erase-all can have memoryID as a parameter, if no ID is specified or 0 is specified, the entire Program Flash will be erased. If 2 is specified as memoryID, the entire radio Program Flash will be erased. If 3 is specified as memoryID, the entire radio User IFR (IFR0) will be erased. See Table 83 for details about memory ID.

6.3.3 receive-sb-file

receive-sb-file command is used to do image update on CM33 flash when security is enforced. It receives a secure binary (SB) file, decrypt, authenticate and program the image to the target memory. CUST_PROD_OEMFW_ENC_SK fuse needs to be programmed correctly to ensure this command can be used successfully. Note that set-property 22 1 needs to be used before receive-sb-file command if the sb file contains "programFuses" command, and that set-property 22 0 needs to be done after receive-sb-file is done.

6.4 ISP protocol

This section explains the general protocol for the packet transfers between the host and the ROM Bootloader. The description includes the transfer of packets for different transactions, such as commands with no data phase and commands with incoming or outgoing data phase. The next section describes various packet types used in a transaction.

Each command sent from the host is replied to with a response command.

Commands may include an optional data phase.

- If the data phase is incoming (from the host to the bootloader), it is part of the original command.
- If the data phase is outgoing (from the bootloader to host), it is part of the response command.

6.4.1 Command with no data phase

The protocol for a command with no data phase contains:

- Command packet (from the host)
- Generic response command packet (to host)

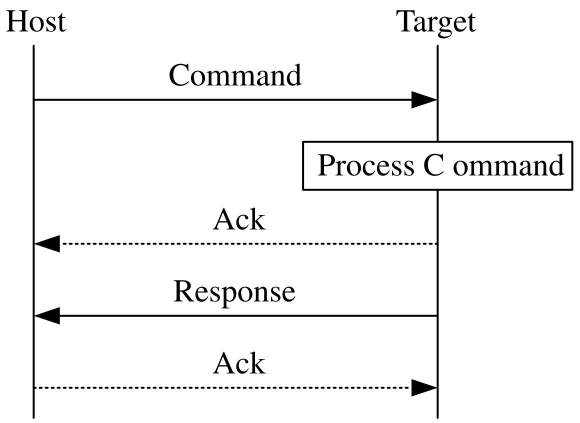


Figure 22. Command with no data phase

NOTE

In these diagrams, the ACK sent in response to a command or a data packet can arrive at any time before, during, or after the command or data packet has processed.

6.4.2 Command with incoming data phase

The protocol for a command with incoming data phase contains:

- Command packet (from host) (kCommandFlag_HasDataPhase set).
- Generic response command packet (to host).
- Incoming data packets (from the host).
- Generic response command packet.

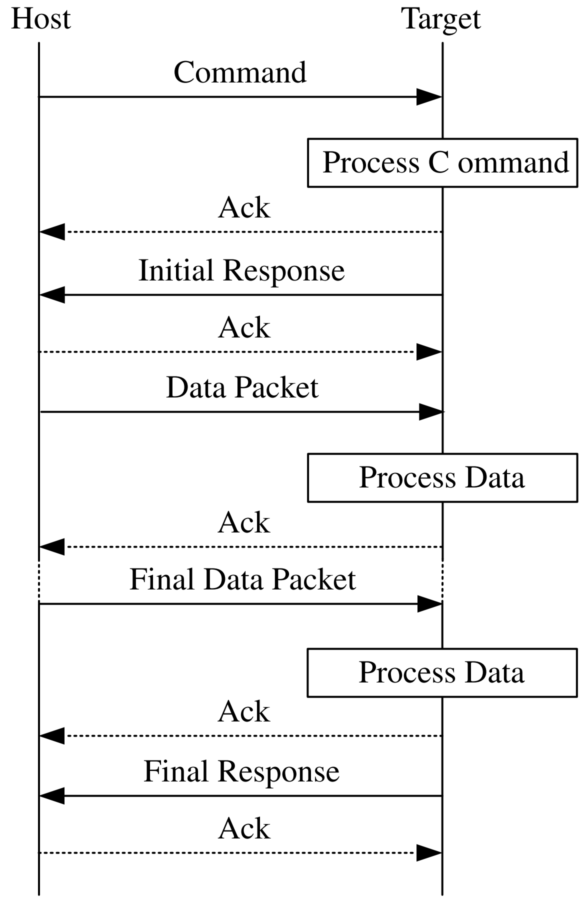


Figure 23. Command with incoming data phase

NOTE

- The host may not send any further packets while it is waiting for the response to a command.
- The data phase is aborted if the Generic Response packet prior to the start of the Data phase does not have a status of kStatus_Success.
- Data phases may be aborted by the receiving side by sending the final GenericResponse early with a status of kStatus_AbortDataPhase. The host may abort the data phase early by sending a zero-length data packet.
- The final Generic Response packet sent after the data phase includes the status of the entire operation.

6.4.3 Command with outgoing data phase

The protocol for a command with an outgoing data phase contains:

- Command packet (from the host).

- ReadMemory Response command packet (to host) (kCommandFlag_HasDataPhase set).
- Outgoing data packets (to host).
- Generic response command packet (to host).

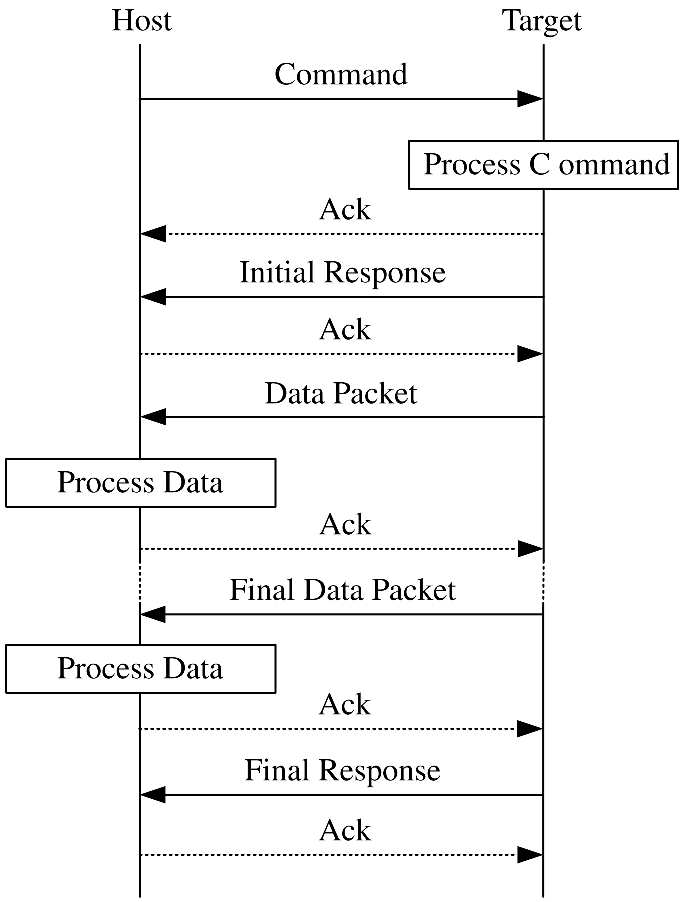


Figure 24. Command with outgoing data phase

NOTE

- The data phase is considered part of the response command for the outgoing data phase sequence. The host may not send any further packets while the host is waiting for the response to a command.
- The data phase is aborted if the ReadMemory Response command packet, prior to the start of the data phase, does not contain the kCommandFlag_HasDataPhase flag. Data phases may be aborted by the host sending the final Generic Response early with a status of kStatus_AbortDataPhase. The sending side may abort the data phase early by sending a zero-length data packet.
- The final Generic Response packet sent after the data phase includes the status of the entire operation.

6.5 ISP packet type

The bootloader device works in slave mode. All data communications are initiated by a host, which is either a PC or an embedded host. The bootloader device is the target, which receives a command or data packet. All data communications between host and target are packetized.

There are six types of packets used:

- Ping packet.

- Ping Response packet.
- Framing packet.
- Command packet.
- Data packet.
- Response packet.

All fields in the packets are in little-endian byte order.

6.5.1 Ping packet

The Ping packet is the first packet sent from a host to the target to establish a connection on the selected peripheral in order to run autobaud detection. The Ping packet can be sent from host to target at any time that the target is expecting a command packet. If the selected peripheral is LPUART, a ping packet must be sent before any other communications. For other serial peripherals, it is optional.

In response to a Ping packet, the target sends a Ping Response packet, discussed in later sections.

Table 63. Ping packet format

Byte #	Value	Name
0	0x5A	start byte
1	0xA6	ping

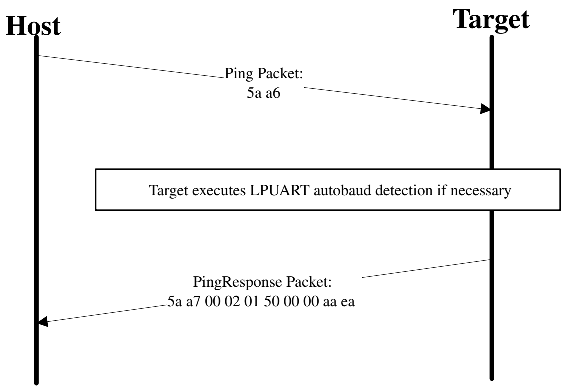

Host
Target
Ping Packet:
5a a6
Target executes LPUART autobaud detection if necessary
PingResponse Packet:
5a a7 00 02 01 50 00 00 aa ea

Figure 25. Ping packet protocol sequence

6.5.2 Ping response packet

The target sends a Ping Response packet back to the host after receiving a Ping packet. If communication is over an LPUART peripheral, the target uses the incoming Ping packet to determine the baud rate before replying with the Ping Response packet. Once the Ping Response packet is received by the host, the connection is established, and the host starts sending commands to the target.

Table 64. ping response packet format

Byte #	Value	Parameter
0	0x5A	Start byte
1	0xA7	Ping response code

Table continues on the next page...

Table 64. ping response packet format (continued)

Byte #	Value	Parameter
2	0x00	Protocol bugfix
3	0x03	Protocol minor
4	0x01	Protocol major
5	0x50	Protocol name = 'P' (0x50)
6	0x00	Options low
7	0x00	Options high
8	0xaa	CRC16 low
9	0xea	CRC16 high

The Ping Response packet can be sent from host to target any time the target expects a command packet. For the LPUART peripheral, it must be sent by the host when a connection is first established, in order to run outbound. For other serial peripherals, it is optional but recommended to determine the serial protocol version. The version number is in the same format as the bootloader version number returned by the GetProperty command.

6.5.3 Framing packet

The framing packet is used for flow control and error detection for the communications links that do not have such features built-in. The framing packet structure sits between the link layer and the command layer. It wraps command and data packets as well.

Every framing packet containing data sent in one direction results in a synchronizing response framing packet in the opposite direction.

The framing packet described in this section is used for serial peripherals including the LPUART, LPI2C, and LPSPI.

Table 65. Framing packet format

Byte #	Value	Parameter	Remark
0	0x5A	Start byte	
1		packetType	
2		length_low	Length is a 16-bit field that specifies the entire command or data packet size in bytes.
3		length_high	
4		crc16_low	This is a 16-bit field. The CRC16 value covers entire framing packet, including the start byte and command or data packets, but does not include the CRC bytes. See the CRC16 algorithm after this table.
5		crc16_high	
6 . . . n		Command or Data packet payload	

A special framing packet that contains only a start byte and a packet type is used for synchronization between the host and target.

Table 66. Special framing packet format

Byte #	Value	Parameter
0	0x5A	Start byte
1	0xA*n*	packetType

The Packet Type field specifies the type of the packet from one of the defined types (below):

Table 67. packetType field

packetType	Name	Description
0xA1	kFramingPacketType_Ack	The previous packet was received successfully; the sending of more packets is allowed.
0xA2	kFramingPacketType_Nak	The previous packet was corrupted and must be re-sent.
0xA3	kFramingPacketType_AckAbort	Data phase is being aborted.
0xA4	kFramingPacketType_Command	The framing packet contains a command packet payload.
0xA5	kFramingPacketType_Data	The framing packet contains a data packet payload.
0xA6	kFramingPacketType_Ping	Sent to verify the other side is alive. Also used for LPUART autobaud.
0xA7	kFramingPacketType_PingResponse	A response to Ping; contains the framing protocol version number and options.

6.5.4 CRC16 algorithm

This section provides the CRC16 algorithm.

The CRC is computed over each byte in the framing packet header, excluding the crc16 field itself, plus all of the payload bytes. The CRC algorithm is the XMODEM variant of CRC-16.

Table 68. Characteristics of the XMODEM variant

Variant	Values
Width	16
Polynomial	0x1021
Init value	0x0000
Reflect in	False
Reflect out	False
xor out	0x0000
Check result	0x31c3

The check result is computed by running the ASCII character sequence "123456789" through the algorithm.

```
uint16_t crc16_update(const uint8_t * src, uint32_t lengthInBytes)
{
   uint32_t crc = 0;
   uint32_t j;
   for (j=0; j < lengthInBytes; ++j)
   {
        uint32_t i;
    uint32_t byte = src[j];
     crc ^= byte << 8;
     for (i = 0; i < 8; ++i)
      {
         uint32_t temp = crc << 1;
         if (crc & 0x8000)
          {
              temp ^= 0x1021;
          }
          crc = temp;
      }
   }
   return crc;
}
```

6.5.5 Command packet

The command packet carries a 32-bit command header and a list of 32-bit parameters.

Table 69. Command packet format (32 bytes)

Command Header (4 bytes)				28 bytes for Parameters (Max 7 parameters)						
Tag	Flags	Rsvd	Param Count	Param1 (32-bit)	Param2 (32-bit)	Param3 (32-bit)	Param4 (32-bit)	Param5 (32-bit)	Param6 (32-bit)	Param7 (32-bit)
Byte 0	Byte 1	Byte 2	Byte 3	-	-	-	-	-	-	-

Table 70. Command header format

Byte #	Command Header Field	Remark
0	Command or Response tag	The command header is 4 bytes long, with these fields.
1	Flags	
2	Reserved. Should be 0x00.	
3	ParameterCount	

The header is followed by 32-bit parameters up to the value of the ParameterCount field specified in the header. Because a command packet is 32 bytes long, only 7 parameters can fit into the command packet.

Command packets are also used by the target to send responses back to the host. As mentioned earlier, command packets and data packets are embedded into framing packets for all the transfers.

Table 71. Command tags[1]

Command Tag	Name
0x01	FlashEraseAll
0x02	FlashEraseRegion
0x03	ReadMemory
0x04	WriteMemory
0x05	FillMemory
0x07	GetProperty
0x08	ReceiveSbFile
0x09	Execute
0x0B	Reset
0x0C	SetProperty
0x14	FuseProgram
0x17	FuseRead

[1] The command tag specifies one of the commands supported by the bootloader. The valid command tags for the bootloader are listed here.

Table 72. Response tags

Response Tag	Name	Remark
0xA0	GenericResponse	The response tag specifies one of the responses the bootloader (target) returns to the host. The valid response tags are listed here.
0xA7	GetPropertyResponse (used for sending responses to GetProperty command only)	
0xA3	ReadMemoryResponse (used for sending responses to ReadMemory command only)	
0xAF	FlashReadOnceResponse (used for sending responses to FlashReadOnce command only)	
0xB0	FlashReadResourceResponse	
0xB5	TrustProvisioningResponse	

Flags: Each command packet contains a Flag byte. Only bit 0 of the flag byte is used. If bit 0 of the flag byte is set to 1, then data packets follow the command sequence. The number of bytes that are transferred in the data phase is determined by a command specific parameter in the parameters array.

ParameterCount: The number of parameters included in the command packet.

Parameters: The parameters are word-length (32 bits). With the default maximum packet size of 32 bytes, a command packet can contain up to 7 parameters.

6.5.6 Response packet

The responses are carried using the same command packet format wrapped with framing packet data. Types of responses include:

- GenericResponse

- GetPropertyResponse
- ReadMemoryResponse
- FlashReadOnceResponse
- FlashReadResourceResponse
- TrustProvisioningResponse

GenericResponse: After the bootloader has processed a command, the bootloader sends a generic response with status and command tag information to the host. The generic response is the last packet in the command protocol sequence. The generic response packet contains the framing packet data and the command packet data (with generic response tag = 0xA0) and a list of parameters (defined in the next section). The parameter count field in the header is always set to 2, for status code and command tag parameters.

Table 73. GenericResponse parameters

Byte #	Parameter	Description
0 - 3	Status code	The Status codes are errors encountered during the execution of a command by the target. If a command succeeds, then a kStatus_Success code is returned.
4 - 7	Command tag	The Command tag parameter identifies the response to the command sent by the host.

GetPropertyResponse: The GetPropertyResponse packet is sent by the target in response to the host query that uses the GetProperty command. The GetPropertyResponse packet contains the framing packet data and the command packet data, with the command/response tag set to a GetPropertyResponse tag value (0xA7).

The parameter count field in the header is set to greater than 1, to always include the status code and one or many property values.

Table 74. GetPropertyResponse parameters

Byte #	Parameter
0 - 3	Status code
4 - 7	Property value
. . .	. . .
	Can be up to maximum 6 property values, limited to the size of the 32-bit command packet and property type.

ReadMemoryResponse: The ReadMemoryResponse packet is sent by the target in response to the host sending a ReadMemory command. The ReadMemoryResponse packet contains the framing packet data and the command packet data, with the command/response tag set to a ReadMemoryResponse tag value (0xA3), the flags field set to kCommandFlag_HasDataPhase (1).

The parameter count set to 2 for the status code and the data byte count parameters shown below.

Table 75. ReadMemoryResponse parameters

Byte #	Parameter	Description
0 - 3	Status code	The status of the associated Read Memory command.
4 - 7	Data byte count	The number of bytes sent in the data phase.

FlashReadOnceResponse:The FlashReadOnceResponse packet is sent by the target in response to the host sending a FlashReadOnce command. The FlashReadOnceResponse packet contains the framing packet data and the command packet data, with the command/response tag set to a FlashReadOnceResponse tag value (0xAF), and the flags field set to 0. The parameter count is set to 2 plus the number of words requested to be read in the FlashReadOnceCommand.

Table 76. FlashReadOnceResponse parameters

Byte #	Parameter
0 – 3	Status Code
4 – 7	Byte count to read
…	…
	Can be up to 20 bytes of requested read data.

FlashReadResourceResponse: The FlashReadResourceResponse packet is sent by the target in response to the host sending a FlashReadResource command. The FlashReadResourceResponse packet contains the framing packet data and the command packet data with the command /response tag set to a FlashReadResource tag value (0xB0), and the flag is set to kCommandFlag_HasDataPhase(1).

TrustProvisioningResponse: The TrustProvisioningResponse packet is sent by the target in response to the host sending a TrustProvisioning command. The TrustProvisioningResponse packet contains the framing packet data and the command packet data, with the command /response tag set to a TrustProvisioning tag value (0xB5), and the flag is set to 0.

6.6 Bootloader command set

All bootloader commands follow the command packet format wrapped by the framing packet as explained in previous sections.

See Available ISP commands for different lifecycle for a list of commands supported by the bootloader.

6.6.1 GetProperty command

The GetProperty command is used to query the bootloader about various properties and settings. Each supported property has a unique 32-bit tag associated with it. The tag occupies the first parameter of the command packet. The target returns a GetPropertyResponse packet with the property values for the property identified with the tag in the GetProperty command.

Properties are the defined units of data that can be accessed with the GetProperty or SetProperty commands. Properties may be read-only or read-write. All read-write properties are 32-bit integers, so they can easily be carried in a command parameter.

The 32-bit property tag is the only parameter required for GetProperty command.

Table 77. Parameters for GetProperty Command

Byte #	Command
0 - 3	Property tag See Available properties for more details.
4 - 7	External Memory Identifier (only applies to get property for external memory)

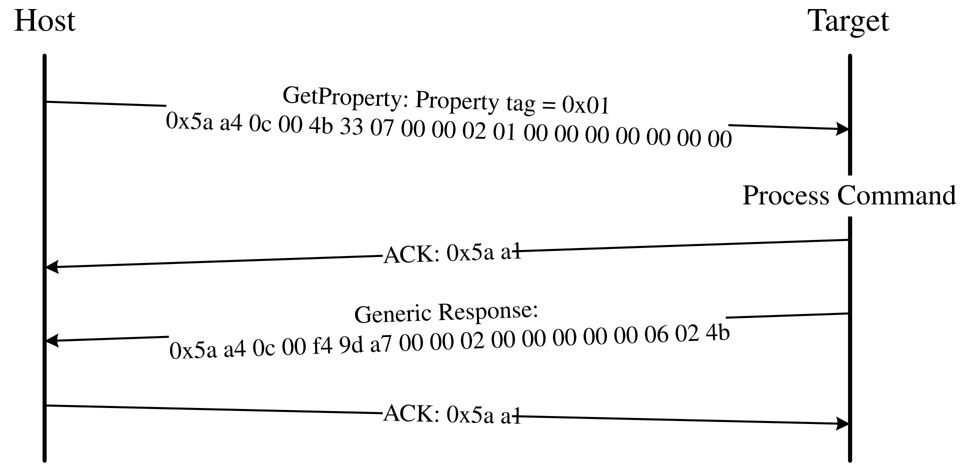

Host
Target
GetProperty: Property tag = 0x01
0x5a a4 0c 00 4b 33 07 00 00 02 01 00 00 00 00 00 00 00
Process Command
ACK: 0x5a a1
Generic Response:
0x5a a4 0c 00 f4 9d a7 00 00 02 00 00 00 00 00 06 02 4b
ACK: 0x5a a1

Figure 26. Protocol Sequence for GetProperty Command

Table 78. GetProperty Command Packet Format (Example)

GetProperty	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x0C 0x00
	crc16	0x4B 0x33
Command packet	commandTag	0x07 – GetProperty
	flags	0x00
	reserved	0x00
	parameterCount	0x02
	propertyTag	0x00000001 - CurrentVersion
	Memory ID	0x00000000 - Internal Flash

The GetProperty command has no data phase.

Response: In response to a GetProperty command, the target sends a

GetPropertyResponse packet with the response tag set to 0xA7. The parameter count indicates the number of parameters sent for the property values, with the first parameter showing status code 0, followed by the property value(s). The next table shows an example of a GetPropertyResponse packet.

Table 79. GetProperty Response Packet Format (Example)

GetPropertyResponse	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x0c 0x00 (12 bytes)
	crc16	0x07 0x7a
Command packet	responseTag	0xA7
	flags	0x00
	reserved	0x00
	parameterCount	0x02
	status	0x00000000
	propertyValue	0x0000014b - CurrentVersion

6.6.2 SetProperty command

The SetProperty command is used to change or alter the values of the properties or options of the bootloader. The command accepts the same property tags used with the GetProperty command. However, only some properties are writable--see Available properties. If an attempt to write a read-only property is made, an error is returned indicating the property is read-only and cannot be changed.

The property tag and the new value to set are the two parameters required for the SetProperty command.

Table 80. Parameters for SetProperty Command

Byte #	Command
0 - 3	Property tag See Available properties for more details.
4 - 7	Property value

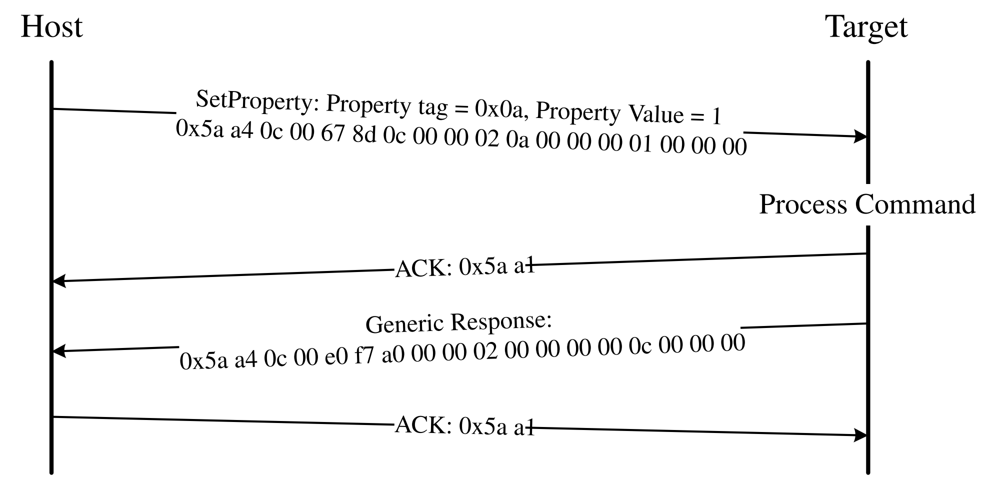

Host
Target
SetProperty: Property tag = 0x0a, Property Value = 1
0x5a a4 0c 00 67 8d 0c 00 00 02 0a 00 00 00 01 00 00 00
Process Command
ACK: 0x5a a1
Generic Response:
0x5a a4 0c 00 e0 f7 a0 00 00 02 00 00 00 00 0c 00 00 00
ACK: 0x5a a1

Figure 27. Parameters for SetProperty Command

Table 81. SetProperty Command Packet Format (Example)

SetProperty	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x0C 0x00
	crc16	0x67 0x8D
Command packet	commandTag	0x0C – SetProperty with property tag 10
	flags	0x00
	reserved	0x00
	parameterCount	0x02
	propertyTag	0x0000000A - VerifyWrites
	propertyValue	0x00000001

The SetProperty command has no data phase.

Response: The target returns a GenericResponse packet with one of following status codes:

Table 82. SetProperty Response Status Codes

Status Code
kStatus_Success
kStatus_ReadOnly
kStatus_UnknownProperty
kStatus_InvalidArgument

6.6.3 FlashEraseAll command

The FlashEraseAll command performs an erase of the entire flash memory. If any flash regions are protected, then the FlashEraseAll command fails and returns an error status code. The Command tag for FlashEraseAll command is 0x01 set in the commandTag field of the command packet.

The FlashEraseAll command requires memory ID. If memory ID is not specified, the internal flash (memory ID =0) will be selected as default.

Table 83. Parameter for FlashEraseAll Command

Byte #	Parameter	
	Memory ID	Descriptions
0-3	0x000	Internal Flash
	0x002	Radio PFlash
	0x003	Radio User IFR

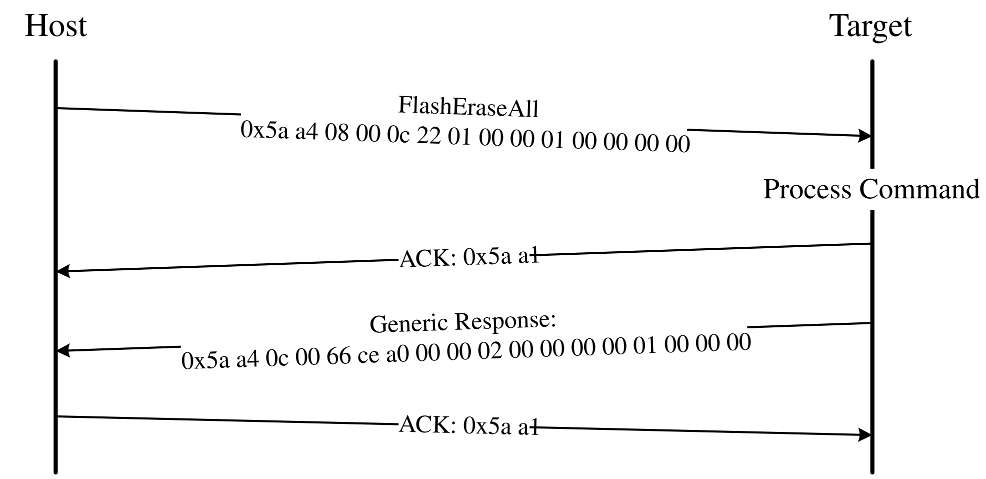

Host
Target
FlashEraseAll
0x5a a4 08 00 0c 22 01 00 00 01 00 00 00 00
Process Command
ACK: 0x5a a1
Generic Response:
0x5a a4 0c 00 66 ce a0 00 00 02 00 00 00 00 01 00 00 00
ACK: 0x5a a1

Figure 28. Protocol Sequence for FlashEraseAll Command

Table 84. FlashEraseAll Command Packet Format (Example)

FlashEraseAll	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x08 0x00
	crc16	0x0C 0x22
Command packet	commandTag	0x01 - FlashEraseAll
	flags	0x00
	reserved	0x00
	parameterCount	0x01
	Memory ID	refer to the above table

The FlashEraseAll command has no data phase.

Response: The target returns a GenericResponse packet with status code either set to kStatus_Success for successful execution of the command or set to an appropriate error status code.

6.6.4 FlashEraseRegion command

The FlashEraseRegion command performs an erase of one or more sectors of the flash memory.

The start address, and number of bytes are the 2 parameters required for the FlashEraseRegion command. The start and byte count parameters must be 32-byte aligned ([3:0] = 0000), or the FlashEraseRegion command fails and returns kStatus_FlashAlignmentError (101). If the region specified does not fit in the flash memory space, the FlashEraseRegion command fails and returns kStatus_FlashAddressError (102). If any part of the region specified is protected, the FlashEraseRegion command fails and returns kStatus_MemoryRangeInvalid (10200).

Table 85. Parameters for FlashEraseRegion Command

Byte #	Parameter
0 - 3	Start address
4 - 7	Byte count
8 - 11	Memory ID

The FlashEraseRegion command has no data phase.

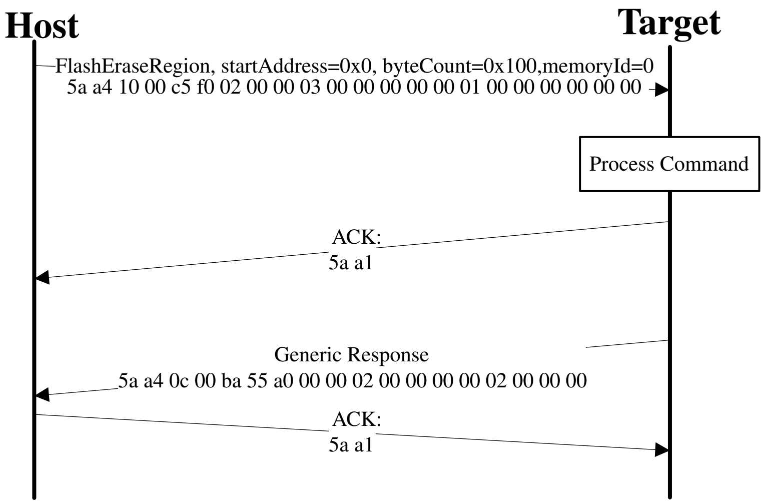

Host
Target
FlashEraseRegion, startAddress=0x0, byteCount=0x100,memoryId=0
5a a4 10 00 c5 f0 02 00 00 03 00 00 00 00 00 01 00 00 00 00 00 00
Process Command
ACK:
5a a1
Generic Response
5a a4 0c 00 ba 55 a0 00 00 02 00 00 00 00 02 00 00 00
ACK:
5a a1

Figure 29. Protocol Sequence for FlashEraseRegion Command

Response: The target returns a GenericResponse packet with one of following error status codes.

Table 86. FlashEraseRegion Response Status Codes

Status Code
kStatus_Success (0)
kStatus_MemoryRangeInvalid (10200)
kStatus_FlashAlignmentError (101)
kStatus_FlashAddressError (102)
kStatus_FlashAccessError (103)
kStatus_FlashCommandFailure (105)

6.6.5 ReadMemory command

The ReadMemory command returns the contents of memory at the given address, for a specified number of bytes. This command can read any region of memory accessible by the CPU and not protected by security.

The start address, and number of bytes are the two parameters required for ReadMemory command. The memory ID is optional. Internal memory will be selected as default if memory ID is not specified.

Table 87. Parameters for read memory command

Byte	Parameter	Description
0-3	Start address	Start address of memory to read from
4-7	Byte count	Number of bytes to read and return to caller
8-11	Memory ID	Internal or external memory Identifier

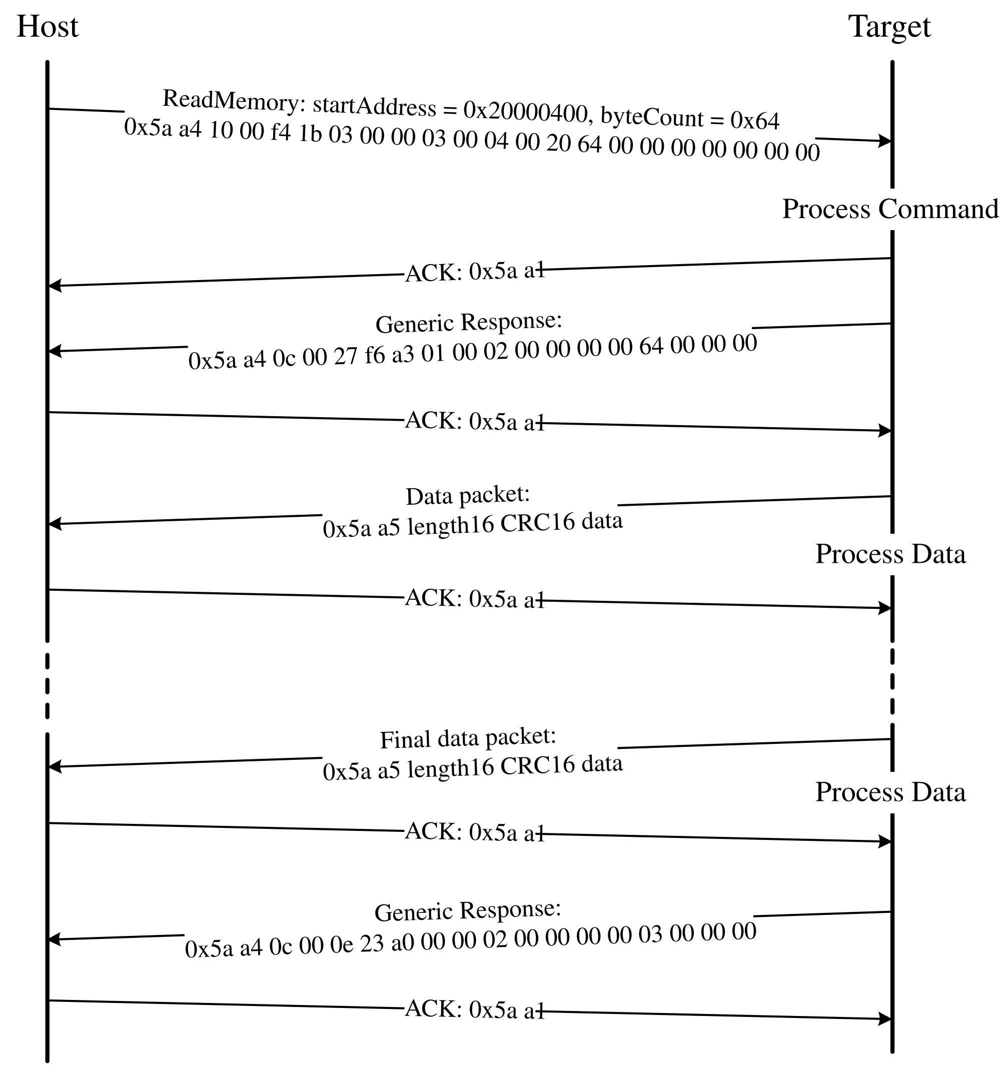

Host
Target
ReadMemory: startAddress = 0x20000400, byteCount = 0x64
0x5a a4 10 00 f4 1b 03 00 00 03 00 04 00 20 64 00 00 00 00 00 00 00
Process Command
ACK: 0x5a a1
Generic Response:
0x5a a4 0c 00 27 f6 a3 01 00 02 00 00 00 00 64 00 00 00
ACK: 0x5a a1
Data packet:
0x5a a5 length16 CRC16 data
Process Data
ACK: 0x5a a1
Final data packet:
0x5a a5 length16 CRC16 data
Process Data
ACK: 0x5a a1
Generic Response:
0x5a a4 0c 00 0e 23 a0 00 00 02 00 00 00 00 03 00 00 00
ACK: 0x5a a1

Figure 30. Command sequence for ReadMemory

Table 88. ReadMemory Command Packet Format (Example)

ReadMemory	Parameter	Value
Framing packet	Start byte	0x5A0xA4,
	packetType	kFramingPacketType_Command
	length	0x10 0x00
	crc16	0xF4 0x1B

Table continues on the next page...

Table 88. ReadMemory Command Packet Format (Example) (continued)

Command packet	commandTag	0x03 - readMemory
	flags	0x00
	reserved	0x00
	parameterCount	0x03
	startAddress	0x20000400
	byteCount	0x00000064
	memoryID	0x0

Data Phase: The ReadMemory command has a data phase. Because the target works in slave mode, the host needs to pull data packets until the number of bytes of data specified in the byteCount parameter of ReadMemory command are received by host.

Response: The target returns a GenericResponse packet with a status code either set to kStatus_Success upon successful execution of the command or set to an appropriate error status code.

6.6.6 WriteMemory command

The WriteMemory command writes data provided in the data phase to a specified range of bytes in memory (flash or RAM). However, if flash protection is enabled, then writes to protected sectors fail.

Special care must be taken when writing to flash.

- First, any flash sector written to must have been previously erased with a FlashEraseAll or FlashEraseRegion command.
- Writing to flash requires the start address to be 16-byte aligned ([3:0] = 0000).
- The byte count is rounded up to a multiple of 4, and trailing bytes are filled with the flash erase pattern (0xff).
- If the VerifyWrites property is set to true, then writes to flash also performs a flash verify program operation.

When writing to RAM, the start address does not need to be aligned, and the data is not padded.

The start address and number of bytes are the 2 parameters required for WriteMemory command. The memory ID is optional. Internal memory will be selected as default if memory ID is not specified.

Table 89. Parameters for WriteMemory Command

Byte #	Command
0 - 3	Start address
4 - 7	Byte count
8 - 11	Memory ID

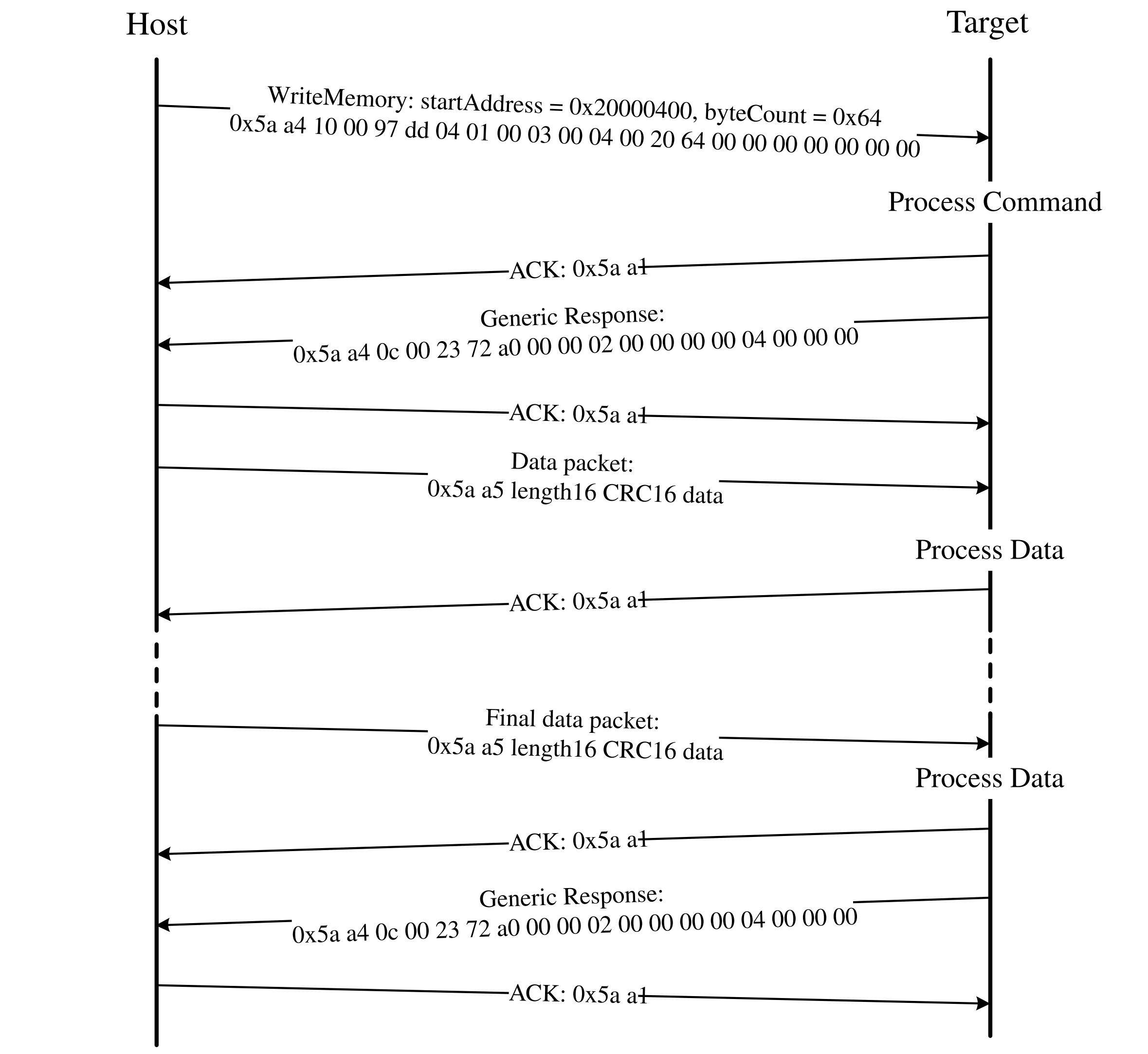

Host
Target
WriteMemory: startAddress = 0x20000400, byteCount = 0x64
0x5a a4 10 00 97 dd 04 01 00 03 00 04 00 20 64 00 00 00 00 00 00
Process Command
ACK: 0x5a a1
Generic Response:
0x5a a4 0c 00 23 72 a0 00 00 02 00 00 00 00 04 00 00 00
ACK: 0x5a a1
Data packet:
0x5a a5 length16 CRC16 data
Process Data
ACK: 0x5a a1
Final data packet:
0x5a a5 length16 CRC16 data
Process Data
ACK: 0x5a a1
Generic Response:
0x5a a4 0c 00 23 72 a0 00 00 02 00 00 00 00 04 00 00 00
ACK: 0x5a a1

Figure 31. Protocol Sequence for WriteMemory Command

Table 90. WriteMemory Command Packet Format (Example)

WriteMemory	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x10 0x00

Table continues on the next page...

Table 90. WriteMemory Command Packet Format (Example) (continued)

	crc16	0x97 0xDD
Command packet	commandTag	0x04 - writeMemory
	flags	0x01
	reserved	0x00
	parameterCount	0x03
	startAddress	0x20000400
	byteCount	0x00000064
	memoryID	0x0

Data Phase: The WriteMemory command has a data phase; the host sends data packets until the number of bytes of data specified in the byteCount parameter of the WriteMemory command are received by the target.

Response: The target returns a GenericResponse packet with a status code set to kStatus_Success upon successful execution of the command, or to an appropriate error status code.

6.6.7 FillMemory command

The FillMemory command fills a range of bytes in memory with a data pattern. It follows the same rules as the WriteMemory command. The difference between FillMemory and WriteMemory is that a data pattern is included in FillMemory command parameter, and there is no data phase for the FillMemory command, while WriteMemory does have a data phase.

Table 91. Parameters for FillMemory Command

Byte #	Command
0 - 3	Start address of memory to fill
4 - 7	Number of bytes to write with the pattern • The start address should be 32-bit aligned. • The number of bytes must be evenly divisible by 4.
8 - 11	32-bit pattern

- To fill with a byte pattern (8-bit), the byte must be replicated 4 times in the 32-bit pattern.
- To fill with a short pattern (16-bit), the short value must be replicated 2 times in the 32-bit pattern.

For example, to fill a byte value with 0xFE, the word pattern is 0xFEFEFEFE; to fill a short value 0x5AFE, the word pattern is 0x5AFE5AFE.

Special care must be taken when writing to flash.

- First, any flash sector written to must have been previously erased with a FlashEraseAll or FlashEraseRegion command.
- Writing to flash requires the start address to be 16-byte aligned ([3:0] = 0000).
- If the VerifyWrites property is set to true, then writes to flash also performs a flash verify program operation.

When writing to RAM, the start address does not need to be aligned, and the data is not padded.

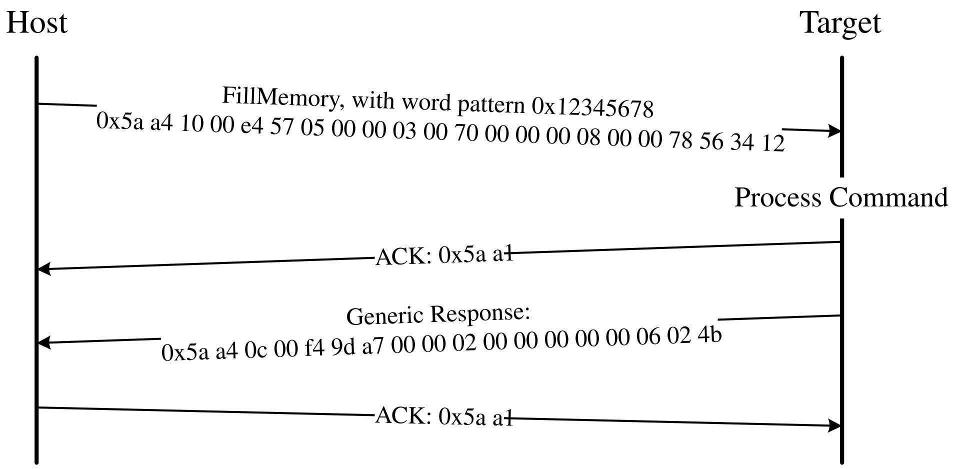

Host
Target
FillMemory, with word pattern 0x12345678
0x5a a4 10 00 e4 57 05 00 00 03 00 70 00 00 00 08 00 00 78 56 34 12
Process Command
ACK: 0x5a a1
Generic Response:
0x5a a4 0c 00 f4 9d a7 00 00 02 00 00 00 00 00 06 02 4b
ACK: 0x5a a1

Figure 32. Protocol Sequence for FillMemory Command

Table 92. FillMemory Command Packet Format (Example)

FillMemory	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x10 0x00
	crc16	0xE4 0x57
Command packet	commandTag	0x05 – FillMemory
	flags	0x00
	Reserved	0x00
	parameterCount	0x03
	startAddress	0x00007000
	byteCount	0x00000800
	patternWord	0x12345678

The FillMemory command has no data phase.

Response: upon successful execution of the command, the target (Kinetis bootloader) returns a GenericResponse packet with a status code set to kStatus_Success, or to an appropriate error status code.

6.6.8 Execute command

The execute command results in the bootloader setting the program counter to the code at the provided jump address, R0 to the provided argument, and a Stack pointer to the provided stack pointer address. Prior to the jump, the system is returned to the reset state.

The Jump address, function argument pointer, and stack pointer are the parameters required for the Execute command. If the stack pointer is set to zero, the called code is responsible for setting the processor stack pointer before using the stack.

Table 93. . Parameters for Execute Command

Byte #	Command
0 - 3	Jump address
4 - 7	Argument word
8 - 11	Stack pointer address

The Execute command has no data phase.

Response: Before executing the Execute command, the target validates the parameters and return a GenericResponse packet with a status code either set to kStatus_Success or an appropriate error status code.

6.6.9 Reset command

The Reset command results in the bootloader resetting the chip.

The Reset command requires no parameters.

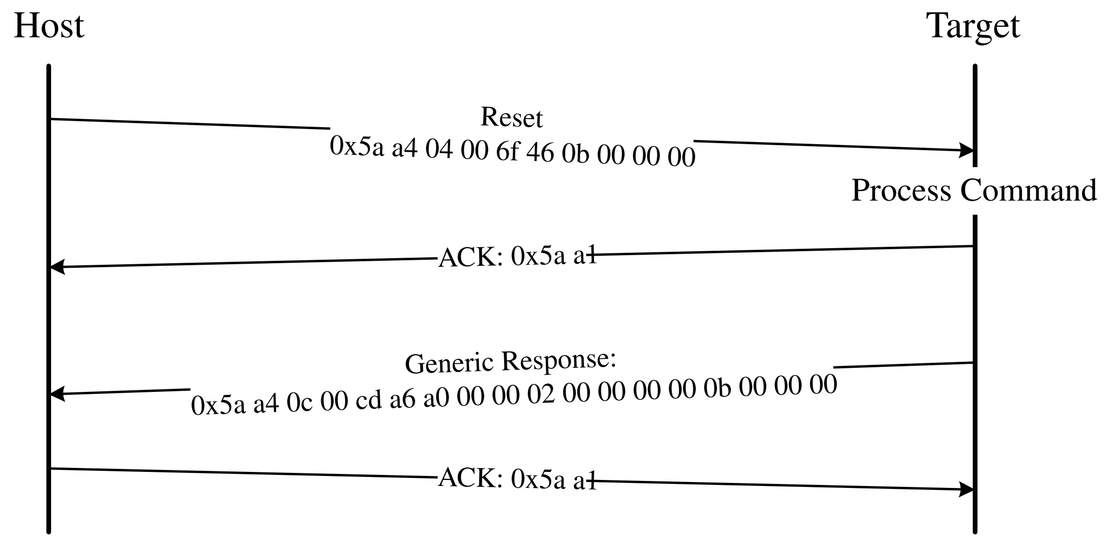

Host
Target
Reset
0x5a a4 04 00 6f 46 0b 00 00 00
Process Command
ACK: 0x5a a1
Generic Response:
0x5a a4 0c 00 cd a6 a0 00 00 02 00 00 00 00 0b 00 00 00
ACK: 0x5a a1

Figure 33. Protocol Sequence for Reset Command

Table 94. Reset Command Packet Format (Example)

Reset	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x04 0x00
	crc16	0x6F 0x46
Command packet	commandTag	0x0B - reset
	flags	0x00
	reserved	0x00
	parameterCount	0x00

The Reset command has no data phase.

Response: The target returns a GenericResponse packet with status code set to kStatus_Success, before resetting the chip.

The reset command can also be used to switch boot from flash after successful flash image provisioning via ROM bootloader. After issuing the reset command, allow 5 seconds for the user application to start running from Flash.

6.6.10 ReceiveSBFile command

The Receive SB File command (ReceiveSbFile) starts the transfer of an SB file to the target. The command only specifies the size in bytes of the SB file that is sent in the data phase. The SB file is processed as it is received by the bootloader. See the Secure boot related sections for more details about the SB file.

Table 95. . Parameters for Receive SB File Command

Byte #	Command
0 – 3	Byte count

Data Phase: The Receive SB file command has a data phase; the host sends data packets until the number of bytes of data specified in the byteCount parameter of the Receive SB File command are received by the target.

Response: The target returns a GenericResponse packet with a status code set to the kStatus_Success upon successful execution of the command or set to an appropriate error code.

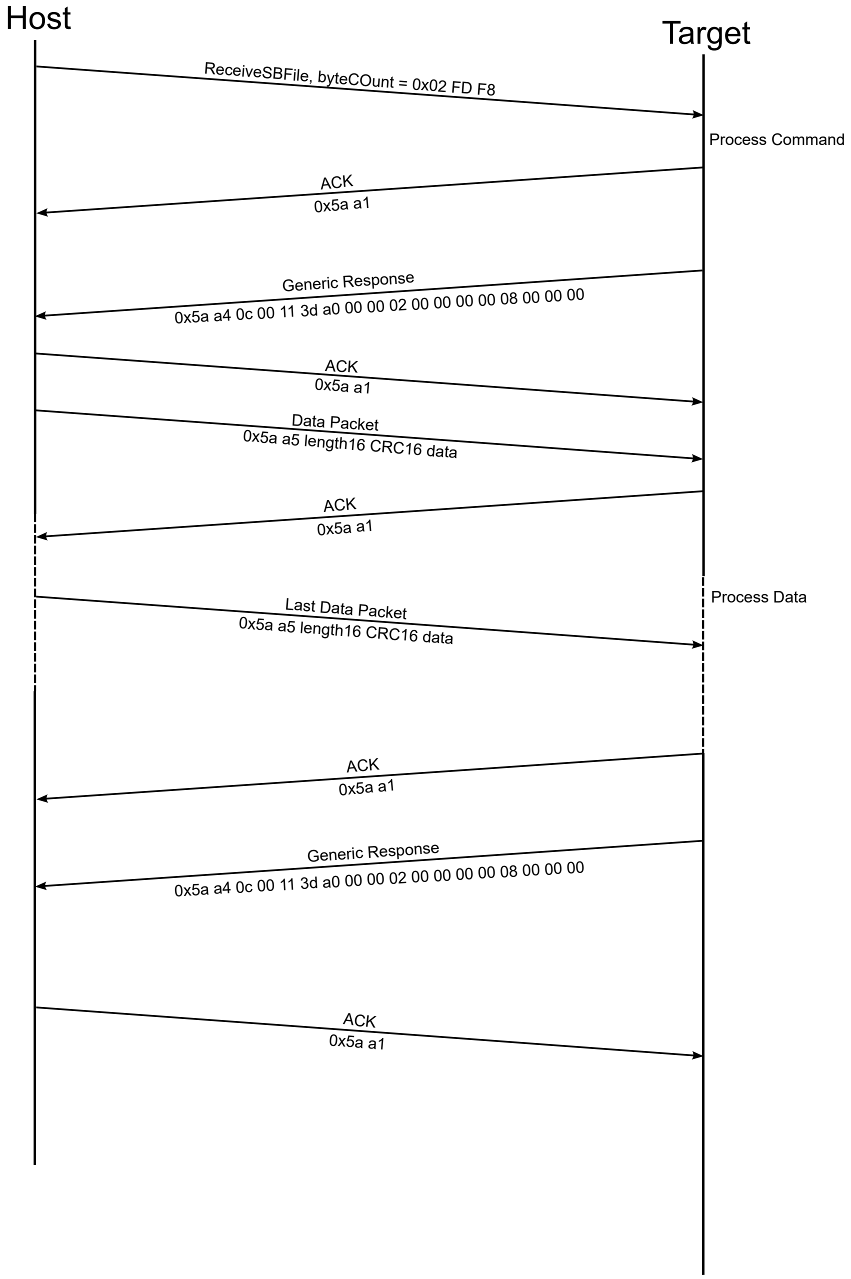

Host
Target
ReceiveSBFile, byteCOunt = 0x02 FD F8
Process Command
ACK
0x5a a1
Generic Response
0x5a a4 0c 00 11 3d a0 00 00 02 00 00 00 00 08 00 00 00
ACK
0x5a a1
Data Packet
0x5a a5 length16 CRC16 data
ACK
0x5a a1
Process Data
Last Data Packet
0x5a a5 length16 CRC16 data
ACK
0x5a a1
Generic Response
0x5a a4 0c 00 11 3d a0 00 00 02 00 00 00 00 08 00 00 00
ACK
0x5a a1

Figure 34. Command sequence for ReceiveSBFile

Table 96. ReceiveSBFile command packet format

ReceiveSBFile	Parameter	Value
Framing packet	startByte	0x5A 0xA4
	packetType	kFramingPacketType_Command
	length	0x08 0x00

Table continues on the next page...

Table 96. ReceiveSBFile command packet format (continued)

ReceiveSBFile	Parameter	Value
	CRC16	0x8b 1e
Command packet	commandTag	0x08 - ReceiveSBFile
	flags	0x00
	reserved	0x00
	parameter count	0x01
	byte count	0x00 02 FD F8
	memoryID	0x00

6.6.11 FuseRead command

The FuseRead command returns the contents of the fuse at the given index, for a specified number of bytes. This command can read any fuse accessible by the CPU and not protected by security.

The index and number of bytes are the two parameters required for FuseRead command. The memory ID is optional. Internal memory will be selected as default if memory ID is not specified.

Table 97. Parameters for FuseRead command

Byte	Parameter	Description
0-3	Index	Index where the fuse is located
4-7	Byte Count	Number of bytes to read and return to caller
8-11	MemoryID	Internal or external memory identifier

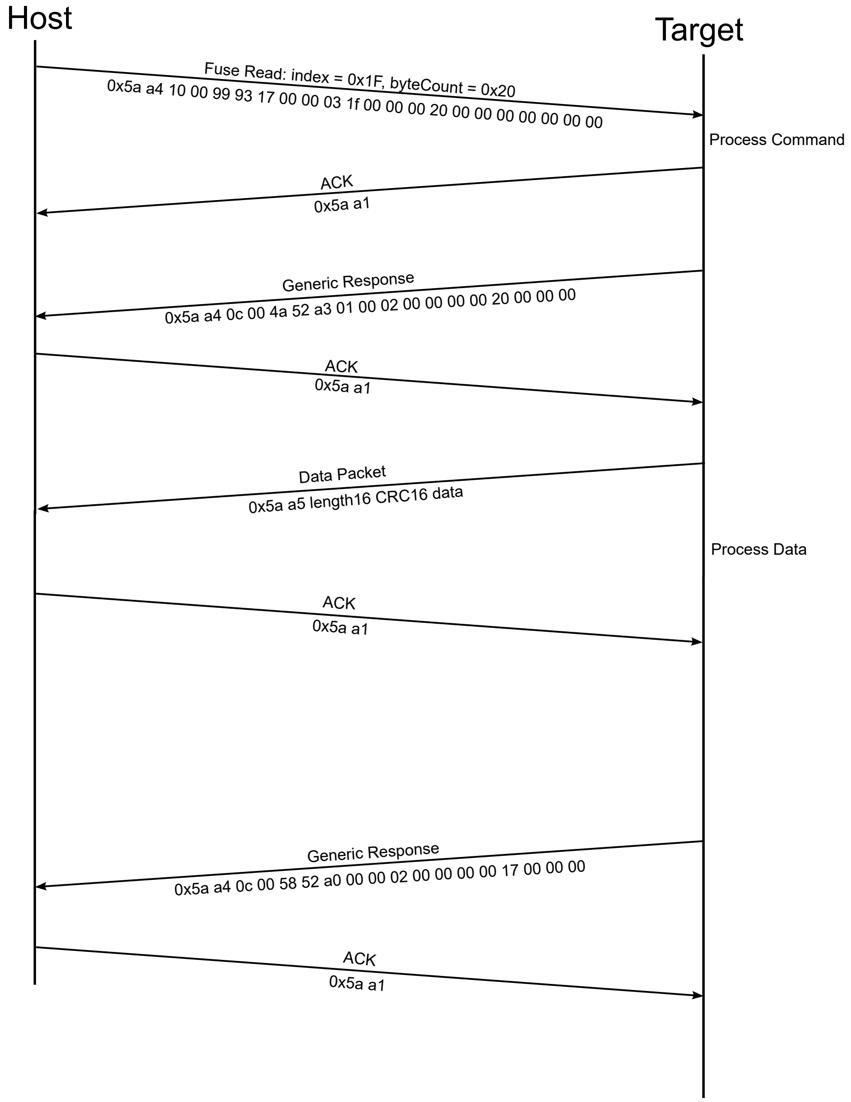

Host
Target
Fuse Read: index = 0x1F, byteCount = 0x20
0x5a a4 10 00 99 93 17 00 00 03 1f 00 00 00 20 00 00 00 00 00 00 00
Process Command
ACK
0x5a a1
Generic Response
0x5a a4 0c 00 4a 52 a3 01 00 02 00 00 00 00 20 00 00 00
ACK
0x5a a1
Data Packet
0x5a a5 length16 CRC16 data
Process Data
ACK
0x5a a1
Generic Response
0x5a a4 0c 00 58 52 a0 00 00 02 00 00 00 00 17 00 00 00
ACK
0x5a a1

Figure 35. Command sequence for FuseRead

Table 98. FuseRead command packet format

FuseRead	Parameter	Value
Framing packet	startByte	0x5A 0xA4
	packetType	kFramingPacketType_Command
	length	0x10 0x00
	CRC16	0x99 0x93
Command packet	commandTag	0x17 - FuseRead
	flags	0x00
	reserved	0x00

Table continues on the next page...

Table 98. FuseRead command packet format (continued)

FuseRead	Parameter	Value
	parameter count	0x03
	index	0x00 00 00 1F
	byte count	0x00 00 00 20
	memoryID	0x00

Data Phase: The FuseRead command has a data phase. Because the target works in slave mode, the host needs to pull data packets until the number of bytes of data specified in the byteCount parameter of FuseRead command are received by host.

Response: The target returns a GenericResponse packet with a status code either set to kStatus_Success upon successful execution of the command or set to an appropriate error status code.

6.6.12 FuseProgram command

The FuseProgram command writes data provided in the data phase to a specified fuse index.

The fuse index and number of bytes are the 2 parameters required for FuseProgram command. The memory ID is optional. Internal memory will be selected as default if memory ID is not specified.

Table 99. Parameters for FuseProgram command

Byte	Parameter	Description
0-3	Index	Index where the fuse is located
4-7	Byte Count	Number of bytes to read and return to caller
8-11	Memory ID	Internal or external memory identifier

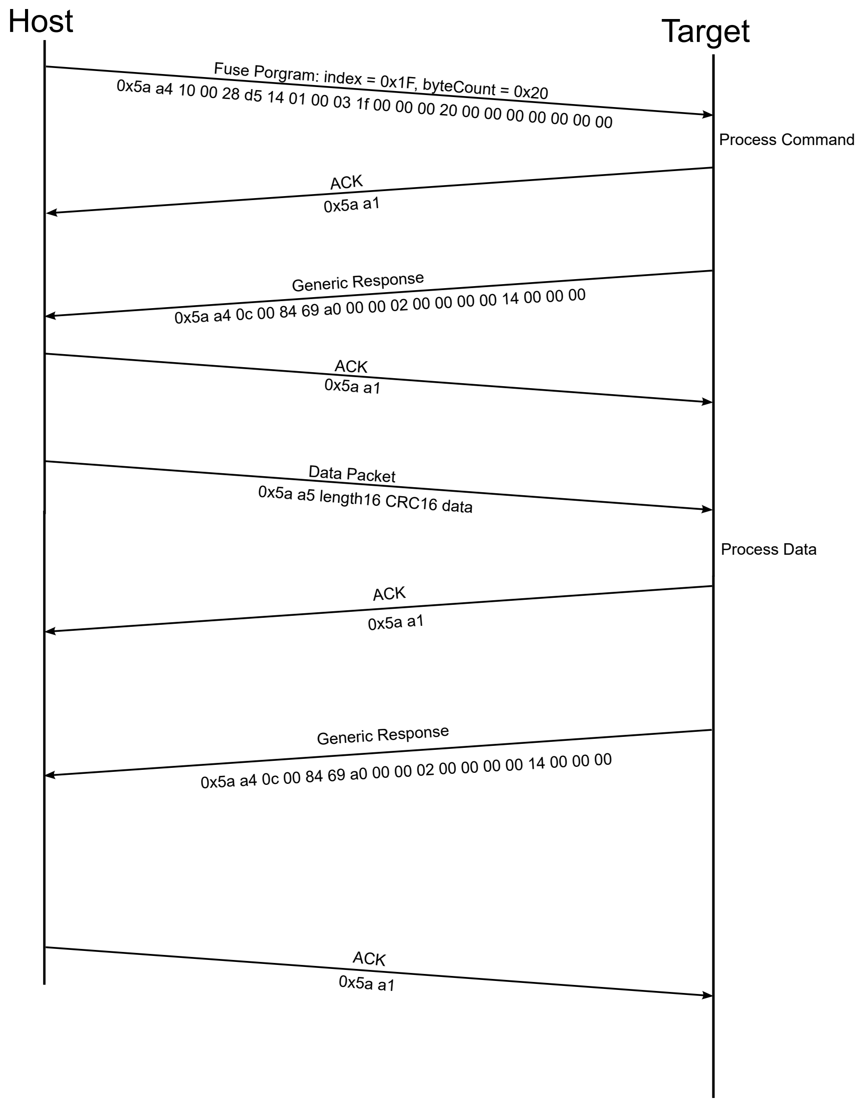

Host
Target
Fuse Porgram: index = 0x1F, byteCount = 0x20
0x5a a4 10 00 28 d5 14 01 00 03 1f 00 00 00 20 00 00 00 00 00 00 00
Process Command
ACK
0x5a a1
Generic Response
0x5a a4 0c 00 84 69 a0 00 00 02 00 00 00 00 14 00 00 00
ACK
0x5a a1
Data Packet
0x5a a5 length16 CRC16 data
Process Data
ACK
0x5a a1
Generic Response
0x5a a4 0c 00 84 69 a0 00 00 02 00 00 00 00 14 00 00 00
ACK
0x5a a1

Figure 36. Command sequence for FuseProgram

Table 100. FuseProgram command packet format

FuseProgram	Parameter	Value
Framing packet	startByte	
	packetType	
	length	
	CRC16	
Command packet	commandTag	0x14 - FusePorgram
	flags	0x00
	reserved	0x00

Table continues on the next page...

Table 100. FuseProgram command packet format (continued)

FuseProgram	Parameter	Value
	parameter count	0x03
	index	0x00 00 00 1F
	byte count	0x00 00 00 20
	memory ID	0x00

Data Phase: The FuseProgram command has a data phase; the host sends data packets until the number of bytes of data specified in the byteCount parameter of the FuseProgram command are received by the target.

Response: The target returns a GenericResponse packet with a status code set to kStatus_Success upon successful execution of the command, or to an appropriate error status code.

6.7 LPUART ISP

The bootloader integrates an autobaud detection algorithm for the LPUART peripheral, thereby providing flexible baud rate choices.

Autobaud feature: If LPUART*n* is used to connect to the bootloader, then the LPUART*n*_RX pin must be kept high and not left floating during the detection phase in order to comply with the autobaud detection algorithm. After the bootloader detects the ping packet (0x5A 0xA6) on LPUART*n*_RX, the bootloader firmware executes the autobaud sequence.

If the baudrate is successfully detected, then the bootloader sends a ping packet response [(0x5A 0xA7), protocol version (4 bytes), protocol version options (2 bytes) and crc16 (2 bytes)] at the detected baudrate. The Kinetis bootloader then enters a loop, waiting for bootloader commands via the LPUART peripheral.

NOTE

The data bytes of the ping packet must be sent continuously (with no more than 80 ms between bytes) in a fixed LPUART transmission mode (8-bit data, no parity bit and 1 stop bit). If the bytes of the ping packet are sent one-by-one with more than 80 ms delay between them, then the autobaud detection algorithm may calculate an incorrect baud rate. In this instance, the autobaud detection state machine should be reset.

Supported baud rates: The baud rate is closely related to the MCU core and system clock frequencies. Typical baud rates supported are 9600, 19200, 38400, 57600, 115200 and 230400.

NOTE

The peripheral baud rate mentioned here is just at ISP packet level, the actual download speed in ISP is lower than the baud rate because of overhead in the ISP protocol and packet handling.

Packet transfer: After autobaud detection succeeds, bootloader communications can take place over the LPUART peripheral. The following flow charts show:

- How the host detects an ACK from the target
- How the host detects a ping response from the target
- How the host detects a command response from the target

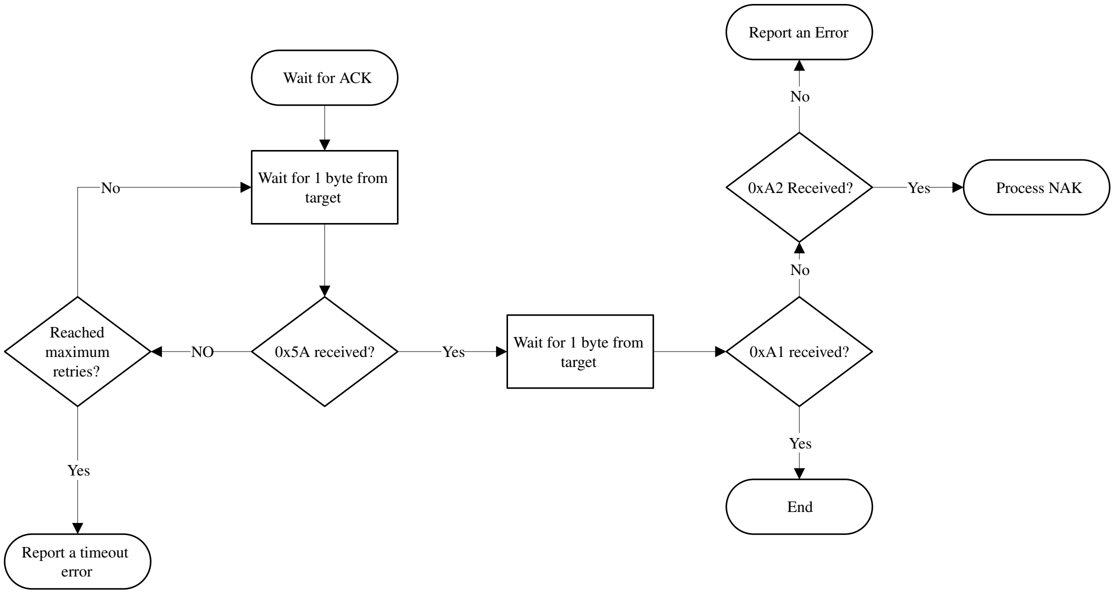


Figure 37. Host reads an ACK from target via LPUART

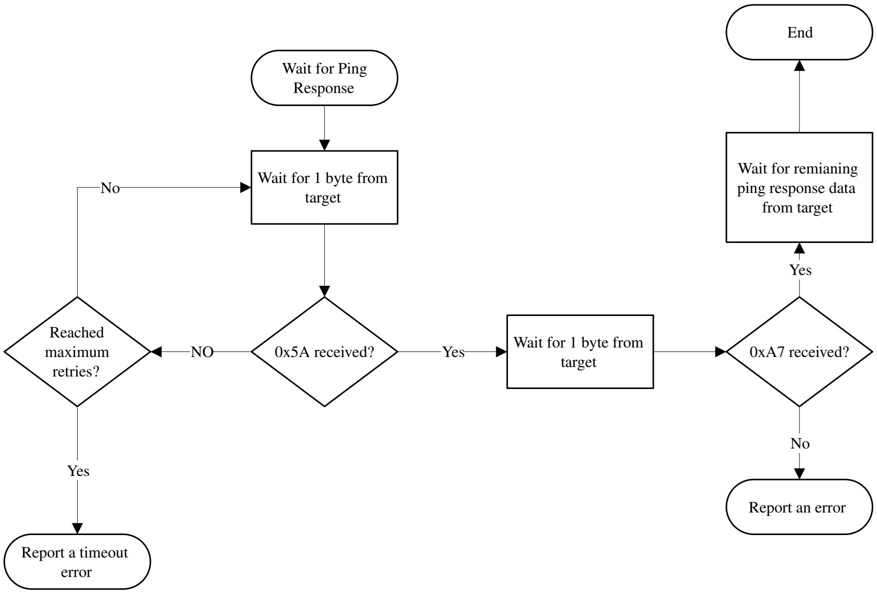


Figure 38. Host reads a ping response from target via LPUART

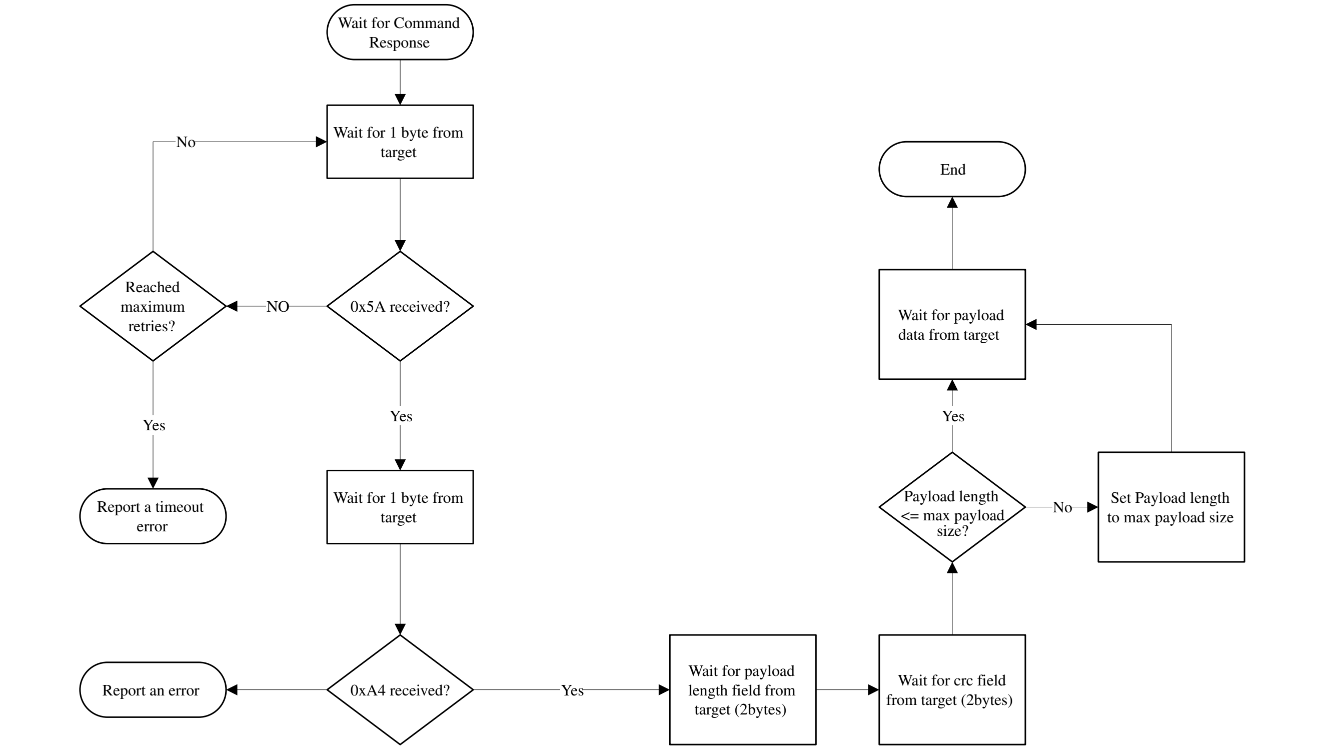

Wait for Command Response
Wait for 1 byte from target
No
0x5A received?
NO
Reached maximum retries?
Yes
Report a timeout error
Yes
Wait for 1 byte from target
0xA4 received?
Report an error
Yes
Wait for payload length field from target (2bytes)
Wait for crc field from target (2bytes)
Payload length <= max payload size?
No
Set Payload length to max payload size
Yes
Wait for payload data from target
End

Figure 39. Host reads a command response from target via LPUART

For information on commands and formats, see ISP packet type

6.8 LPI2C ISP

The bootloader supports In-System Programming or serial boot via the LPI2C peripheral, where the LPI2C peripheral serves as the LPI2C target. The bootloader uses 0x10 as the 7-bit LPI2C target address and supports up to 400 kbit/s speed during transfer. The maximum supported LPI2C baud rate depends on the core clock frequency when the bootloader is running. The typical baud rate is 400 kbit/s with factory settings.

NOTE

The peripheral baud rate mentioned here is just at ISP packet level, the actual download speed in ISP is lower than the baud rate because of overhead in the ISP protocol and packet handling.

The LPI2C peripheral serves as an LPI2C target device, hence each transfer should be started by the host, and each outgoing packet should be fetched by the host as well.

- An incoming packet is sent by the host with a selected LPI2C target address and the direction bit is set to write.
- An outgoing packet is read by the host with a selected LPI2C target address and the direction bit is set as read.
- 0x00 is sent as the response to host if the target is busy with processing or preparing data.

The following charts show the communication flow of the host reading the ACK packet and the corresponding responses from the target.

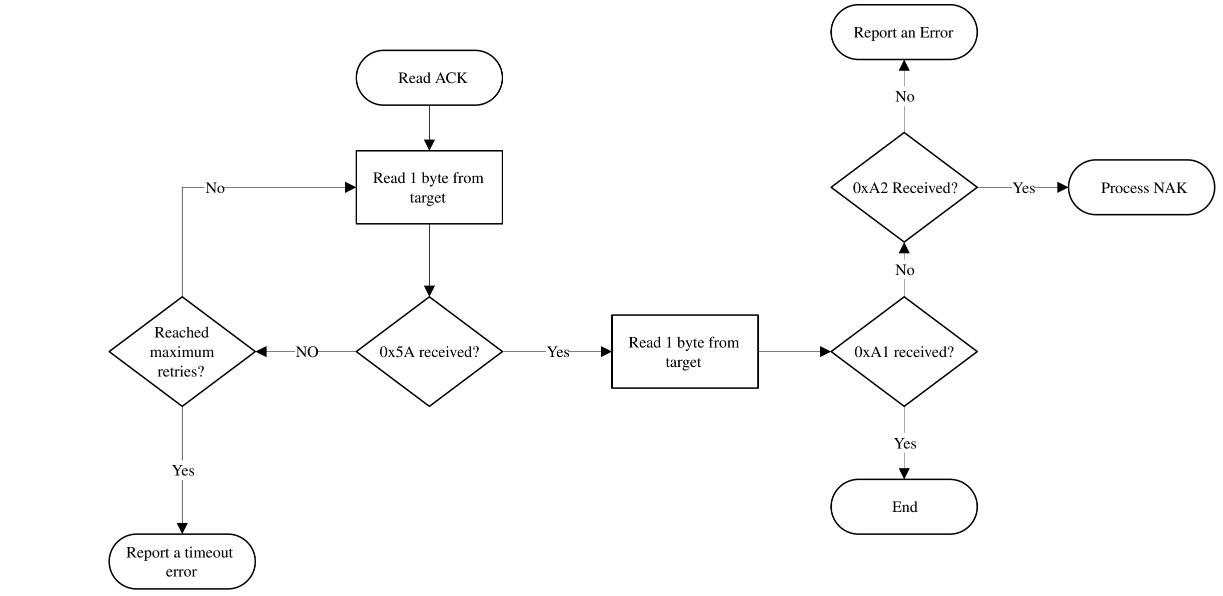


Figure 40. Host reads ACK packet from target via LPI2C

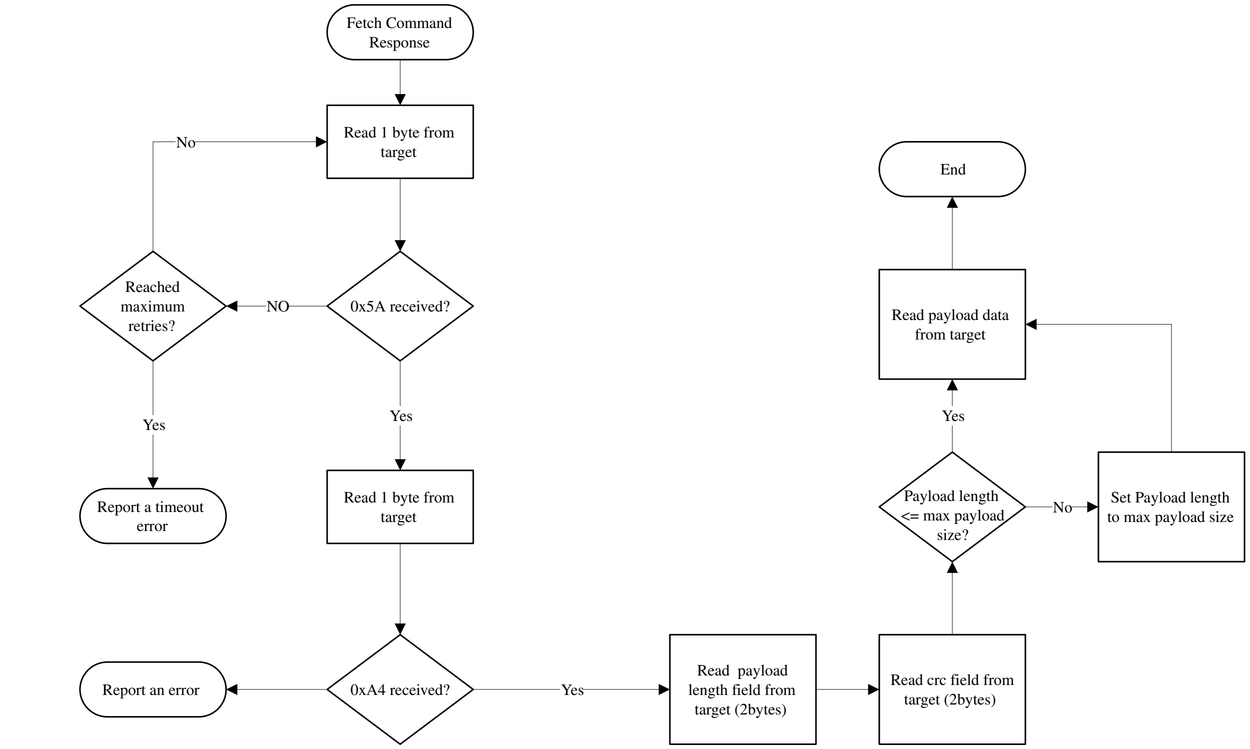

Fetch Command Response
Read 1 byte from target
No
Reached maximum retries?
NO
0x5A received?
Yes
Report a timeout error
Yes
Read 1 byte from target
0xA4 received?
Report an error
Yes
Read payload length field from target (2bytes)
Read crc field from target (2bytes)
Payload length <= max payload size?
No
Set Payload length to max payload size
Yes
Read payload data from target
End

Figure 41. Host reads response from target via LPI2C

For information on commands and formats, see ISP packet type

6.9 LPSPI ISP

The bootloader supports In-System Programming or serial boot via the LPSPI peripheral.

The maximum supported baud rate of the LPSPI depends on the core clock frequency when the bootloader is running. The typical baud rate is 1 MHz with the factory settings. The actual baud rate is up to 4 MHz when the core is running at high-speed boot mode.

The LPSPI peripheral in the bootloader serves as an LPSPI target device. Each transfer, therefore, must be started by the host, and each outgoing packet should be fetched by the host as well.

NOTE

The peripheral baud rate mentioned here is just at ISP packet level, the actual download speed in ISP is lower than the baud rate because of overhead in the ISP protocol and packet handling.

Compared to LPUART and LPI2C peripherals, the transfer on LPSPI is slightly different:

- The host receives 1 byte after it sends out any byte.
- Received bytes should be ignored when the host is sending out bytes to the target
- The host starts reading bytes by sending 0x00s to target
- The byte 0x00 is sent as a response to host if the target is under the following conditions:
  - Processing incoming packet
  - Preparing outgoing data
  - Received invalid data

The following flowcharts show how the host reads a ping response, an ACK and a command response from target via LPSPI without the nIRQ pin enabled.

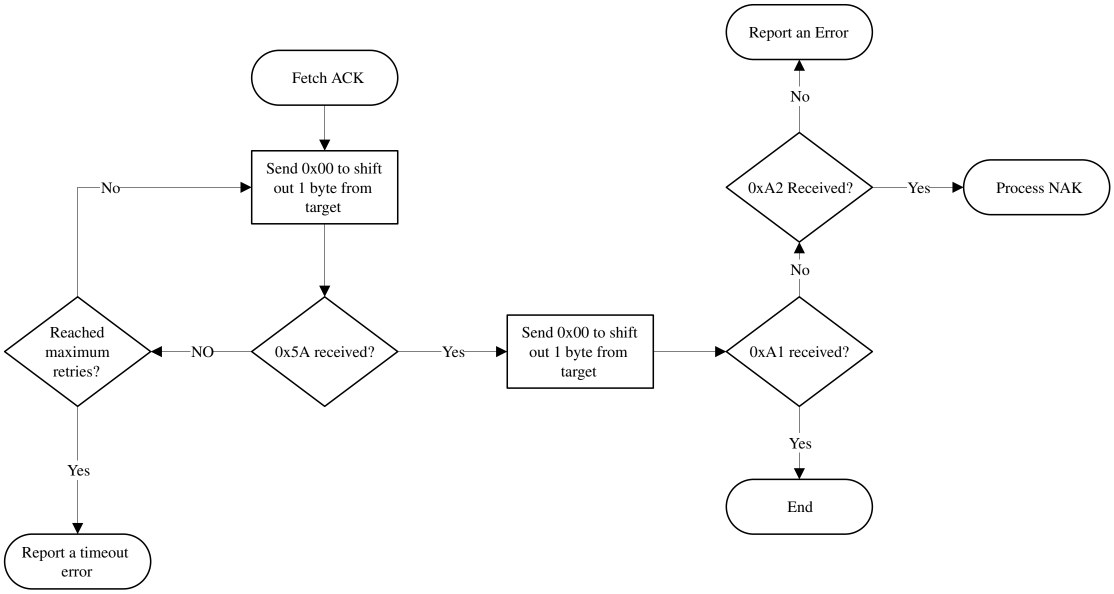


Figure 42. Host reads ACK from target via LPSPI

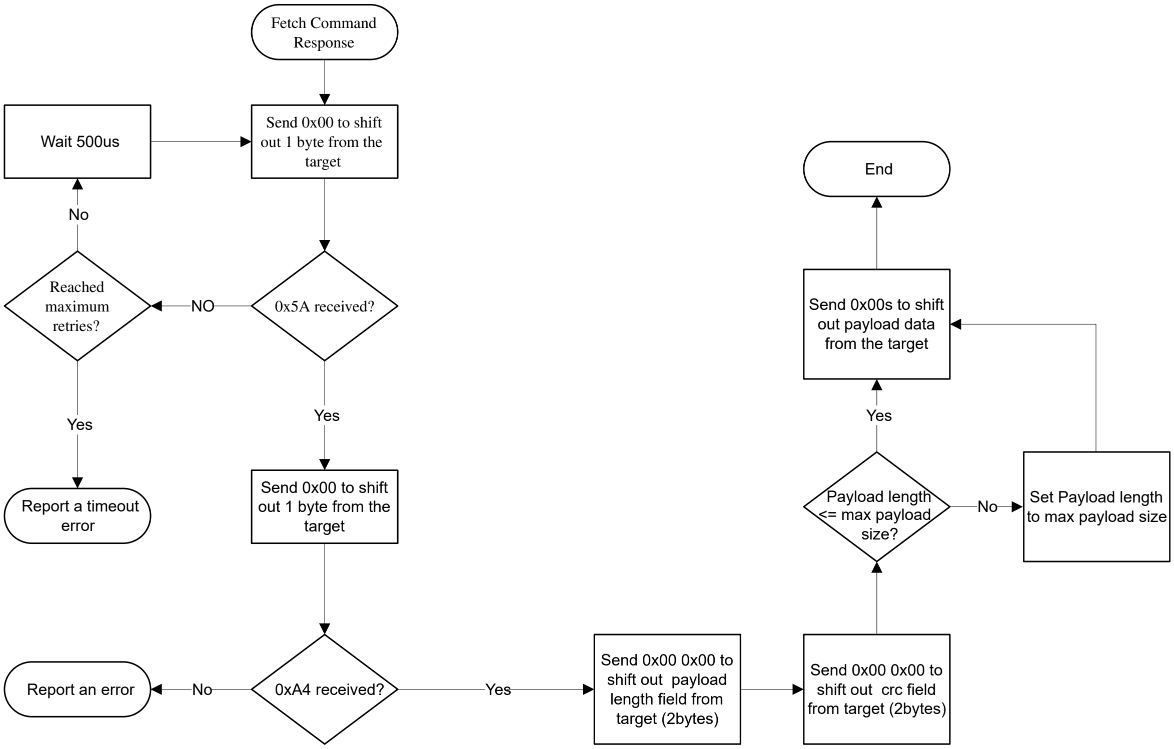

Fetch Command Response
Send 0x00 to shift out 1 byte from the target
Wait 500us
No
Reached maximum retries?
NO
0x5A received?
Yes
Yes
Report a timeout error
Send 0x00 to shift out 1 byte from the target
Report an error
No
0xA4 received?
Yes
Send 0x00 0x00 to shift out payload length field from target (2bytes)
Send 0x00 0x00 to shift out crc field from target (2bytes)
Payload length <= max payload size?
No
Set Payload length to max payload size
Yes
Send 0x00s to shift out payload data from the target
End

Figure 43. Host reads response from target via LPSPI

For information on commands and formats, see ISP packet type

6.10 CAN ISP

The bootloader supports loading data into flash via the FlexCAN peripheral. It supports four predefined speeds on FlexCAN transferring:

- 125 kHz
- 250 kHz
- 500 kHz
- 750 kHz
- 1 MHz

NOTE

The peripheral baud rate mentioned here is just at ISP packet level, the actual download speed in ISP is lower than the baud rate because of overhead in the ISP protocol and packet handling.

In host applications, the user can specify the speed for FlexCAN by providing the speed index as 0 through 4, which represents those 5 speeds (default is 1 MHz).

In bootloader, this supports the auto speed detection feature within supported speeds. In the beginning, the bootloader enters the listen mode with the initial speed (default speed 1 MHz). Once the host starts sending a ping to a specific node, it generates traffic on the CAN bus. Because the bootloader is in a listen mode, it can check if the local node speed is correct by detecting errors. If there is an error, some traffic will be visible, but it may not be on the right speed to see the real data. If this happens, the speed setting changes and checks for errors again. No error means the speed is correct. The settings change back to the normal receiving mode to see if there is a package for this node. It then stays in this speed until another host is using another speed and try to communicate with any node. It repeats the process to detect a right speed before sending host timeout and aborting the request.

The host side should have a reasonable time tolerance during the auto speed detect period. If it sends as timeout, it means there is no response from the specific node, or there is a real error and it needs to report the error to the application.

The flow chart below shows the communication flow for how the host reads the ping packet, ACK, and response from the target.

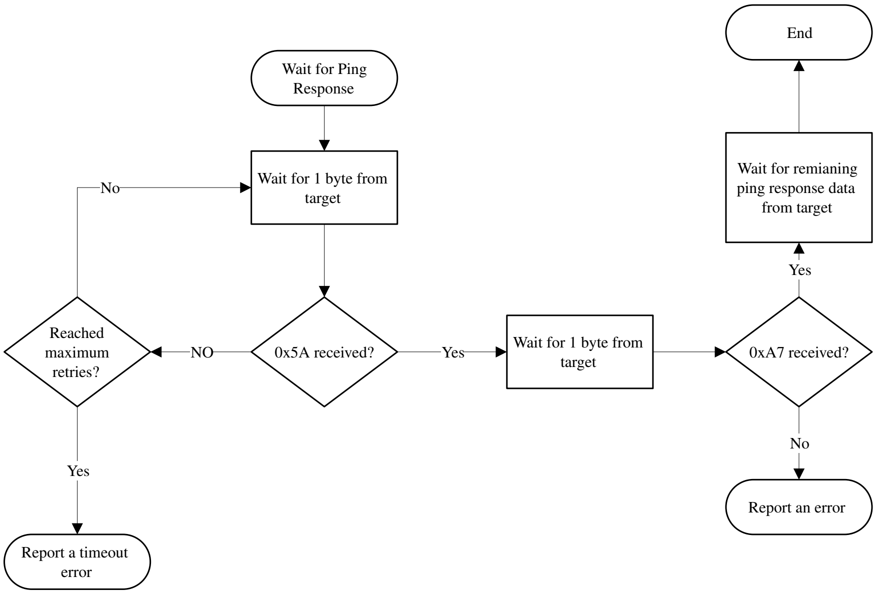


Figure 44. Host reads ping response from target via FlexCAN

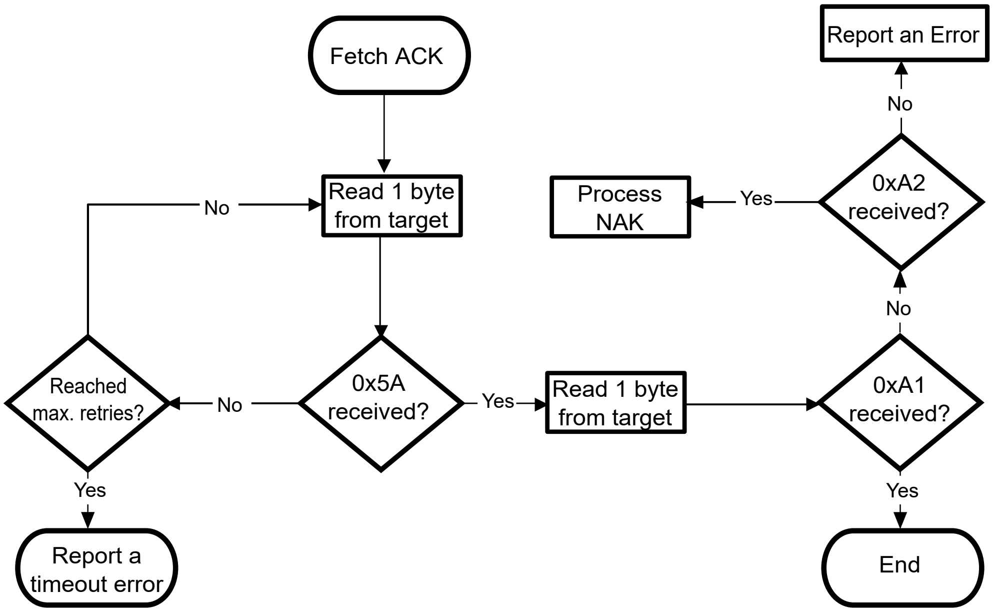


Figure 45. Host reads ACK packet from target via FlexCAN

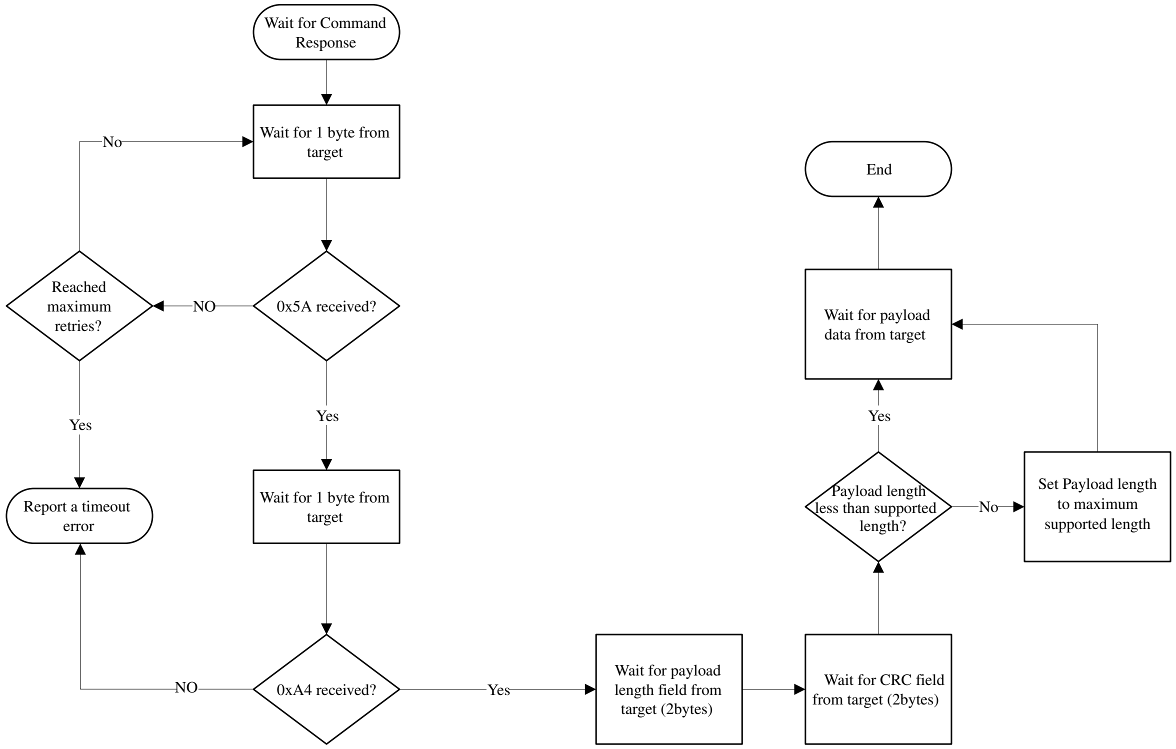

Wait for Command Response
Wait for 1 byte from target
No
Reached maximum retries?
NO
0x5A received?
Yes
Yes
Report a timeout error
Wait for 1 byte from target
0xA4 received?
NO
Yes
Wait for payload length field from target (2bytes)
Wait for CRC field from target (2bytes)
Payload length less than supported length?
No
Set Payload length to maximum supported length
Yes
Wait for payload data from target
End

Figure 46. Host reads command response from target via FlexCAN

Chapter 7
EdgeLock Secure Enclave (ELE)

7.1 Overview

The EdgeLock Secure Enclave (ELE) is an independent security domain that provides security services, which include key management and execution of cryptographic services. The ELE provides a secure environment, which enables applications to execute secure cryptographic services.

7.1.1 Block diagram

A high-level block diagram of the ELE is shown below.

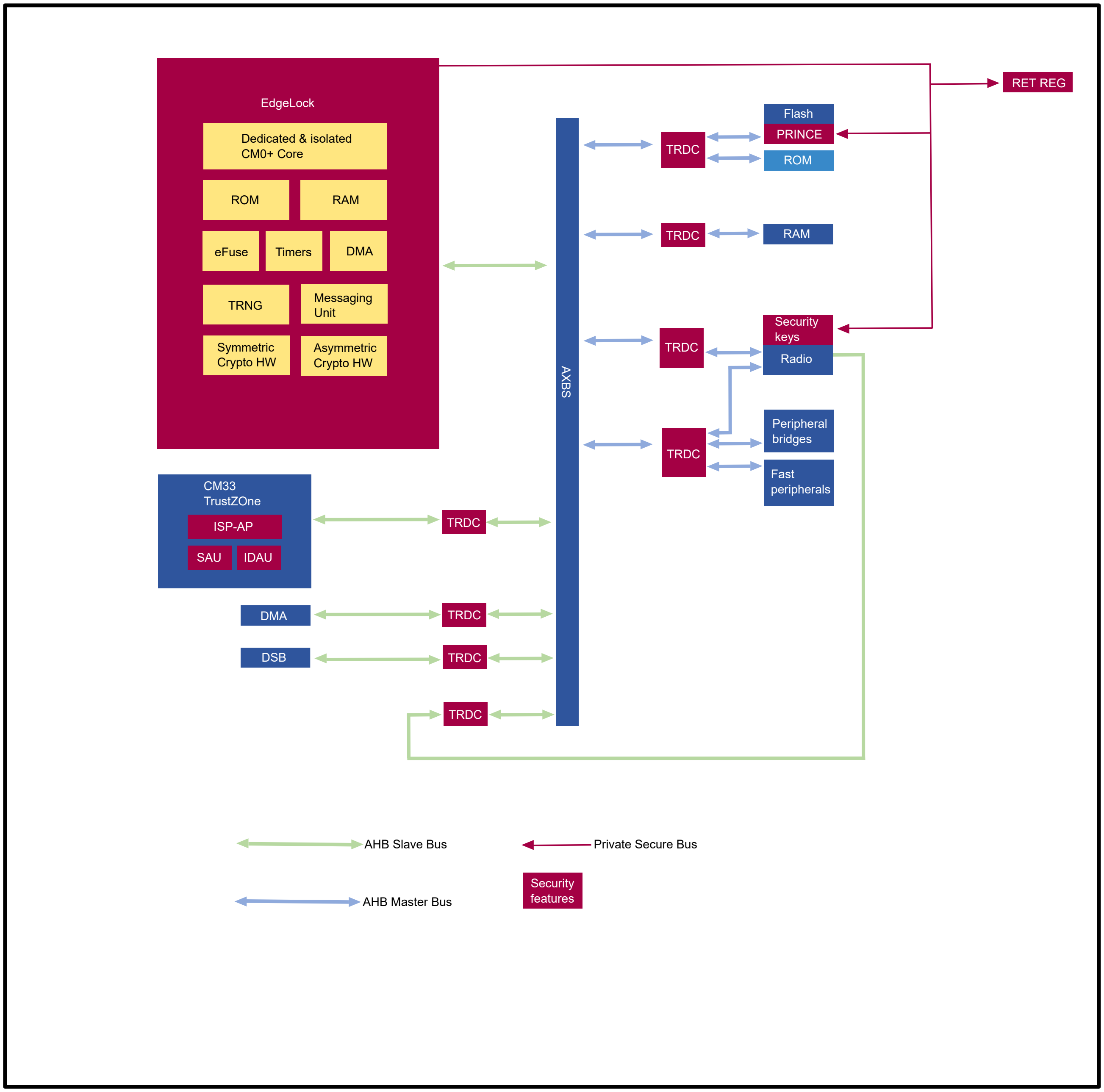


Figure 47. ELE high-level block diagram

7.1.2 Features

The key ELE architecture includes an independent processing element with optimized crypto IP for acceleration. It provides the following security features:

- Secret key generation, storage, and management
  - — Key derivation within ELE and use within ELE for cryptography operation
  - — Facilitating key storage in non-volatile memory like flash, supporting key wrap and protection
  - — Host key derivation

- Secure crypto services during secure boot and runtime software by host
  - Symmetric and asymmetric operation
  - Key wrap and unwrap funcitons
  - Secure DMA
  - Protection of host key in external NVM
  - Host key generation
  - Support of specific application security specification

7.2 Symmetric algorithms

Advanced Encryption Standard (AES) - The AES algorithm is a symmetric block cipher. The algorithm processes 128-bit data blocks using the cipher key. The cipher key can be 128-, 192-, or 256-bits in length. These different versions of the AES algorithm are known as AES-128, AES-192, and AES-256. The amount of processing performed by the AES algorithm is determined by the length of the cipher key and affects the number of rounds used in the calculations.

The AES algorithm was standardized by the U.S. National Institute of Standards and Technology (NIST) in 2001. For more information on the details of the AES algorithm, please refer to Federal Information Processing Standard Publication 197 (FIPS 197) that can be downloaded from the NIST website at https://www.nist.gov.

The ELE natively supports the following AES algorithms along with ECB, CBC, and CRT block cipher modes.

- Advanced Encryption Standard 128 (AES-128)
- Advanced Encryption Standard 192 (AES-192)
- Advanced Encryption Standard 256 (AES-256)

AES algorithm is supported as one go operation only. It means all data are encrypted/decrypted in single step without any interruption. The version of the AES algorithm is selected according to bit length of used key, defined in SYMMETRIC_CONTEXT_INIT command. See Table 101 for the commands supported for symmetric algorithms.

Table 101. Symmetric algorithms commands

Command ID	Commands	Descriptions
0x25	SYMMETRIC_CONTEXT_INIT	Initialize context for symmetric operation
0x23	CIPHER_ONE_GO	Perform symmetric encryption or decryption in one step

7.3 Asymmetric algorithms

Elliptic-Curve Cryptography (ECC) - The ECC algorithms are based on algebraic operations on elliptic curves over finite fields. The algorithms can be used for digital signatures, key agreement, plus encryption and can operate over prime fields or binary fields with varying cipher key lengths.

The ECC curves were originally standardized by the U.S. National Institute of Standards and Technology (NIST) in 2000. For more information on the details of the ECC curves, please refer to Federal Information Processing Standard Publication 186-4 (FIPS 186-4) that can be downloaded from the NIST website at https://www.nist.gov.

ELE natively support the following ECC curves:

- Elliptic Prime Curve 192 (P-192)
- Elliptic Prime Curve 224 (P-224)
- Elliptic Prime Curve 256 (P-256)
- Elliptic Prime Curve 384 (P-384)
- Elliptic Prime Curve 521 (P-521)

- Elliptic Brainpool Curve brainpoolP192r1
- Elliptic Brainpool Curve brainpoolP224r1
- Elliptic Brainpool Curve brainpoolP256r1
- Elliptic Brainpool Curve brainpoolP320r1
- Elliptic Brainpool Curve brainpoolP384r1
- Elliptic Brainpool Curve brainpoolP512r1
- Elliptic Brainpool Curve brainpoolP192t1
- Elliptic Brainpool Curve brainpoolP224t1
- Elliptic Brainpool Curve brainpoolP256t1
- Elliptic Brainpool Curve brainpoolP320t1
- Elliptic Brainpool Curve brainpoolP384t1
- Elliptic Brainpool Curve brainpoolP512t1
- Elliptic Binary Field (Koblitz) Curve sect163k1 (K-163)
- Elliptic Binary Field (Koblitz) Curve sect1283k1 (K-163)
- Curve25519

The following table shows the supported commands for asymmetric algorithms.

Table 102. Asymmetric algorithms commands

Command ID	Commands	Descriptions
0x37	ASYMMETRIC_CONTEXT_INIT	Initialize context for asymmetric (sign/verify) operation
0x3A	ASYMMETRIC_SIGN_DIGEST	Perform asymmetric sign operation
0x3B	ASYMMETRIC_VERIFY_DIGEST	Perform asymmetric verify operation

7.4 Message authentication (MAC) algorithms

The ELE supports following MAC algorithms:

- CMAC with AES
- HMAC with SHA1
- HMAC with SHA2-224
- HMAC with SHA2-256
- HMAC with SHA2-384
- HMAC with SHA2-512

The following table shows the supported commands for message authentication algorithms.

Table 103. Message authentication algorithms commands

Command ID	Commands	Descriptions
0x32	MAC_CONTEX_INIT	Initialize context for message authentication code (MAC) operation

Table continues on the next page...

Table 103. Message authentication algorithms commands (continued)

Command ID	Commands	Descriptions
0x33	MAC_FINISH	Finish block (init/update/finish) MAC operation and returns MAC.
0x34	MAC_INIT	Initialize block (init/update/finish) MAC operation.
0x35	MAC_ONE_GO	Perform MAC operation in one step.
0x36	MAC_UPDATE	Perform block (init/update/finish) MAC operation.
0x84	MAC_IMPORT	Import exported state of MAC context from the MAC context blob into selected MAC context.
0x85	MAC_EXPORT	Export actual state of MAC context out of the ELE into the MAC context blob. The blob is encrypted by device die unique key.

7.5 Hash functions

SHA-1 generates a 160-bit HASH or messages digest. Otherwise SHA-2 or SHA-3 is applicable.

Secure Hash Algorithm 2 (SHA-2) and Secure Hash Algorithm 3 (SHA-3) are cryptographic hash functions. The algorithms process the input information and generate a hash value. This hash value is known as a message digest. The hash value can be 224, 256, 384, or 512 bits in length. These hashes are known as SHA2-224, SHA2-256, SHA2-384, SHA2-512, SHA3-224, SHA3-256, SHA3-384 and SHA3-512. The message digest can be used to verify the integrity of a set of information.

The SHA-2 and SHA-3 algorithms were standardized by the U.S. National Institute of Standards and Technology (NIST) in 2001 and 2015.

For more information on the details of the SHA-2 and SHA-3 algorithms, please refer to Federal Information Processing Standard Publication 180-4 and 202 (FIPS 180-4 and FIPS 202) that can be downloaded from the NIST website at https://www.nist.gov.

ELE supports:

- Secure Hash Algorithm 1 (SHA-1)
- Secure Hash Algorithm 2 224 (SHA2-224)
- Secure Hash Algorithm 2 256 (SHA2-256)
- Secure Hash Algorithm 2 384 (SHA2-384)
- Secure Hash Algorithm 2 512 (SHA2-512)
- Secure Hash Algorithm 3 224 (SHA3-224)
- Secure Hash Algorithm 3 256 (SHA3-256)
- Secure Hash Algorithm 3 384 (SHA3-384)
- Secure Hash Algorithm 3 512 (SHA3-512)

NOTE

All SHA2 algorithms are supported by hardware. SHA1 and SHA3 algorithms are software implementation.

The following table shows the supported commands for HASH functions.

Table 104. HASH commands

Command ID	Commands	Descriptions
0x2C	DIGEST_CONTEXT_INIT	Initialize context for message digest operation
0x2F	DIGEST_ONE_GO	Perform message digest operation in one step
0x2E	DIGEST_INIT	Initialize block (init/update/finish) message digest operation
0x30	DIGEST_UPDATE	Perform block (init/update/finish) message digest operation for new data block/message
0x2D	DIGEST_FINISH	Finish block (init/update/finish) message digest operation and returns digest
0x81	DIGEST_CLONE	Clone actual state of digest context to another digest context.
0x82	DIGEST_IMPORT	Import exported state of digest context from the digest context blob into selected digest context.
0x83	DIGEST_EXPORT	Export actual state of digest context out of ELE into the digest context blob. The blob is encrypted by device die unique key.

7.6 Authenticated encryption with associated data (AEAD) algorithms

The ELE supports following AEAD algorithms:

- AES-CCM
- AES-GCM

The following table shows the supported commands for authenticated encryption with associated data algorithms.

Table 105. Authenticated encryption with associated data algorithms commands

Command ID	Commands	Descriptions
0x26	AEAD_CONTEXT_INIT	Initialize context for AEAD operation
0x29	AEAD_ONE_GO	Perform authenticated symmetric encryption or decryption in one step

7.7 Key derivation function (KDF) algorithms

The following table shows the supported commands for key derivation algorithms.

Table 106.

Command ID	Command	Description
0x40	DERIVE_KEY_CONTEXT_INIT	Initialize key derivation operation.
0x3C	ASYMMETRIC_DH_DERIVE_KEY	Perform Diffie-Hellman key exchange operation.
0x3F	DERIVE_KEY	Perform symmetric derivation function.
0x86	ASYMMETRIC_SPAKE2_DERIVE_KEY	Perform SPAKE2+ key derivation function.

7.8 Other services

The following table shows additional commands and messages for other services.

Table 107. Commands for other services

Command ID	Commands	Descriptions
0x11	PING	Check whether the ELE is alive and ready to receive new command
0x13	OPEN_SESSION	Open session between the host and the ELE
0x14	CLOSE_SESSION	Close opened session
0x60	MGMT_ADVANCE_LIFECYCLE	Change lifecycle state into the next state
0x65	MGMT_CONTEXT_INIT	Initialize context for ELE management commands
0x67	MGMT_FUSE_PROGRAM	Perform ELE fuse programming
0x68	MGMT_FUSE_READ	Perform ELE fuse read
0x6B	MGMT_GET_LIFECYCLE	Get actual device lifecycle
0x6C	MGMT_GET_PROPERTY	Get security device properties
0x71	MGMT_SET_PROPERTY	Set security device properties
0x70	MGMT_SET_HOST_ACCESS_PERMISSION	Force lower security level for all bus transactions performed during next ELE command
0x73	MGMT_GET_RANDOM	Get random number of requested length
0x74	MGMT_CLEAR_ALL_KEYS	Overwrite complete key store with random numbers

7.9 Main boot image (MBI) and secure binary file (SB3) authentication

The following table shows the supported commands for MBI and SB3 authentication.

Table 108. MBI and SB3 authentication commands

Command ID	Commands	Descriptions
0x3D	TUNNEL_CONTEXT_INIT	Initialize context for tunnel operation
0x23	TUNNEL_REQUEST	Perform tunnel operation

7.10 TRNG

A nonce is a random or pseudo-random number that is used only once in a cryptographic calculation. Nonces are important for initialization vectors and cryptographic hash functions.

The ELE natively supports a True Random Number Generator (TRNG) which is a source of entropy that can be used to generate random numbers for use in cryptographic algorithms

The following table shows the supported commands for TRNG.

Table 109. TRNG commands

Command ID	Commands	Descriptions
0x73	MGMT_GET_RANDOM	Get random number of requested length

7.11 Key storage services

The ELE implements a key store, which can be used to store user keys for crypto operations executed in the ELE. All crypto operations use keys stored in ELE key store only. User key from outside can be stored into ELE key store by KEY_STORE_SET_KEY command. Any key in ELE key store can be also exported by KEY_STORE_EXPORT_KEY command in encrypted key blob. This key blob can be stored in NV memory and later imported back into ELE key store. See Key Management for more details.

The following table shows the supported commands for key storage.

Table 110. Key storage commands

Command ID	Commands	Descriptions
0x49	KEY_STORE_ INIT	Initialize the user key store
0x76	KEY_STORE_FREE	Delete all keys stored in the key store and de-initialize the key store
0x4C	KEY_STORE_SET_KEY	Store plain key/key pair into key store if allowed by key properties
0x4E	KEY_STORE_GET_KEY	Read key in plaintext from the key store if allowed by key properties
0x79	KEY_STORE_EXPORT_KEY	Export key/key pair as key blob if allowed by key properties
0x78	KEY_STORE_IMPORT_KEY	Import key/key pair blob into ELE internal key store if allowed by key properties
0x4D	KEY_STORE_GENERATE_KEY	Generate random key/key pair
0x4F	KEY_STORE_OPEN_KEY	Send internal ELE key to another peripheral via private key bus
0x51	KEY_STORE_ERASE_KEY	Erase the unlocked key
0x77	KEY_STORE_GET_PROPERTY	Get configuration of the key store and actual available key store data size and key objects number
0x41	KEY_OBJECT_INIT	Initialize key object and returns key object ID
0x42	KEY_OBJECT_ALLOCATE_HANDLE	Allocate key slot in the key store for the key object
0x43	KEY_OBJECT_GET_HANDLE	Get key object ID
0x45	KEY_OBJECT_GET_PROPERTIES	Get key object properties
0x44	KEY_OBJECT_SET_PROPERTIES	Set key object properties
0x47	KEY_OBJECT_FREE	Delete key object

7.12 ELE TrustZone and multicore/multithread support

The ELE uses several mechanisms to support TrustZone and multicore or multithread applications:

- Host bus attributes masquerading
- Security level protection
- Random identifiers
- Hardware semaphore

7.12.1 Host bus attributes masquerading

Some crypto operation commands require access to the host memory resources. To create security isolation at host level for memory, bus and peripherals TrustZone is enabled. TrustZone memory is a secure memory which cannot be accessed by non-secure access. The ELE has access to both secure and non-secure access and this needs to be handled by the ELE so that secure asset is handled with secure access and non-secure asset by non-secure access. This masquerading process helps to ensure that non-secure host cannot use the ELE to dump secure asset in non-secure memory. Stated differently, the ELE uses the same security level as the host, which initiated the ELE command by a write into ELEMUA_TR0 register.

However, there are situations, when the host needs to perform some crypto operation in memory with a lower security attribute. For example, a secure application wants to calculate the digest over some non-secure memory buffer. The access to the memory with lower security attribute is allowed using MGMT_SET_HOST_ACCESS_PERMISSION command. After this command is executed, the ELE does not use masquerading to inherit security level for upcoming bus transactions, but uses the security level defined in the MGMT_SET_HOST_ACCESS_PERMISSION command. This forced security level is valid for the subsequent command, executed after MGMT_SET_HOST_ACCESS_PERMISSION command only. For any other following commands, the ELE switches back to masquerading mode and inherits the bus transaction security level from the host.

7.12.2 ELE security level protection

If TrustZone environment is configured and enabled, the ELE utilizes a protection mechanism to prevent an application thread with a lower security level to interrupt an unfinished operation. For example, if the ELE session was opened with a secure-privilege level, only the secure-privilege level can close this session. Similarly, if a symmetric context is initialized with a non-secure privilege level, the secure user or secure privilege level can perform the symmetric operation or de-initialize the symmetric context.

The same approach is used for user keys. The key security level and domain ID (DID) is captured during the KEY_OBJECT_INIT command and key user must have the same or higher security level. If DID_MATCH key property is set, also master domain ID must match, when key is used.

For more details, see the Security level in the detailed command descriptions.

7.12.3 Random object identifiers

Every ELE object (session, operation, key store or key object) has its own unique identifier (ID), which is used by the host to address given objects in ELE commands. The object identifier is generated in runtime as a random number. The random identifier helps to better isolate ELE objects among different cores or application threads and thus avoid misuse of ELE objects.

7.12.4 Hardware semaphore

The ELE message supports an embedded hardware semaphore, which can be used to share ELE resources among different cores or application threads.

The following registers are related to hardware semaphore.

- SEMA4_SR
- SEMA4_OWNR
- SEMA4_ACQ
- SEMA4_REL
- SEMA4_FREL

See ELEMU register definition for detailed description of these registers.

7.13 Response messages

The following table shows the supported response messages.

Table 111. Response messages

Response messages	Description
INITIALIZATION_FINISHED	Asynchronously sent by ELE after RESET
ABORT	ELE sends this message asynchronously when unexpected error detected during command execution.
SECURITY_VIOLATION	ELE sends this message asynchronously when security violation detected during command execution.
UNKNOWN_COMMAND	ELE sends this message when the command ID is not recognized or supported

Chapter 8
ELE Software Architecture and API

8.1 Overview

ELE services are available via command messages communicated using a messaging unit ELEMUA. Every ELE service works on a request-response basis. It means that every service includes a command message (see Command message format for details) to write to the ELEMUA_TRn registers to request the ELE service. After the service is completed, the ELE replies every command request with a response message (see Response message format for details), which can be read via ELEMUA_RRn registers. The following figure illustrates this process.

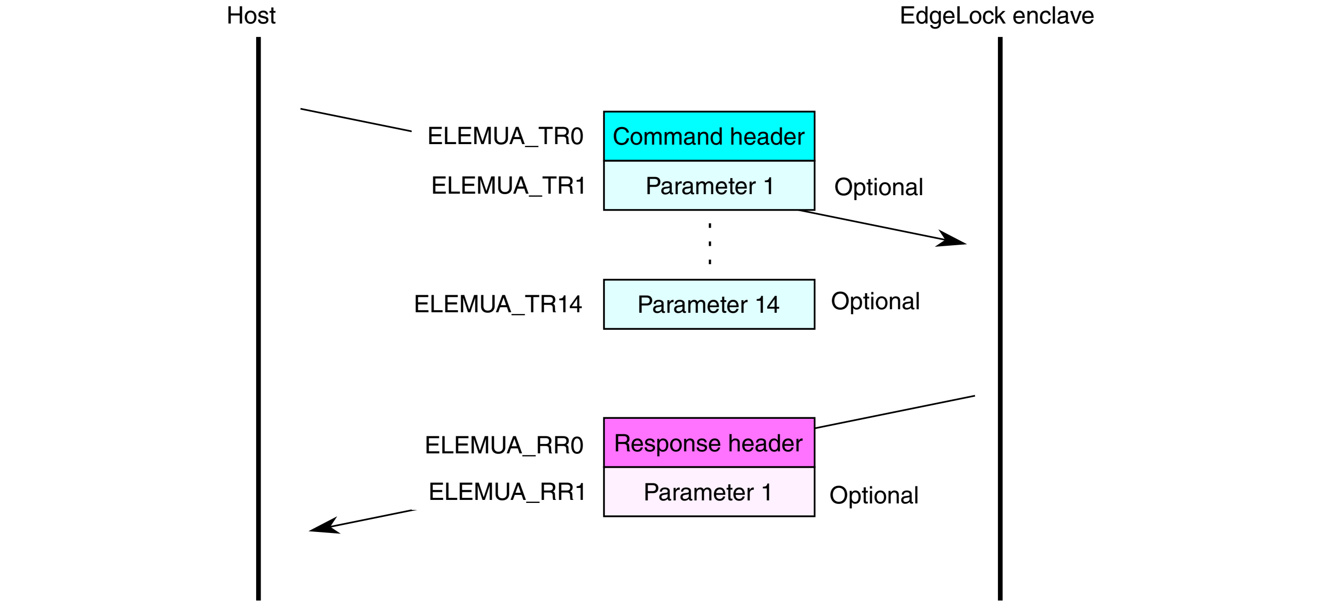
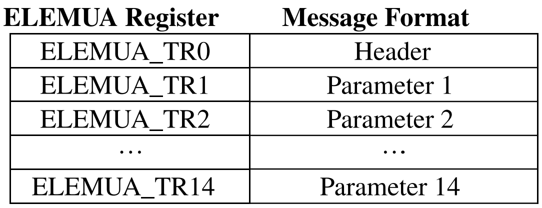


Figure 48. Request-response based ELE service

8.2 Command message format

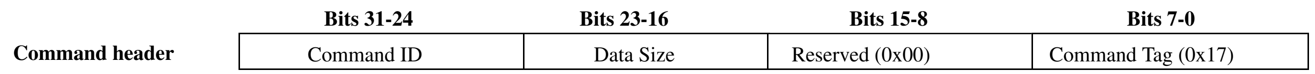
The command message consists of mandatory 32-bit header and up to 14 optional 32-bit parameters as follows.



ELEMUA Register	Message Format
ELEMUA_TR0	Header
ELEMUA_TR1	Parameter 1
ELEMUA_TR2	Parameter 2
…	…
ELEMUA_TR14	Parameter 14

Figure 49. Command message format

The header format for command message can be seen on following figure:



	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
Command header	Command ID	Data Size	Reserved (0x00)	Command Tag (0x17)

Figure 50. Command Header format

The fields in header above are defined as:

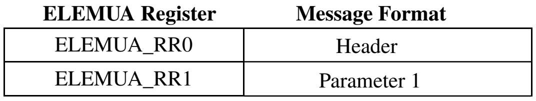
Field Name	Description
Command ID	8-bit command field defining the ELE service to be performed.
Data Size	Number of 32-bit message parameters (0-14)
Command Tag	0x17

NOTE

The command header has to be written as the first word of the command message. After the data size in the command header is decoded, the ELE disables writes to the ELEMUA_TRn registers, which are not used by the command. Any write to a disabled ELEMUA_TRn is terminated by bus error. For example, if command data size is equal to two message parameters, the user is allowed to write to the ELEMUA_TR0, ELEMUA_TR1 and ELEMUA_TR2 registers only. After the command is executed, the ELE is ready to receive a new command again.

8.3 Response message format

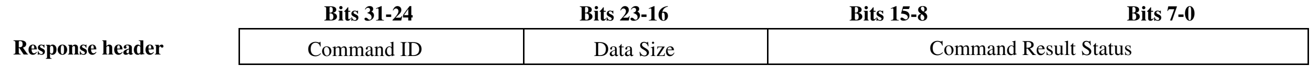
The response message consists of a mandatory 32-bit header and one optional 32-bit parameter as follows:



ELEMUA Register	Message Format
ELEMUA_RR0	Header
ELEMUA_RR1	Parameter 1

Figure 51. Response message format

The header format for the response header message can be seen in the following figure:



	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
Response header	Command ID	Data Size	Command Result Status	

Figure 52. Response Header format

The fields in the header above are defined as:

Field Name	Description
Command ID	8-bit command field defining the ELE service that was performed. The command field is "returned" to the host in the response header. This allows the host software to precisely associate ELE responses with their messages.
Data size	Number of 32-bit message parameters (0-1)
Command result status	Result of command execution (for details see command descriptions)

8.4 Asynchronous response messages

The ELE typically sends response messages on requests only. However, there are some exceptions described in the following sections.

8.4.1 INITIALIZATION_FINISHED

This message is asynchronously sent by ELE after RESET and consists of two 32-bit words. The first 32-bit word is written into ELEMUA_RR0 at the beginning of ELE initialization sequence. The second 32-bit word is written into ELEMUA_RR1 after ELE initialization is completed. After these two words are received by a host, the ELE is ready for operation.

NOTE

During regular secure boot the INITIALIZATION_FINISHED message is processed by boot ROM, so after ROM jumps to user application, the ELE is already ready for operation. However, if the ELE is reset by the user application, the ELE sends this message again during initialization sequence and user needs to process it appropriately.

Table 112. INITIALIZATION_FINISHED message format

Parameter	ELEMUA_RRn	Description
Message header	0	0x1700D03C, initialization sequence has been started
Parameter 1	1	0x1B00803C, initialization has been finished

8.4.2 ABORT

The ELE sends this message asynchronously in case of any unexpected error detected during command execution. After this message is received, the ELE must be reset in order to continue operation.

Table 113. ABORT response message format

Parameter	ELEMUA_RRn	Description
Message header	0	0x51008309, ABORT message header
Parameter 1	1	Error code

Table 114. Error code

Error code	Description
0xc3, 0xff, 0xb5	General error
0x88, 0xb2	Security violation
0xEAEA0000, 0xEAEA0001, 0xEAEA0004, 0xEAEA0005, 0xEAEA0006, 0xEAEA0007, 0xEAEA0009	Initialization error
0xEAEA0002	Invalid lifecycle
0xEAEA0010	Non-secure privilege access to IFR1

8.4.3 SECURITY_VIOLATION

The ELE sends this message asynchronously in case of any security violation detected. After this message is received, the ELE must be reset in order to continue operation.

Table 115. SECURITY_VIOLATION response message format

Parameter	ELEMUA_RRn	Description
Message header	0	0x1D014CC3, SECURITY_VIOLATION message header
Parameter 1	1	0x00009211

8.5 ELE commands

The ELE provides the following commands.

8.5.1 PING

This command is used to check whether the ELE is alive and ready to receive a new command. It can be executed any time the ELE is ready to receive new commands. If the ELE is ready for the next command, it returns the response-command succeeded.

Table 116. Ping command format

Command ID	0x11	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	0/0
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x11000017	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x1100xxxx, where xxxx: 0x3C3C: Command success NOTE It does not receive response message when the command fails.	

8.5.2 OPEN_SESSION

This command opens a session between the host and the ELE. This command must be called before any other command is requested except the commands listed below:

- PING
- MGMT_GET_RANDOM
- MGMT_SET_HOST_ACCESS_PERMISSION

Only one session can be opened at a time.

Table 117. OPEN_SESSION command format

Command ID	0x13	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	2/1

Table continues on the next page...

Table 117. OPEN_SESSION command format (continued)

Security level	Any	
Command message format		
Parameter	**ELEMUA_TRn number**	**Description**
Message header	0	0x13020017
Subsystem type	1	0x00000002: ELE
User session ID	2	User session ID: Any 32-bit user value to identify session NOTE • When user session ID=0, this command can be called just once. Any repeated call returns error. • When user session ID is non-zero, this command can be called repeatedly. When this command is called repeatedly with the same user session ID, it returns the session ID of opened session. This feature allows ELE sharing among different masters.
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x1301xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Session ID	1	Session ID. Session ID is used as parameter for other commands.

8.5.3 CLOSE_SESSION

This command closes an opened session.

Table 118. CLOSE_SESSION command format

Command ID	0x14	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/0
Security level	The same or higher security level used with the OPEN_SESSION command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x14010017	
Session ID	1	Session ID obtained by OPEN_SESSION command	

Table continues on the next page...

Table 118. CLOSE_SESSION command format (continued)

Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x1400xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

8.5.4 CONTEXT_FREE

This command closes an opened operation context. This command can be used to close any operation type context.

Table 119. CONTEXT_FREE command format

Command ID	0x15	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/0
Security level	The same or higher security level used with the xxx_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x15010017	
Operation context ID	1	Operation context ID	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x1500xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	

8.5.5 SYMMETRIC_CONTEXT_INIT

This command initializes context for a symmetric operation. Only one SYMMETRIC context can be opened at a time.

Table 120. SYMMETRIC_CONTEXT_INIT command format

Command ID	0x25	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	4/1
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x25040017	
Session ID	1	Session ID	
Key object ID	2	Key object ID used for symmetric crypto operation	

Table continues on the next page...

Table 120. SYMMETRIC_CONTEXT_INIT command format (continued)

Block cipher mode	3	Block cipher mode: 0x00000000: AES ECB (Electronic Code Book) 0x00000001: AES CBC (Cipher Block Chaining) 0x00000002: AES CTR (Counter Mode)
Cipher operation mode	4	Cipher operation mode: 0x00000000: Encryption mode 0x00000001: Decryption mode
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x2501xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Operation context ID	1	Symmetric operation context ID

8.5.6 CIPHER_ONE_GO

This command performs symmetric encryption or decryption in one step.

Table 121. Table 8. CIPHER_ONE_GO command format

Command ID	0x23	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	6/0
Security level	The same or higher security level used with SYMMETRIC_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x23060017	
Operation context ID	1	Operation context ID obtained by SYMMETRIC_CONTEXT_INIT command	
IV	2	Start address of initial vector (IV) source buffer	
IV length	3	IV length in bytes (must always be 16 bytes)	
Source data	4	Start address of source data buffer	
Destination data	5	Start address of destination data buffer	
Data length	6	Data length in bytes to be processed	

Table continues on the next page...

Table 121. Table 8. CIPHER_ONE_GO command format (continued)

		NOTE When encrypting data, the input length have to be block-size aligned (multiple of 16 bytes) except CTR block cipher mode, which doesn't require data alignment.
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x2300xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

8.5.7 AEAD_CONTEXT_INIT

This command initializes context for AEAD operation. Only one AEAD context can be opened at a time.

Table 122. AEAD_CONTEXT_INIT command format

Command ID	0x26	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	4/1
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x26040017	
Session ID	1	Session ID	
Key object ID	2	Key object ID used for AEAD crypto operation	
Block cipher mode	3	Block cipher mode: 0x00000003: For AES GCM (Galois/Counter Mode) 0x00000004: For AES CCM (counter with cipher block chaining message authentication code)	
Cipher operation mode	4	Cipher operation mode: 0x00000000: Authenticated encryption mode 0x00000001: Authenticated decryption mode	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x2601xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	
Operation context ID	1	AEAD operation context ID	

8.5.8 AEAD_ONE_GO

This command performs an authenticated symmetric encryption or decryption in one step. This operation is non-interruptible.

Table 123. AEAD_ONE_GO command format

Command ID	0x29	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	10/1
Security level	The same or higher security level used with the AEAD_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x290A0017	
Operation context ID	1	Operation context ID obtained by AEAD_CONTEXT_INIT command	
Source data	2	Start address of source data buffer	
Destination data	3	Start address of destination data buffer	
Data length	4	Data length in bytes to be processed NOTE When encrypting data, the input length does not have to be block-size aligned (multiple of 16 bytes). ELE will pad the input data to the next block size internally. The output data will always be block-size aligned though. To avoid buffer overflows, the destination data buffer must be large enough to hold the block-size aligned output.	
IV	5	Start address of initialization vector buffer	
IV length	6	Initialization vector data length in bytes	
Authentication data	7	Start address of authentication data buffer	
Authentication data length	8	Authentication data length in bytes	
TAG	9	Start address of TAG buffer	
TAG length	10	Tag length in bytes	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x2901xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	
TAG length	1	0x00000000: for decryption mode Others: Number of bytes written into TAG buffer for encryption mode	

8.5.9 DIGEST_CONTEXT_INIT

This command initializes context for message digest operation. Four digest contexts can be opened at the same time.

Table 124. DIGEST_CONTEXT_INIT command Format

Command ID	0x2C	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	3/1
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x2C030017	
Session ID	1	Session ID	
Digest algorithm	2	Hash algorithm: 0x0000008: SHA-1 0x00000009: SHA2-224 0x0000000A: SHA2-256 0x0000000B: SHA2-384 0x0000000C: SHA2-512 0x00000070: SHA3-224 0x00000071: SHA3-256 0x00000072: SHA3-384 0x00000073: SHA3-512	
Operation mode	3	0x05: Digest mode	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x2C01xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	
Operation context ID	1	Hash operation context ID	

8.5.10 DIGEST_ONE_GO

This command performs message digest operation in one step. This operation is non-interruptible.

Table 125. DIGEST_ONE_GO command format

Command ID	0x2F	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	5/1
Security level	The same or higher security level used with the DIGEST_CONTEXT_INIT command		

Table continues on the next page...

Table 125. DIGEST_ONE_GO command format (continued)

Command message format		
Parameter	**ELEMUA_TRn number**	**Description**
Message header	0	0x2F050017
Operation context ID	1	Operation context ID obtained by DIGEST_CONTEXT_INIT command
Message	2	Start address of message buffer
Message buffer length	3	Message buffer length in bytes
Digest	4	Start address of digest buffer
Digest length	5	Digest buffer length in bytes
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x2F01xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Digest length	1	Number of bytes written into digest buffer

8.5.11 DIGEST_INIT

This command initializes a block (init/update/finish) message digest operation. This operation can be interrupted by any other digest operation.

Table 126. DIGEST_INIT command format

Command ID	0x2E	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/0
Security level	The same or higher security level used with the DIGEST_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x2E010017	
Operation context ID	1	Operation context ID obtained by DIGEST_CONTEXT_INIT command	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x2E00xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	

8.5.12 DIGEST_UPDATE

This command performs a block (init/update/finish) message digest operation for new data block/message. This operation can be interrupted by any other digest operation.

Table 127. DIGEST_UPDATE command format

Command ID	0x30	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	3/0
Security level	The same or higher security level used with the DIGEST_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x30030017	
Operation context ID	1	Operation context ID obtained by DIGEST_CONTEXT_INIT command	
Message	2	Start address of message buffer	
Message buffer length	3	Message buffer length in bytes	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x3000xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	

8.5.13 DIGEST_FINISH

This command finishes a block (init/update/finish) message digest operation and returns digest. This operation can be interrupted by any other digest operation.

Table 128. DIGEST_FINISH command format

Command ID	0x2D	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	3/1
Security level	The same or higher security level used with the DIGEST_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x2D030017	
Operation context ID	1	Operation context ID obtained by DIGEST_CONTEXT_INIT command	
Digest	2	Start address of digest buffer	
Digest length	3	Digest buffer length in bytes	
Response message format			

Table continues on the next page...

Table 128. DIGEST_FINISH command format (continued)

Parameter	ELEMUA_RRn number	Description
Message header	0	0x2D01xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Digest length	1	Number of bytes written into digest buffer

8.5.14 DIGEST_CLONE

This command clone actual state of the digest context into another digest context/object. The command can be called any time between DIGEST_CONTEXT_INIT and CONTEXT_FREE commands.

Table 129. DIGEST_CLONE command format

Command ID	0x81	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	2/0
Security level	The same or higher security level used at DIGEST_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x81020017	
Source context ID	1	Source context ID obtained by DIGEST_CONTEXT_INIT command	
Destination context ID	2	Destination context ID obtained by DIGEST_CONTEXT_INIT command	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x8100xxxx, where xxxx: 0x3c3c: in case of command success 0xc3c3: in case of command fail	

8.5.15 DIGEST_IMPORT

This command imports digest context blob into selected digest context/object. The command can be called any time between DIGEST_CONTEXT_INIT and CONTEXT_FREE commands.

NOTE

The digest context blob is encrypted by device die unique key. Therefore the blob exported on the same device can be imported only.

Table 130. DIGEST_IMPORT command format

Command ID	0x82	**Command Data Size [ELEMUA_TRn/ ELEMUA_RRn]**	3/0

Table continues on the next page...

Table 130. DIGEST_IMPORT command format (continued)

Security level	The same or higher security level used at DIGEST_CONTEXT_INIT command	
Command message format		
Parameter	**ELEMUA_TRn number**	**Description**
Message header	0	0x82030017
Destination context ID	1	Destination context ID obtained by DIGEST_CONTEXT_INIT command
Digest context blob	2	Start address of digest context blob
Digest context blob length	3	Digest context blob data length in bytes (412 bytes)
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x8200xxxx, where xxxx: 0x3c3c: in case of command success 0xc3c3: in case of command fail

8.5.16 DIGEST_EXPORT

This command exports actual state of selected digest context/object into digest context blob. The command can be called any time between DIGEST_CONTEXT_INIT and CONTEXT_FREE commands.

NOTE

The digest context blob is encrypted by device die unique key. Therefore this blob cannot be imported on different devices.

Table 131. DIGEST_EXPORT command format

Command ID	0x83	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	3/1
Security level	The same or higher security level used at DIGEST_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x83030017	
Source context ID	1	Source context ID obtained by DIGEST_CONTEXT_INIT command	
Digest context blob	2	Start address of digest context blob	
Digest context blob length	3	Digest context blob data length in bytes (412 bytes)	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x8301xxxx, where xxxx:	

Table continues on the next page...

Table 131. DIGEST_EXPORT command format (continued)

		0x3c3c: in case of command success 0xc3c3: in case of command fail
Blob length	1	Number of bytes written into digest context blob buffer

8.5.17 MAC_CONTEX_INIT

This command initializes context for a message authentication code (MAC) operation. Four MAC contexts can be opened at the same time.

Table 132. MAC_CONTEX_INIT command format

Command ID	0x32	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	4/1
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x32040017	
Session ID	1	Session ID	
Key object ID	2	Key object ID used for MAC operation	
MAC algorithm	3	0x0000000D: CMAC with AES 0x00000060: HMAC with SHA1 0x00000061: HMAC with SHA2-224 0x0000000E: HMAC with SHA2-256 0x00000063: HMAC with SHA2-384 0x00000064: HMAC with SHA2-512	
Operation mode	4	0x00000006: MAC mode	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x3201xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	
Operation context ID	1	MAC operation context ID	

8.5.18 MAC_ONE_GO

This command performs a message authentication code (MAC) operation in one step. This operation is non-interruptible.

Table 133. MAC_ONE_GO command format

Command ID	0x35	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	5/1
Security level	The same or higher security level used with the MAC_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x35050017	
Operation context ID	1	Operation context ID obtained by MAC_CONTEX_INIT command	
Message	2	Start address of message buffer	
Message buffer length	3	Message buffer length in bytes	
MAC	4	Start address of MAC buffer	
MAC length	5	MAC buffer length in bytes	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x3501xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	
MAC length	1	Number of bytes written into MAC buffer	

8.5.19 MAC_INIT

This command initializes block (init/update/finish) message authentication code (MAC) operation. This operation can be interrupted by any other MAC operation.

Table 134. MAC_INIT command format

Command ID	0x34	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/0
Security level	The same or higher security level used at MAC_CONTEXT_INIT command		
Command Message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x34010017	
Operation context ID	1	Operation context ID obtained by MAC_CONTEXT_INIT command	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x3400xxxx, where xxxx:	

Table continues on the next page...

Table 134. MAC_INIT command format (continued)

		0x3c3c: in case of command success 0xc3c3: in case of command fail

8.5.20 MAC_UPDATE

This command performs block (init/update/finish) message authentication code (MAC) operation for new data block/message. This operation can be interrupted by any other MAC operation.

Table 135. MAC_UPDATE command format

Command ID	0x36	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	3/0
Security level	The same or higher security level used at MAC_CONTEXT_INIT command		
Command Message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x36030017	
Operation context ID	1	Operation context ID obtained by MAC_CONTEXT_INIT command	
Message	2	Start address of message buffer	
Message buffer length	3	Message buffer length	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x3600xxxx, where xxxx: 0x3c3c: in case of command success 0xc3c3: in case of command fail	

8.5.21 MAC_FINISH

This command finishes block (init/update/finish) message authentication code (MAC) operation and returns MAC tag. This operation can be interrupted by any other MAC operation.

Table 136. MAC_FINISH command format

Command ID	0x33	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	3/1
Security level	The same or higher security level used at MAC_CONTEXT_INIT command		
Command Message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x33030017	

Table continues on the next page...

Table 136. MAC_FINISH command format (continued)

Operation context ID	1	Operation context ID obtained by MAC_CONTEXT_INIT command
MAC tag	2	Start address of MAC tag buffer
MAC tag length	3	MAC tag buffer length
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x3301xxxx, where xxxx: 0x3c3c: in case of command success 0xc3c3: in case of command fail
MAC length	1	Number of bytes written into MAC buffer

8.5.22 MAC_IMPORT

This command imports MAC context blob into selected MAC context/object. The command can be called any time between MAC_CONTEXT_INIT and CONTEXT_FREE commands.

> NOTE
>
> The MAC context blob is encrypted by device die unique key. Therefore the blob exported on the same device can be imported only.

Table 137. MAC_IMPORT command format

Command ID	0x84	**Command Data Size [ELEMUA_TRn/ ELEMUA_RRn]**	3/0
Security level	The same or higher security level used at MAC_CONTEXT_INIT command		
Command Message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x84030017	
Destination context ID	1	Operation context ID obtained by MAC_CONTEXT_INIT command	
MAC context blob	2	Start address of MAC context blob	
MAC context blob length	3	MAC context blob data length in bytes (384 bytes)	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x8400xxxx, where xxxx: 0x3c3c: in case of command success 0xc3c3: in case of command fail	

8.5.23 MAC_EXPORT

This command exports actual state of selected MAC context/object into MAC context blob. The command can be called any time between MAC_CONTEXT_INIT and CONTEXT_FREE commands.

> NOTE
>
> The MAC context blob is encrypted by device die unique key. Therefore this blob cannot be imported on different devices.

Table 138. MAC_EXPORT command format

Command ID	0x85	**Command Data Size [ELEMUA_TRn/ ELEMUA_RRn]**	3/1
Security level	The same or higher security level used at MAC_CONTEXT_INIT command		
Command Message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x85030017	
Source context ID	1	Operation context ID obtained by MAC_CONTEXT_INIT command	
MAC context Blob	2	Start address of MAC context blob	
MAC context Blob Length	3	MAC context blob data length in bytes (384 bytes)	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x8301xxxx, where xxxx: 0x3c3c: in case of command success 0xc3c3: in case of command fail	
Blob length	1	Number of bytes written into MAC context blob buffer	

8.5.24 ASYMMETRIC_CONTEXT_INIT

This command initializes context for an asymmetric (sign/verify) operation. Only one ASYMMETRIC context can be opened at a time.

Table 139. ASYMMETRIC_CONTEXT_INIT command format

Command ID	0x37	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	4/1
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x37040017	
Session ID	1	Session ID	

Table continues on the next page...

Table 139. ASYMMETRIC_CONTEXT_INIT command format (continued)

Key object ID	2	Key object ID used for sign/verify digest operation
Algorithm	3	Algorithm used for sign/verify digest operation: 0x0000001F - 0x00000022: ECDSA (EC type selected from "Key Cipher Type" option of used key) • 0x0000001F: ECDSA_SHA224 • 0x00000020: ECDSA_SHA256 • 0x00000021: ECDSA_SHA384 • 0x00000022: ECDSA_SHA512 0x00000031: Ed25519 (EdDSA with Curve25519)
Operation mode	4	Operation mode: 0x00000002: Signature generation (sign mode) 0x00000003: Signature verification (verify mode)
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x3701xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Operation context ID	1	Sign/verify digest operation context ID

8.5.25 ASYMMETRIC_SIGN

This command performs an asymmetric sign operation.

Table 140. ASYMMETRIC_SIGN command format

Command ID	0x3A	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	5/1
Security level	The same or higher security level used with the ASYMMETRIC_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x3A050017	
Operation context ID	1	Operation context ID obtained by ASYMMETRIC_CONTEXT_INIT command	
Digest	2	Start address of digest buffer	
Digest buffer length	3	Digest buffer length in bytes	
Signature	4	Start address of signature buffer	
Signature buffer length	5	Signature buffer length in bytes	

Table continues on the next page...

Table 140. ASYMMETRIC_SIGN command format (continued)

Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x3A01xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Signature length	1	Number of bytes written into signature buffer

8.5.26 ASYMMETRIC_VERIFY

This command performs an asymmetric verify operation.

Table 141. ASYMMETRIC_VERIFY command format

Command ID	0x3B	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	5/0
Security level	The same or higher security level used with the ASSYMETRIC_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x3B050017	
Operation context ID	1	Operation context ID obtained by ASYMMETRIC_CONTEXT_INIT command	
Digest	2	Start address of digest buffer	
Digest buffer length	3	Digest buffer length in bytes	
Signature	4	Start address of signature buffer	
Signature buffer length	5	Signature buffer length in bytes	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x3B00xxxx, where xxxx: 0x3C3C: verification success 0xC3C3: verification or Command fail	

8.5.27 DERIVE_KEY_CONTEXT_INIT

This command initializes context for a Diffie-Hellman key exchange operation. Only one DERIVE_KEY context can be open at a time.

Table 142. DERIVE_KEY_CONTEXT_INIT command format

Command ID	0x40	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	4/1

Table continues on the next page...

Table 142. DERIVE_KEY_CONTEXT_INIT command format (continued)

Security level	Any	
Command message format		
Parameter	**ELEMUA_TRn number**	**Description**
Message header	0	0x40040017
Session ID	1	Session ID
Key object ID	2	Key object ID used for key derivation
Algorithm	3	**Symmetric key derivation functions:** 0x00000000: AES ECB key derivation function, derived key size is fixed to 128 bits. Input derivationData size is fixed to 16 bytes. Used KDK must be symmetric key with size of 128 bits. Derivation function: derivedKey = AES_ECB_encrypt(KDK, derivationData). 0x00000050: Bluetooth LE 5.x secure Connections key generation function f5 0x00000051: CMAC key derivation function (CKDF) 0x00000052: HMAC SHA1 key derivation function (HKDF), step1 – extract 0x00000053: HMAC SHA2-224 key derivation function (HKDF), step1 – extract 0x00000054: HMAC SHA2-256 key derivation function (HKDF), step1 – extract 0x00000055: HMAC SHA2-384 key derivation function (HKDF), step1 – extract 0x00000056: HMAC SHA2-512 key derivation function (HKDF), step1 – extract 0x00000057: HMAC SHA1 key derivation function (HKDF), step1 – extract 0x00000058: HMAC SHA2-224 key derivation function (HKDF), step2 – expand 0x00000059: HMAC SHA2-256 key derivation function (HKDF), step2 – expand 0x0000005A: HMAC SHA2-384 key derivation function (HKDF), step2 – expand 0x0000005B: HMAC SHA2-512 key derivation function (HKDF), step2 – expand **Asymmetrical key derivation function:** 0x00000010: ECDH (Diffie-Hellman key exchange using NIST curves (P-224, P-256, P-384 and P-521). The NIST curve is selected based on key bit length. 0x00000030: X25519 (Diffie-Hellman key exchange using Curve25519)

Table continues on the next page...

Table 142. DERIVE_KEY_CONTEXT_INIT command format (continued)

		0x00000040: ELE to ELE blob key derivation 0x00000051: CMAC key derivation function, derived
Operation mode	4	0x00000004: Asymmetric key derivation function 0x00000007: Symmetric key derivation function
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x4001xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Operation context ID	1	Operation context ID

8.5.28 ASYMMETRIC_DH_DERIVE_KEY

This command performs a Diffie-Hellman key exchange operation.

Table 143. ASYMMETRIC_DH_DERIVE_KEY command format

Command ID	0x3C	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	3/1
Security level	The same or higher security level used with DERIVE_KEY_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x3C030017	
Operation context ID	1	Operation context ID obtained by DERIVE_KEY_CONTEXT_INIT command	
Other party key object ID	2	Other party key object ID	
Derived key object ID	3	Derived key object ID **NOTE** This parameter is ignored for ELE to ELE blob key derivation, because resultant key is not stored into user key store. In this case 0 should be written as derived key object ID.	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x3C01xxxx, where xxxx: 0x3C3C: Command success	

Table continues on the next page...

Table 143. ASYMMETRIC_DH_DERIVE_KEY command format (continued)

		0xC3C3: Command fail
Reserved	1	Reserved. Always returns 0

8.5.29 DERIVE_KEY

This command performs symmetric key derivation function operation. This command has two format based on number of needed parameters. All key derivation algorithms use format1 except CKDF and HKDF based algorithms, which use format 2.

Table 144. DERIVE_KEY command format 1

Command ID	0x3F	**Command data size (ELEMUA_TRn/ ELEMUA_RRn)**	4/1
Security level	The same or higher security level used at DERIVE_KEY command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x3F040017	
Operation context ID	1	Operation context ID obtained by DERIVE_KEY_CONTEXT_INIT command	
Salt data	2	Salt data	
Salt data length	3	Salt data length in bytes	
Derived key object ID	4	Derived key object ID	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x3F01xxxx, where xxxx: 0x3c3c: in case of command success 0xc3c3: in case of command fail	
Reserved	1	Reserved Always returns 0	

Table 145. DERIVE_KEY command format 2

Command ID	0x3F	**Command data size (TRx/RRx)**	5/1
Security level	The same or higher security level used at DERIVE_KEY command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x3F050017	
Operation context ID	1	Operation context ID obtained by DERIVE_KEY_CONTEXT_INIT command	

Table continues on the next page...

Table 145. DERIVE_KEY command format 2 (continued)

Salt data	2	Salt data
Salt data length	3	Salt data length in bytes
Derived key object ID	4	Derived key object ID
Derived key bit length	5	Derived key bit length
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x3F01xxxx, where xxxx: 0x3c3c: in case of command success 0xc3c3: in case of command fail
Reserved	1	Reserved Always returns 0

8.5.29.1 AES ECB key derivation function

This function implements following key derivation function:

K_D = AES ECB (K_{IN}, salt data), where

K_{IN} – key object ID used for key derivation set in DERIVE_KEY_INIT command

K_D – destination derived key object ID

This function is limited to 128-bit key length so K_{IN}, K_D and salt data must have fixed size 128-bit.

8.5.29.2 CMAC key derivation function (CKDF)

This function implements following key derivation function:

$K_D(i)$ = CMAC (K_{IN}, salt data || counter)

$K_D = K_D(1) \,||\, K_D(2) \ldots \,||\, K_D(n)$

where

i = 1 .. n

K_{IN} – key object ID used for key derivation set in DERIVE_KEY_INIT command

K_D – destination derived key object ID

counter – four byte counter value in big endian format

The counter and n are calculated internally based on required derived key length.

8.5.29.3 HMAC key derivation function (HKDF)

This function implements following HMAC key derivation function. This key derivation consists of two step. When the first step extraction is selected the command calculates following formula:

K_{PRK} = HMAC-hash(salt, IKM)

where

K_{PRK} – pseudo random key

$K_D(i)$ = CMAC(K_{IN}, salt data || counter)

$K_D = K_D(1) || K_D(2) … || K_D(n)$

where

i = 1 .. n

K_{IN} – key object ID used for key derivation set in DERIVE_KEY_INIT command

K_D – destination derived key object ID

counter – four byte counter value in big endian format

The counter and n are calculated internally based on required derived key length.

8.5.29.4 Bluetooth 5.x LE secure connections key generation function

The Bluetooth LE Secure Connections key generation function f5 is defined as

f5(W, N1, N2, A1, A2) = AES-CMACT (Counter = 0 || keyID ||N1 || N2|| A1|| A2|| Length = 256)||AES-CMAC$_T$ (Counter = 1 || keyID || N1 || N2|| A1|| A2 || Length = 256)

The DERIVE_KEY command implements this function as two step operation. In the first step the MacKey is calculated as

MacKey = AES-CMAC$_T$ (Counter = 0 || keyID ||N1 || N2|| A1|| A2|| Length = 256)

To calculate MacKey user needs to provide key object containing DHKey as derivation key during DERIVE_KEY_CONTEXT_INIT command (Key object ID parameter) and salt data as

saltDataMacKey[53] = Counter = 0 || keyID ||N1 || N2|| A1|| A2|| Length = 256

After DERIVE_KEY command execution, the derived key MacKey is stored in key object defined by Derived key object ID parameter.

Similarly LTK key is calculated in second step. The LTK key is calculated as

LTK = AES-CMAC$_T$ (Counter = 1 || keyID || N1 || N2|| A1|| A2 || Length = 256)

To calculate LTK key user needs to provide key object containing DHKey key as derivation key during DERIVE_KEY_CONTEXT_INIT command (Key object ID parameter) and salt data as

saltDataLTK[53] = Counter = 1 || keyID ||N1 || N2|| A1|| A2|| Length = 256

After DERIVE_KEY command execution, the derived key LTK is stored in key object defined by Derived key object ID parameter.

The MacKey and LTK key calculation is fully independent and can be executed in any order. For more information about DHKey and salt data calculation see Bluetooth 5.x Specification.

8.5.30 TUNNEL_CONTEXT_INIT

This command initializes context for a tunnel operation. Only one TUNNEL context can be opened at a time. The tunnel service performs the following actions:

- Main boot file/image authentication
- SB3 file processing (authentication and decryption)
- ELE firmware upload (authenticates, decrypts and runs ELE firmware)

Table 146. TUNNEL_CONTEXT_INIT command format

Command ID	0x3D	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	2/1
Security level	Any		
Command message format			

Table continues on the next page...

Table 146. TUNNEL_CONTEXT_INIT command format (continued)

Parameter	ELEMUA_TRn number	Description
Message header	0	0x3D020017
Session ID	1	Session ID
Tunnel type	2	0x20: Master boot image authentication 0x21: SB3 file authentication 0x22: Upload firmware 0x25: SB3 file verification 0x80000021: SB3 file authentication with tester share input 0x80000023: Cmac calculate 0x80000024: Cmac authenticate
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x3D01xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Operation context ID	1	Tunnel operation context ID

8.5.31 TUNNEL_REQUEST

This command performs a tunnel operation. It has two formats based on number of needed parameters.

Table 147. TUNNEL_REQUEST command format 1[1]

Command ID	0x3E	Command data size [ELEMUA_TRn/ ELEMUA_RRn]	3/1
Security level	The same or higher security level used at the TUNNEL_CONTEXT_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x3E030017	
Operation context ID	1	Operation context ID obtained by TUNNEL_CONTEXT_INIT command	
Tunnel data	2	Start address of tunnel data buffer	
Tunnel data length	3	Tunnel data buffer length	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x3E01xxxx, where xxxx: 0x3C3C: in case of command success	

Table continues on the next page...

Table 147. TUNNEL_REQUEST command format 1[1] (continued)

		0xC3C3: in case of command fail
Secure counter result	1	0x0A0B0C0D: secure counter OK Any other values: secure counter fail

[1] Format 1 is used for tunnel types 0x20, 0x21, and 0x22.

Table 148. TUNNEL_REQUEST command format 2[1]

<table>
<tr><td><b>Command ID</b></td><td>0x3E</td><td><b>Command data size [ELEMUA_TRn/ ELEMUA_RRn]</b></td><td>5/1</td></tr>
<tr><td><b>Security level</b></td><td colspan="3">The same or higher security level used at the TUNNEL_CONTEXT_INIT command</td></tr>
<tr><td colspan="4"><b>Command message format</b></td></tr>
<tr><td><b>Parameter</b></td><td><b>ELEMUA_TRn number</b></td><td colspan="2"><b>Description</b></td></tr>
<tr><td>Message header</td><td>0</td><td colspan="2">0x3E050017</td></tr>
<tr><td>Operation context ID</td><td>1</td><td colspan="2">Operation context ID obtained by TUNNEL_CONTEXT_INIT command</td></tr>
<tr><td>Tunnel data</td><td>2</td><td colspan="2">Start address of tunnel data buffer</td></tr>
<tr><td>Tunnel data length</td><td>3</td><td colspan="2">Tunnel data buffer length</td></tr>
<tr><td>Data buffer</td><td>4</td><td colspan="2">Pointer to additional data buffer</td></tr>
<tr><td>Data buffer length</td><td>5</td><td colspan="2">Buffer length in bytes</td></tr>
<tr><td colspan="4"><b>Response message format</b></td></tr>
<tr><td><b>Parameter</b></td><td><b>ELEMUA_RRn number</b></td><td colspan="2"><b>Description</b></td></tr>
<tr><td>Message header</td><td>0</td><td colspan="2">0x3E01xxxx, where xxxx:
0x3C3C: in case of command success
0xC3C3: in case of command fail</td></tr>
<tr><td>Secure counter result</td><td>1</td><td colspan="2">0x0A0B0C0D: secure counter OK
Any other values: secure counter fail</td></tr>
</table>

[1] Format 2 is used for tunnel types 0x80000021, 0x80000023, and 0x80000024.

8.5.31.1 Master boot image authentication (tunnel type = 0x21)

For tunnel type 0x21 the tunnel data (ELEMUA_TR2) contains the start address of master boot image data and tunnel data length (ELEMUA_TR3) total image length in bytes. The command returns success in case of successful authentication or fail in case of failed authentication or another error. The master boot image authentication command is a one go command.

8.5.31.2 SB3 file processing (tunnel type = 0x21)

The SB3 file is processed block by block. The parameter tunnel data (ELEMUA_TR2) contains start address of actual SB3 block and Tunnel data length (ELEMUA_TR3) actual SB3 block length in bytes. An SB3 file consists of two different block types. The first block is the SB3 block manifest (block 0) followed by specified number of data blocks (block 1 ... block n). The number of data blocks can be obtained from the SB3 block manifest. The length of both blocks is variable and depends on number of used root keys and keys size (SB3 manifest block) or data size and keys size (SB3 data block). The SB3 processing command automatically distinguishes between SB3 manifest block and SB3 data blocks (first block is considered as SB3 manifest block). The user is responsible to provide SB3 blocks in the right order and with correct length.

There are two types of SB3 manifest blocks: SB3 OEM file (image type = 0x6; signed by OEM key) and ELE firmware (image type = 0x07; signed by NXP key). The OEM SB3 file must be stored in RAM memory. The decrypted data are written back to the source SB3 data block location. The ELE firmware is decrypted and uploaded into internal ELE RAM. After all data blocks are successfully decrypted, ELE starts to execute the loaded firmware.

The command returns success in case of successful authentication and decryption, or fail in case of failed authentication or another error.

8.5.31.3 ELE firmware upload (tunnel type = 0x22)

This command is one go alternative to SB3 file processing tunnel command. The parameter tunnel data (ELEMUA_TR2) contains start address of ELE firmware and Tunnel data length (ELEMUA_TR3) total length of the firmware in bytes. The whole SB3 file processing block by block is done internally by ELE in one step. This command can be used if the whole firmware is available before the upload is started.

The command returns success if ELE firmware was successfully loaded and executed, or fail in case of failed authentication, decryption, or another error.

8.5.31.4 SB3 file verify (tunnel type = 0x25)

This command verifies integrity of SB3 file without data decryption. The parameter tunnel data contains start address of ELE firmware and tunnel data length contains total length of the firmware. The whole SB3 file verification is done by ELE in one step.

The command returns success if ELE firmware was successfully verified or fail in case verification or another error.

8.5.32 ASYMMETRIC_SPAKE2_DERIVE_KEY

This command performs SPAKE2+ algorithm. The private key pair of responder is stored in key object provided during DERIVE_KEY_CONTEXT_INIT command.

Table 149. ASYMMETRIC_SPAKE2_DERIVE_KEY command format

Command ID	0x86	**Command Data Size [TRx/RRx]**	10/0
Security Level	The same or higher security level used at ASYMMETRIC_DH_DERIVE_KEY command		
Command Message format			
Parameter	**TRx number**	**Description**	
Message Header	0	0x860A0017	
Operation Context ID	1	Operation context ID obtained by DERIVE_KEY_CONTEXT_INIT command	
pA key object ID	2	pA key object ID Public share pA received from initiator provided as data in key object	
w0 key object ID	3	w0 key object ID verification value w0 provided as data in key object	
L key object ID	4	L key object ID verification value L provided as data in key object	

Table continues on the next page...

Table 149. ASYMMETRIC_SPAKE2_DERIVE_KEY command format (continued)

Context Data	5	Context data Context is an application-specific customization string shared between both parties
Context Data Length	6	Context data length in bytes (maximal length is 512 bytes)
pB key Object ID	7	pB key Object ID Public share (key) pB computed by responder stored as data in key object
cA key Object ID	8	cA key Object ID Confirmation message cA stored as data in key object
cB key Object ID	9	cB key Object ID Confirmation message cB stored as data in key object
Ke key Object ID	10	Ke key Object ID Computed shared secret Ke stored as data in key object
Response message format		
Parameter	**RRx number**	**Description**
Message Header	0	0x8600xxxx, where xxxx: 0x3c3c: in case of command success 0xc3c3: in case of command fail

The key object parameters can be seen in table below:

Table 150. Key object parameters

Key Object Name	Key type	Key size [bits]
y (private key)	Asymmetric key pair	256 (P-256)
pA	Asymmetric public key	256 (P-256)
w0	Symmetric key	256
L	Asymmetric public key	256 (P-256)
pB	Asymmetric public key	256 (P-256)
cA	Symmetric key	256
cB	Symmetric key	256
Ke	Symmetric key	256

8.5.33 KEY_STORE_ INIT

This command initializes the user key store. Only one key store context can be opened at a time.

Table 151. KEY_STORE_CONTEXT_INIT command format

Command ID	0x49	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	2/1
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x49020017	
Session ID	1	Session ID	
User key store ID	2	User key store ID: Any 32-bit user value to identify key store. NOTE • When user key store ID=0, this command can be called just once. Any repeated call returns error. • When user key store ID is non-zero, this command can be called repeatedly. When this command is called repeatedly with the same user key store ID, it returns the key store ID of the initialized key store. This feature allows ELE sharing among different masters. Any 32-bit value used by user to identify key store. Because only one key store is supported, this value can be kept as 0.	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x4901xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	
Key store context ID	1	Key store context ID	

8.5.34 KEY_STORE_FREE

This command deletes all keys stored in the key store and de-initializes the key store. If the key store is not initialized, the command returns error.

Table 152. KEY_STORE_FREE command format

Command ID	0x76	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/0
Security level	The same or higher security level used with the KEY_STORE_INIT command		
Command message format			

Table continues on the next page...

Table 152. KEY_STORE_FREE command format (continued)

Parameter	ELEMUA_TRn number	Description
Message header	0	0x76010017
Key store ID	1	Key store ID
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x7600xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

8.5.35 KEY_STORE_SET_KEY

This command stores plaintext key/key pair into key store if allowed by key properties (see Key object properties). In case of asymmetric ECC key, user can select whether private/public part or key pair will be stored. For the key pair, user needs to store the key data in the following data format:

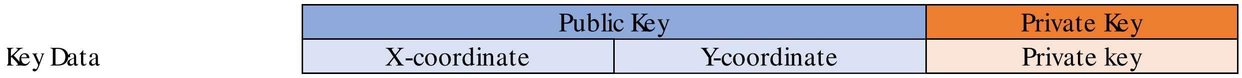


Figure 53. Asymmetric key format

Table 153. KEY_STORE_SET_KEY command format

Command ID	0x4c	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	6/0
Security level	The same or higher security level as key object		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x4C060017	
Key store ID	1	Key store ID	
Key object ID	2	Key object ID	
Key data	3	Start address of buffer with plaintext key data	
Key data Length	4	Key data length in bytes	
Key bit length	5	Key length in bits	
Key part	6	Key part: 0x00000001: Symmetric key 0x00000002: Public part of asymmetric key 0x00000003: Private part of asymmetric key 0x00000004: Asymmetric key pair	

Table continues on the next page...

Table 153. KEY_STORE_SET_KEY command format (continued)

Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x4c00xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: command fail

8.5.36 KEY_STORE_GET_KEY

This command reads a key in plaintext from the key store if allowed by key properties (see Key object properties). In case of asymmetric ECC key, user can select whether private/public part or key pair will be read in respect which key part is stored in the key object. The public part reading of ECC key is implicitly enabled. For the key pair, the command returns key data in Asymmetric key format

Table 154. KEY_STORE_GET_KEY command format

Command ID	0x4e	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	6/1
Security level	The same or higher security level as key object		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x4E060017	
Key store ID	1	Key store ID	
Key object ID	2	Key object ID	
Key data	3	Start address of key data buffer	
Key data Length	4	Key data length in bytes NOTE The buffer size should be bigger than the biggest key size expected to store. KEY_STORE_GET_KEY is used to return real key size.	
Key bit Length buffer	5	Pointer pointing to 32-bit unsigned word where ELE will write the size of the key in bits.	
Key part	6	Key part: 0x00000001: Symmetric key 0x00000002: Public part of asymmetric key 0x00000003: Private part of asymmetric key 0x00000004: Asymmetric key pair	
Response message format			

Table continues on the next page...

Table 154. KEY_STORE_GET_KEY command format (continued)

Parameter	ELEMUA_RRn number	Description
Message header	0	0x4E01xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Key length	1	Number of bytes written to the key data buffer

8.5.37 KEY_STORE_EXPORT_KEY

This command exports a key/key pair as key blob if allowed by key properties (see chapter Key Object Properties).

Table 155. KEY_STORE_EXPORT_KEY command format

Command ID	0x79	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	5/1
Security level	The same or higher security level as key object		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x79050017	
Key store ID	1	Key store ID	
Key object ID	2	Exported key object ID	
Blob data	3	Start address of buffer for blob data	
Blob data length	4	Blob data buffer size in bytes: ELE die unique blob: key size + 24 bytes ELE to ELE blob: key size + 24 bytes NBU ESK blob: key size NBU EIRK blob: key size	
Key blob type	5	Key blob type: 0x00000001: ELE die unique blob 0x00000002: ELE to ELE blob 0x00000003: NBU ESK blob 0x00000004: NBU EIRK blob NOTE Refer to Bluetooth ELE key export for details of NBU blob.	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	

Table continues on the next page...

Table 155. KEY_STORE_EXPORT_KEY command format (continued)

Message header	0	0x7901xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Blob data length	1	Number of bytes written to the blob data buffer

8.5.38 KEY_STORE_IMPORT_KEY

This command imports a key/key pair blob into the ELE internal key store if allowed by key properties (see chapter Key Object Properties).

Table 156. KEY_STORE_IMPORT_KEY command format

Command ID	0x78	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	5/0
Security Level	The same or higher security level as key object		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x78050017	
Key store ID	1	Key store ID	
Key object ID	2	Imported key object ID	
Blob data	3	Start address of buffer with blob data	
Blob data length	4	Blob data buffer size in bytes: ELE die unique blob: key size + 24 bytes ELE to ELE blob: key size + 24 bytes NBU ESK blob: key size NBU EIRK blob: key size	
Key blob type	5	Key blob type: 0x00000001: ELE die unique blob 0x00000002: ELE to ELE blob	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x7800xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	

8.5.39 KEY_STORE_GENERATE_KEY

This command generates a random key/key pair. This command requires high quality random number for correct operation. Therefore, the MGMT_GET_RANDOM command has to be called with non-zero Random Number Quality parameter before the KEY_STORE_GENERATE_KEY command is executed.

Table 157. Table 31. KEY_STORE_GENERATE_KEY command format

Command ID	0x4D	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	3/0
Security level	The same or higher security level as key object		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x4D030017	
Key store ID	1	Key store ID	
Key object ID	2	Key object ID	
Key bit length	3	Key size in bits For symmetric key, the key size is limited by the key store size. For asymmetric key, the key size is limited to 192, 224, 256, 384 and 512, which are the ECC curve sizes supported.	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x4D00xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	

8.5.40 KEY_STORE_OPEN_KEY

This command manages secure transfer of ELE keys or other sensitive data from ELE to different SoC peripheral. This command has two format based on number of needed parameters.

Table 158. KEY_STORE_OPEN_KEY command format 1

Command ID	0x4F	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	2/0
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x4F020017	
Key store ID	1	Key store ID	
Key ID	2	ID of internal key:	

Table continues on the next page...

Table 158. KEY_STORE_OPEN_KEY command format 1 (continued)

		0x80000007: NPX keys 0x80000009: NBU_DKEY_SK blob key 0x8000000A: NBU_DKEY_IRK blob key
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x4F00xxxx, where xxxx: 0x3C3C: in case of command success 0xC3C3: in case of command fail

Table 159. KEY_STORE_OPEN_KEY command format 2

Command ID	0x4F	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	3/0
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x4F020017	
Key store ID	1	Key store ID	
Key ID	2	ID of internal key: 0x8000000B: NBU BRICK KEY0 key 0x8000000C: NBU BRICK KEY1 key	
Key object ID	3	Key object ID	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x4F00xxxx, where xxxx: 0x3C3C: in case of command success 0xC3C3: in case of command fail	

8.5.40.1 NPX keys

The command with NPX keys parameters initialize NPX module with four device die unique keys.

8.5.40.2 NBU_DKEY_SK and NBU_DKEY_IRK blob keys

The command with NBU_DKEY_SK or NBU_DKEY_IRK parameters generates random 16 bytes DKEY SK or DKEY IRK blob keys. Then the key is sent via private key bus into NBU module. Both keys are used as blob encryption keys for key export, see KEY_STORE_EXPORT_KEY command.

8.5.40.3 NBU_BRICK_KEY0 and NBU_BRICK_KEY1 keys

The command sends any key defined by Key Object ID into NBU BRIC module KEY0 or KEY1 registers. The key sent into BRIC module must have CRYPTO_PKB key attribute set, key type is limited to symmetric key and key size must be 128 bits only. For more details see NBU module description.

8.5.41 KEY_STORE_ERASE_KEY

This command erases the unlocked key.

NOTE

This command does not free key object or allocated key slot. The KEY_OBJECT_FREE command is used to free key object and KEY_STORE_ALLOCATE_HANDLE is used to allocate the key slot.

Table 160. KEY_STORE_ERASE_KEY command format

Command ID	0x51	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	2/0
Security level	The same or higher security level as key object		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x51020017	
Key store ID	1	Key store ID	
Key object ID	2	Key object ID	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x5100xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	

8.5.42 KEY_STORE_GET_PROPERTY

This command returns the configured and actual key store property (key store data size, maximum key objects number).

Table 161. KEY_STORE_GET_PROPERTY command format

Command ID	0x77	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	2/1
Security level	The same or higher security level used with the KEY_STORE_INIT command		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x77020017	
Key store ID	1	Key store ID	

Table continues on the next page...

Table 161. KEY_STORE_GET_PROPERTY command format (continued)

Key store property ID	2	Key store property ID: 0x00000000: Total key store data memory in bytes 0x00000001: Currently available key store data memory 0x00000002: Total number of key objects 0x00000003: Number of currently available key objects
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x7701xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Key length	1	Requested key store property value

8.5.43 KEY_OBJECT_INIT

This command initializes a key object and returns key object ID. The key object is used as a reference (handle) to the keys stored in the key store.

Table 162. KEY_OBJECT_INIT command format

Command ID	0x41	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/1
Security level	The same or higher security level as key object		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x41010017	
Key store ID	1	Key store ID	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x4101xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	
Key object ID	1	Key object ID	

8.5.44 KEY_OBJECT_ALLOCATE_HANDLE

This command allocates a key slot in the key store for the key object. The key object can hold any key with size equal to or smaller than the key slot size.

Table 163. KEY_OBJECT_ALLOCATE_HANDLE command format

Command ID	0x42	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	6/0
Security level	The same or higher security level as key object		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x42060017	
Key object ID	1	Key object ID	
User key ID	2	User key ID: Bit31: Reserved. It should be set as "0" for future compatibility Bits 30-0: Any 31-bit user value to identify key NOTE User key ID has to be unique number except ID=0. The ID = 0 can be used for any number of key object. If key object has User key ID = 0, user cannot use command KEY_OBJECT_GET_HANDLE to get key object ID.	
Key part	3	Key part: 0x00000001: Symmetric key 0x00000002: Public part of asymmetric key 0x00000003: Private part of asymmetric key 0x00000004: Asymmetric key pair	
Key cipher type	4	Key cipher type: 0x00000010: Symmetric key 0x00000032[1]: RSA CRT in CRT format 0x00000033[1]: RSA CRT in CRT compact format 0x00000040: Asymmetric ECC NIST-P 0x00000042: Asymmetric ECC Brainpool r1 curves 0x00000043: Asymmetric ECC Brainpool t1 curves 0x00000050: Asymmetric ECC MONTGOMERY 0x00000051: Asymmetric ECC TWISTED EDWARDS 0x00000060[1]: Asymmetric OSCCA SM2 key 0x00000070: Asymmetric ECC GF(2m) (Koblitz) curves 0x00000080[1]: Asymmetric Dilithium2 key 0x00000081[1]: Asymmetric Dilithium3 key	

Table continues on the next page...

Table 163. KEY_OBJECT_ALLOCATE_HANDLE command format (continued)

		0x00000082[1]: Asymmetric Dilithium5 key
Key slot size	5	Key slot size Size of key slot in bytes, which is allocated in the key store for the key. The slot size has to be word (32-bit) size aligned. If not, the slot size is automatically rounded up to be word aligned. The keys slot size has to be at least equal to or larger than key data size. NOTE This command automatically allocates one more word (32-bit) above required key slot size. This additional word is used to store internal information about the key slot. This additional word is invisible to the user, but needs to be considered as consumed key store space. For example, if user allocates key slot of 16 bytes, then 20 bytes are consumed in total from the key store size.
Key properties	6	Key properties For details see Key object properties
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x4200xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

[1] This functionality is available with loaded additional ELE firmware only.

8.5.45 KEY_OBJECT_GET_HANDLE

This command returns the key object ID if a user key ID is found in the key store.

Table 164. KEY_OBJECT_GET_HANDLE command format

Command ID	0x43	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/1
Security level	The same or higher security level as key object with the corresponding user key ID		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x43010017	
User key ID	1	User key ID:	

Table continues on the next page...

Table 164. KEY_OBJECT_GET_HANDLE command format (continued)

		Bit31: Reserved. It should be set to "0" for future compatibility Bits 30-0: Any 31-bit user value to identify key. If User Key ID is 0, this command returns fail. See KEY_OBJECT_ALLOCATE_HANDLE for details.
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x4301xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Key object ID	1	Key object ID

8.5.46 KEY_OBJECT_GET_PROPERTIES

This command returns key object properties.

Table 165. KEY OBJECT_GET_PROPERTIES command format

Command ID	0x45	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/1
Security level	The same or higher security level as the key object		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x45010017	
Key object ID	1	Key object ID	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x4501xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	
Key object properties	1	Key object properties	

8.5.47 KEY_OBJECT_SET_PROPERTIES

This command sets key object properties.

Table 166. KEY_OBJECT_SET_PROPERTIES command format

Command ID	0x44	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	2/0

Table continues on the next page...

Table 166. KEY_OBJECT_SET_PROPERTIES command format (continued)

Security level	The same or higher security level as the key object	
Command message format		
Parameter	**ELEMUA_TRn number**	**Description**
Message header	0	0x44020017
Key object ID	1	Key object ID
Key object properties	2	Key object properties For more details see Key object properties.
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x4400xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

8.5.48 KEY_OBJECT_FREE

This command deletes a key object. It clears key data and deallocates key object including associated key slot. If option=0, the key store memory is not defragmented and the deallocated memory space cannot be used for a new key allocation. If option=1, the key store memory is defragmented and deallocated memory space is available for a new key allocation.

Table 167. KEY_OBJECT_FREE command format

Command ID	0x47	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	2/0
Security level	The same or higher security level as the key object		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x47020017	
Key object ID	1	Key object ID	
Options	2	Options: 0x0000000: The key store memory is not defragmented, and the deallocated memory space cannot be used for a new key allocation (static operation) 0x00000001: The key store memory is defragmented, and the deallocated memory space can be used for a new key allocation (dynamic operation)	

Table continues on the next page...

Table 167. KEY_OBJECT_FREE command format (continued)

		NOTE Setting this option to "0x00000001" defragments all deallocated key store memory. Thus user can control when the key store memory is deallocated/defragmented.
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x4700xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

8.5.49 MGMT_CONTEXT_INIT

This command initializes context for ELE management commands. Only one management context can be opened at a time.

Table 168. MGMT_CONTEXT_INIT command format

Command ID	0x65	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/1
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x65010017	
Session ID	1	Session ID	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x6501xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	
Management context ID	1	Management context ID	

8.5.50 MGMT_ADVANCE_LIFECYCLE

This command changes lifecycle state into the next state. The next allowed state depends on lifecycle state machine.

Table 169. MGMT_ADVANCE_LIFECYCLE command format

Command ID	0x60	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	4/0

Table continues on the next page...

Table 169. MGMT_ADVANCE_LIFECYCLE command format (continued)

Security level	If TZ-M is enabled, then secure privileged, else nonsecure privileged	
Command message format		
Parameter	**ELEMUA_TRn number**	**Description**
Message header	0	0x60040017
Management context ID	1	Management context ID
Life cycle	2	Requested life cycle: 0x01: Reserved for NXP internal use 0x03: Reserved for NXP internal use 0x07: OEM opened 0x0F: OEM secure world closed 0x1F: OEM closed 0x9F: OEM locked 0x3F: OEM field return[1] 0x7F: Reserved for NXP internal use
Options data	3	Address pointing to 32-bit value holding CPU_CLK clock frequency in MHz. Allowed values are 6, 16, 24, 32, 48, 64 or 96.
Options data length	4	Options Data length in bytes: has to be 4.
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x6000xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

[1] MGMT_SET_RETURN_FA_MODE must be used to transition to OEM field return instead of MGMT_ADVANCE_LIFECYCLE.

8.5.51 MGMT_FUSE_PROGRAM

This command performs ELE fuse programming. The fuse programming requires the CPU_CLK clock frequency in MHz, which is provided as option data.

Table 170. MGMT_FUSE_PROGRAM command format

Command ID	0x67	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	6/0
Security level	If TZ-M is enabled, then Secure Privileged, else Nonsecure Privileged		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	

Table continues on the next page...

Table 170. MGMT_FUSE_PROGRAM command format (continued)

Message header	0	0x67060017
Management context ID	1	Management context ID
Fuse ID	2	Fuse ID (see Fuses ID list)
Fuse value	3	Address pointing to 32-bit variable or buffer holding fuse value to be programmed.
Fuse value length	4	Unsigned 32-bit value containing size of fuse value buffer in bytes.
Options data	5	Address pointing to 32-bit value holding CPU_CLK bus frequency in MHz. Allowed values are 6, 16, 24, 32, 48, 64 or 96.
Options data length	6	Options data length in bytes: has to be 0x00000004.
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x6700xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

NOTE

The RO in the following table indicates that user cannot use MGMT_FUSE_PROGRAM to write the lifecycle fuses, instead, he can use MGMT_ADVANCE_LIFECYCLE or MGMT_SET_RETURN_FA_MODE.

Table 171. Fuses ID list

Fuse ID	Fuse name	Fuse size (bits)	Access
0	Reserved	—	—
1	Reserved	—	—
2	Reserved	—	—
3	Reserved	—	—
4	Reserved	—	—
5	CUST_PROD_OEMFW_AUTH_PUK_LOCK	3	R/W
6	CUST_PROD_OEMFW_ENC_SK_LOCK	3	R/W
7	Reserved	—	—
8	DCFG_CC_SOCU_L1_LOCK	3	R/W
9	DCFG_CC_SOCU_L2_LOCK	3	R/W
10	LIFECYCLE	8	RO
11	DBG_EN_LOCK	1	R/W
12	DBG_AUTH_DIS	1	R/W

Table continues on the next page...

Table 171. Fuses ID list (continued)

Fuse ID	Fuse name	Fuse size (bits)	Access
13	TZM_EN	1	R/W
14	DICE_EN	1	R/W
15	Reserved	—	—
16	Reserved	—	—
17	SERIAL_DIS	1	R/W
18	WAKEUP_DIS	1	R/W
19	CUST_PROD_OEMFW_AUTH_PUK_REVOKE	4	R/W
20	SWD_ID	4	R/W
21	DBG_AUTH_VU	16	R/W
22	IMG_KEY_REVOKE	16	R/W
23	Reserved	—	—
24	Reserved	—	—
25	Reserved	—	—
26	Reserved	—	—
27	Reserved	—	—
28	Reserved	—	—
29	Reserved	—	—
30	Reserved	—	—
31	CUST_PROD_OEMFW_AUTH_PUK	256	R/W
32	CUST_PROD_OEMFW_ENC_SK	256	WO
33	Reserved	—	—
34	DCFG_CC_SOCU_L1	32	R/W
35	DCFG_CC_SOCU_L2	32	R/W
36	Reserved	—	—
37	CM33_S_VER_CNT	64	R/W
38	CM33_NS_VER_CNT	256	R/W
39	RADIO_VER_CNT	128	R/W
40	SNT_VER_CNT	32	R/W
41	CM33_BOOTLOADER_VER_CNT	32	R/W
42	CM33_S_VER_CNT_VIRTUAL	n/a	R/W
43	CM33_NS_VER_CNT_VIRTUAL	n/a	R/W
44	RADIO_VER_CNT_VIRTUAL	n/a	R/W

Table continues on the next page...

Table 171. Fuses ID list (continued)

Fuse ID	Fuse name	Fuse size (bits)	Access
45	SNT_VER_CNT_VIRTUAL	n/a	R/W
46	CM33_BOOTLOADER_VER_CNT_VIRTUAL	n/a	R/W
47	Reserved	—	—
48	Reserved	—	—
49	Reserved	—	—
50	Reserved	—	—
51	NBU_SEC_BOOT_EN	4	R/W
52	PQC_EN	2	R/W
53	OEM_SEC_BOOT_EN	4	R/W
54			
55			
56	SECURE_STORAGE_VER_CN	96	R/W
57	SECURE_STORAGE_VER_CNT_VIRTUAL	n/a	R/W

8.5.52 MGMT_FUSE_READ

This command performs ELE fuse read. The fuse reading requires the CPU_CLK clock frequency in MHz, which is provided as option data.

Table 172. MGMT_FUSE_READ command format

Command ID	0x68	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	6/0
Security level	If TZ-M is enabled, then secure privileged, else nonsecure privileged		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x68060017	
Management context ID	1	Management context ID	
Fuse ID	2	Fuse ID (see Table 171)	
Fuse value	3	Address pointing to 32-bit variable or buffer used to load the requested fuse value.	
Fuse value length	4	Address pointing to unsigned 32-bit variable containing maximum size of Fuse value buffer in bytes. This value will be overwritten with the actual number of bytes written to the fuse value buffer. For virtual counter fuses, the expected Length value is fixed to 4 bytes.	

Table continues on the next page...

Table 172. MGMT_FUSE_READ command format (continued)

Options data	5	Address pointing to 32-bit value holding CPU_CLK clock frequency in MHz. Allowed values are 6, 16, 24, 32, 48, 64 or 96.
Options data length	6	Options data length in bytes: has to be 4.
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x6800xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

8.5.53 MGMT_GET_LIFECYCLE

This command returns actual device lifecycle.

Table 173. MGMT_GET_LIFECYCLE command format

Command ID	0x6B	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/1
Security level	If TZ-M is enabled, then secure privileged, else nonsecure privileged		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x6B010017	
Management context ID	1	Management context ID	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x6B01xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	
Life Cycle	1	Actual device life cycle 0x01: Reserved for NXP internal use. 0x03: Reserved for NXP internal use. 0x07: OEM opened 0x0F: OEM secure world closed 0x1F: OEM closed 0x9F: OEM locked 0x3F: Reserved for NXP internal use. 0x7F: Reserved for NXP internal use.	

8.5.54 MGMT_GET_PROPERTY

This command returns security device properties, which are managed by ELE. For successful command operation, the correct property data length must be provided.

Table 174. MGMT_GET_PROPERTY command format

Command ID	0x6C	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	4/1
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x6C040017	
Management context ID	1	Management context ID	
Property ID	2	Property ID: 0x00000020: Image hash (data length = 48 bytes) 0x00000040: Last authentication state (data length = 4 bytes), the expected returns are: • 0x3C3CC3C3: Authentication succeed • 0x5A5AA500: Authentication failed 0x00000050: Image version (data length = 8 bytes) 0x00000051: ELE firmware version (data length = 8 bytes). If it returns 0xFFFFFFFFFFFFFFFF no firmware has been downloaded yet 0x00000060: Public portion of Attest authentication key (data length = 64 bytes) 0x00000090: UUID (data length = 16 bytes) 0x000000A0: Fast Wake Up Context (data length = 196 bytes) 0x000000B0: Number of repeated AES calculation (data length = 4 bytes)	
Property data buffer	3	Start address of property data buffer	
Property data buffer length	4	Property data buffer length in bytes	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x6C01xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail 0x1212: Command error: invalid argument 0x1B1B: Command error: unknown property ID	
Data length	1	Number of bytes written into property data buffer	

8.5.55 MGMT_SET_PROPERTY

This command sets security device properties, which are managed by ELE. For successful command operation, the correct property data length must be provided.

Table 175. MGMT_SET_PROPERTY command format

Command ID	0x71	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	4/0
Security level	If TZ-M is enabled, then secure privileged, else nonsecure privileged		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message Header	0	0x71040017	
Management context ID	1	Management context ID	
Property ID	2	Property ID: • 0x11: ELE RAM Power Control (data length = 4 bytes) Source data: **Bit0:** 0: PKC RAM disabled 1: PKC RAM enabled **Bit1:** 0: ELE firmware RAM power is OFF during sleep and deep sleep low power mode. 1: ELE firmware RAM power is ON during sleep and deep sleep low power mode. • 0xA0: Fast Wake Up Context (data length = 196 bytes) • 0xB0: Number of repeated AES calculation (data length = 4 bytes)	
Property data buffer	3	Start address of property source data buffer	
Property data buffer length	4	Property source data buffer length in bytes	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message Header	0	0x7100xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail 0x1212: Command error, invalid argument 0x1B1B: Command error, unknown property ID	

8.5.56 MGMT_SET_HOST_ACCESS_PERMISSION

This command controls secure/non-secure and privilege/user bus attributes, which host uses for all bus transactions performed during next ELE command execution. It can be executed any time the ELE is ready to receive a new command. An open session is not required.

The TrustZone implementation on this device requires the bus transaction security level to be equal to the security attribute of the target memory address. It means memory address configured as secure privilege can be accessed by secure-privilege bus owner only.

The ELE uses masquerading technique to inherit bus transaction security level for all data accesses to the SoC memory. It means that ELE uses the same bus security attribute as the host, which writes command into ELEMUA_TRn register. Said another way, if ELE command was initiated by secure privilege host, then the ELE behaves as secure privilege bus owner for all bus transactions performed within the current command execution. If the ELE command was initiated by non-secure privilege host, the ELE behaves as non-secure privilege bus owner for all the bus transactions.

However, there are situations, when host needs to perform crypto operation in memory with lower security attribute than host itself. For example, secure application wants to calculate digest over non-secure memory buffer. The access to the memory with lower security attribute is allowed using MGMT_SET_HOST_ACCESS_PERMISSION command. After this command is executed, the ELE does not use masquerading to inherit security level for upcoming bus transactions, but uses security level defined by the MGMT_SET_HOST_ACCESS_PERMISSION command. This forced security level is valid for subsequent command, executed after MGMT_SET_HOST_ACCESS_PERMISSION command only. For any other following commands, the ELE switches back to masquerading mode and inherits bus transaction security level from the host.

This command must be used also before KEY_OBJECT_INIT command, if the user needs to create key object with lower security level than host executing the KEY_OBJECT_INIT command.

Table 176. MGMT_SET_HOST_ACCESS_PERMISSION command format

Command ID	0x70	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/0
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x70010017	
Security level	1	Security level required for next command: 0x00000000: Non-secure user 0x00000001: Non-secure privileged 0x00000002: Secure user 0x00000003: Secure privileged NOTE Required security level must be equal to or lower than the security level of the host sending this command	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x7000xxxx, where xxxx:	

Table continues on the next page...

Table 176. MGMT_SET_HOST_ACCESS_PERMISSION command format (continued)

		0x3C3C: Command success 0xC3C3: Command fail

8.5.57 MGMT_SET_RETURN_FA_MODE

This command sets OEM/NXP field return analysis modes.

Table 177. MGMT_SET_RETURN_FA_MODE command format

Command ID	0x72	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	5/0
Security level	If TZ-M is enabled, then secure privileged, else nonsecure privileged		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x72050017	
Management context ID	1	Management context ID	
Source data buffer	2	Start address of the source buffer with field return analysis mode signed message	
Source data buffer length	3	Length of failure analysis mode request image in bytes (784 bytes)	
Options data	4	Address pointing to 32-bit value holding CPU_CLK clock frequency in MHz. Allowed values are 6, 16, 24, 32, 48, 64 or 96.	
Options length	5	Options data length in bytes: has to be 4.	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x7200xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	

8.5.58 MGMT_GET_RANDOM

This command returns a random number of the requested length. This command can be executed any time the ELE is ready to receive new command and without opening ELE session or management context. The first high quality random number request includes TRNG initialization.

NOTE

Some commands require high quality random number for their correct operation. Therefore this command has to be called with non-zero Random Number Quality parameter, before these commands are executed.

Table 178. MGMT_GET_RANDOM command format

Command ID	0x73	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	3/0
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x73030017	
Random number quality	1	0x00000000: Best available quality. If TRNG was already initialized by previous MGMT_GET_RANDOM request, it returns high-quality number. If not, it returns low quality number. 0x00000033: Low quality random number. Any other value: High quality random number. First high-quality number request includes TRNG initialization. Therefore, first request takes longer time due this initialization. If random data length is zero, TRNG initialization is executed only.	
Random number data buffer	2	Start address of random number data buffer	
Random number length	3	Length of requested random data in bytes (length of random number data buffer)	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x7300xxyy, where xx: 0x33: Low quality number returned 0x55: High quality number returned Any other value means fail. and yy: 0x3C: Command success 0xC3: Command fail Any other value means fail.	

8.5.59 MGMT_CLEAR_ALL_KEYS

This command overwrites complete key store with the random numbers. But it does not delete or free any of the key objects. The KEY_OBJECT_FREE command can be used to delete the key object.

Table 179. MGMT_CLEAR_ALL_KEYS command format

Command ID	0x74	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	1/0

Table continues on the next page...

Table 179. MGMT_CLEAR_ALL_KEYS command format (continued)

Security level	If TZ-M is enabled, then Secure Privileged, else Nonsecure Privileged	
Command message format		
Parameter	**ELEMUA_TRn number**	**Description**
Message header	0	0x74010017
Management context ID	1	Management context ID
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x7400xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

8.5.60 UKNOWN_COMMAND response message

The ELE sends this message in case the command ID is not recognized or not supported.

Table 180. UKNOWN_COMMAND message format

Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message Header	0	0xXX001616, where XX is command ID sent in command message and which was not recognized.

8.6 ELE code example

This code example demonstrates ELE usage for AES CBC operation. The AES CBC operation is performed in the following steps:

1. Opens ELE session
2. Creates key store
3. Creates and allocate key object
4. Sets the key
5. Initializes AES CBC operation context
6. Performs AES CBC operation
7. Closes all opened contexts and created objects

NOTE

This example does not close already opened contexts or objects in case of failed command.

```
uint32_t  sessionID;
uint32_t  keyStoreContextID;
uint32_t  keyStoreID;
uint32_t  keyObjectID;
uint32_t  aesOperationID;
uint32_t  temp;
//KEY = 1f8e4973953f3fb0bd6b16662e9a3c17
```

```
uint8_t symKeyData[16] = {0x1f, 0x8e, 0x49, 0x73, 0x95, 0x3f, 0x3f, 0xb0, 0xbd, 0x6b, 0x16, 0x66,
0x2e, 0x9a, 0x3c, 0x17};
// IV = 2fe2b333ceda8f98f4a99b40d2cd34a8
uint8_t ivData[16] = {0x2f, 0xe2, 0xb3, 0x33, 0xce, 0xda, 0x8f, 0x98, 0xf4, 0xa9, 0x9b, 0x40, 0xd2,
0xcd, 0x34, 0xa8};
//PLAINTEXT = 45cf12964fc824ab76616ae2f4bf0822
uint8_t plainData[16] = {0x45, 0xcf, 0x12, 0x96, 0x4f, 0xc8, 0x24, 0xab, 0x76, 0x61, 0x6a, 0xe2,
0xf4, 0xbf, 0x08, 0x22};
//CIPHERTEXT = 0f61c4d44c5147c03c195ad7e2cc12b2
uint8_t cipherDataRef[16] = {0x0f, 0x61, 0xc4, 0xd4, 0x4c, 0x51, 0x47, 0xc0, 0x3c, 0x19, 0x5a, 0xd7,
0xe2, 0xcc, 0x12, 0xb2};
uint8_t cipherData[16] = {0};

do
{
  /* OPEN_SESSION command - Open EdgeLock enclave session */
  ELEMUA_TR0 = 0x13020017;
  ELEMUA_TR1 = 0x2;    // EdgeLock ID
  ELEMUA_TR2 = 0;      // Session User ID
  while ((ELEMUA_RSR & 0x3) != 0x3)
  {}
  if (ELEMUA_RR0 == 0x13013C3C)
  {
    sessionID = ELEMUA_RR1;
  }
  else
  {
    result = FAIL;
    break;
  }

  /* KEY_STORE_INIT command - Initialize key store */
  ELEMUA_TR0 = 0x49020017;
  ELEMUA_TR1 = sessionId;
  ELEMUA_TR2 = 0; // User ID=0
  while ((ELEMUA_RSR & 0x3) != 0x3)
  {}
  if (ELEMUA_RR0 == 0x49013C3C)
  {
    keyStoreContextID = ELEMUA_RR1;
  }
  else
  {
    result = FAIL;
    break;
  }

  /* KEY_OBJECT_INIT command - Key object init */
  ELEMUA_TR0 = 0x41010017;
  ELEMUA_TR1 = keyStoreContextID;
  while ((ELEMUA_RSR & 0x3) != 0x3)
  {}
  if (ELEMUA_RR0 == 0x41013C3C)
  {
    keyObjectID = ELEMUA_RR1;
  }
  else
  {
    result = FAIL;
    break;
  }

  /* KEY_OBJECT_ALLOCATE_HANDLE command -  Allocate slot in key store */
  ELEMUA_TR0 = 0x42060017;
  ELEMUA_TR1 = keyObjectID;
  ELEMUA_TR2 = 0;    // key user ID=0
  ELEMUA_TR3 = 0x01; // symmetric key part
  ELEMUA_TR4 = 0x10; // symmetric key cipher type
  ELEMUA_TR5 = 16;   // 16 byte key slot size
  ELEMUA_TR6 = 0x01; // Allow AES operation
  while ((ELEMUA_RSR & 0x1) != 0x1)
  {}
  if (ELEMUA_RR0 != 0x42003C3C)
  {
    result = FAIL;
    break;
  }

  /* KEY_STORE_SET_KEY command - Set key */
  ELEMUA_TR0 = 0x4c060017;
  ELEMUA_TR1 = keyStoreContextID;
  ELEMUA_TR2 = keyObjectID;
  ELEMUA_TR3 = &symKeyData;
  ELEMUA_TR4 = 16;    // key data buffer size
  ELEMUA_TR5 = 128;   // key length in bits
  ELEMUA_TR6 = 0x01;  // key part
  while ((ELEMUA_RSR & 0x1) != 0x1)
  {}
  if (ELEMUA_RR0 != 0x4c003C3C)
  {
    result = FAIL;
    break;
  }

  /* SYMMETRIC_CONTEXT_INIT command - Initialize AES operation */
  ELEMUA_TR0 = 0x25040017;
  ELEMUA_TR1 = sessionID;
  ELEMUA_TR2 = keyObjectID;
  ELEMUA_TR3 = 1;    // AES CBC selected
  ELEMUA_TR4 = 0;    // encryption mode
  while ((ELEMUA_RSR & 0x3) != 0x3)
  {}
  if (ELEMUA_RR0 == 0x25013C3C)
  {
    aesOperationID = ELEMUA.RR1;
  }
  else
  {
    result = FAIL;
    break;
  }

  /* CIPHER_ONE_GO command - Perform AES CBC operation */
  ELEMUA_TR0 = 0x23060017;
  ELEMUA_TR1 = aesOperationID;
  ELEMUA_TR2 = &ivData;    // IV data
  ELEMUA_TR3 = 16;   // IV length -16 bytes
  ELEMUA_TR4 = &plainData;  // Source data - data to be encrypted
  ELEMUA_TR5 = &cipherData; // Destination data - encrypted data
  ELEMUA_TR6 = 16;     // Data length in bytes
```

```
  while ((ELEMUA_RSR & 0x1) != 0x1)
  {}
  if (ELEMUA_RR0 != 0x23003C3C)
  {
    result = FAIL;
    break;
  }

  /* CONTEXT_FREE command - Free operation context */
  ELEMUA_TR0 = 0x15010017;
  ELEMUA_TR1 = aesOperationID;
  while ((ELEMUA_RSR & 0x1) != 0x1)
  {}
  if (ELEMUA_RR0 != 0x15003C3C)
  {
    result = FAIL;
    break;
  }

  /* KEY_OBJECT_FREE command - Free key object */
  ELEMUA_TR0 = 0x47020017;
  ELEMUA_TR1 = keyObjectID;
  ELEMUA_TR2 = 0;    // no key store defragmentation

  while ((ELEMUA_RSR & 0x1) != 0x1)
  {}
  if (ELEMUA_RR0 != 0x47003C3C)
  {
    result = FAIL;
    break;
  }

  /* KEY_STORE_FREE command - Free key store context */
  ELEMUA_TR0 = 0x76010017;
  ELEMUA_TR1 = keyStoreContextID;
  while ((ELEMUA_RSR & 0x1) != 0x1)
  {}
  if (ELEMUA_RR0 != 0x76003C3C)
  {
    result = FAIL;
    break;
  }

  /* CLOSE_SESSION command - Close session context */
  ELEMUA_TR0 = 0x14010017;
  ELEMUA_TR1 = sessionID;
  while ((ELEMUA_RSR & 0x1) != 0x1)
  {}
  if (ELEMUA_RR0 != 0x14003C3C)
  {
    result = FAIL;
    break;
  }
} while (0)
```

Chapter 9
ELE Messaging Unit (ELEMU)

9.1 Chip-specific ELE Messaging Unit information

Table 181. Reference links to related information[1]

Topic	Related module	References
Full description	ELE Messaging Unit	ELE Messaging Unit (MU)
System memory map		See the section "System memory map"
CLocking		See the chapter "Clocking"
Signal multiplexing	Port control	See the chapter "Signal Muxing and Pinout "

[1] For all the reference sections and chapters mentioned in this table, refer the Reference Manual.

9.1.1 Module instance

This device has one instance of the ELEMU module.

9.2 Overview

The Messaging Unit module enables two processing elements within the SoC to communicate and coordinate by passing messages (e.g., data, status and control) through its interfaces. The EdgeLock Enclave Messaging Unit (ELE_MU) is specifically targeted for use in Edgelock enclave. The ELE_MU also provides the ability for one processing element to signal the other processing element using interrupts based on data transmission status.

This module's design is based on synchronous clock domain crossings, that is, each side of the ELE_MU operates in a phase-aligned clock domain. The two clock domains may be operating at integer multipliers or dividers, but they are phase aligned so there is no need for logic to handle asynchronous clock domain crossings. The ELE_MU accomplishes synchronization using two sets of matching registers (Processor A-facing, Processor B-facing).

In the Edgelock enclave application, the ELE_MU "A-side" port corresponds to the SoC host processor while the ELE_MU "B-side" port is the security subsystem.

Throughout this chapter, "ELE_MUA" is used to denote hardware resources associated with the "A-side" port, and "ELE_MUB" denotes those associated with the "B-side" port. This manual only documents the MU "A-side" programming model as it is visible and accessible to the host processor, while the MU "B-side" programming model is restricted to the internal operation of the EdgeLock enclave and not visible nor accessible to the host processor.

Finally, the ELE_MUA port implements a secure semaphore (SEMA4) function to provide hardware enforcement of the Edgelock enclave serialization mechanism in multi-core/multi-host device configurations.

9.2.1 Block Diagram

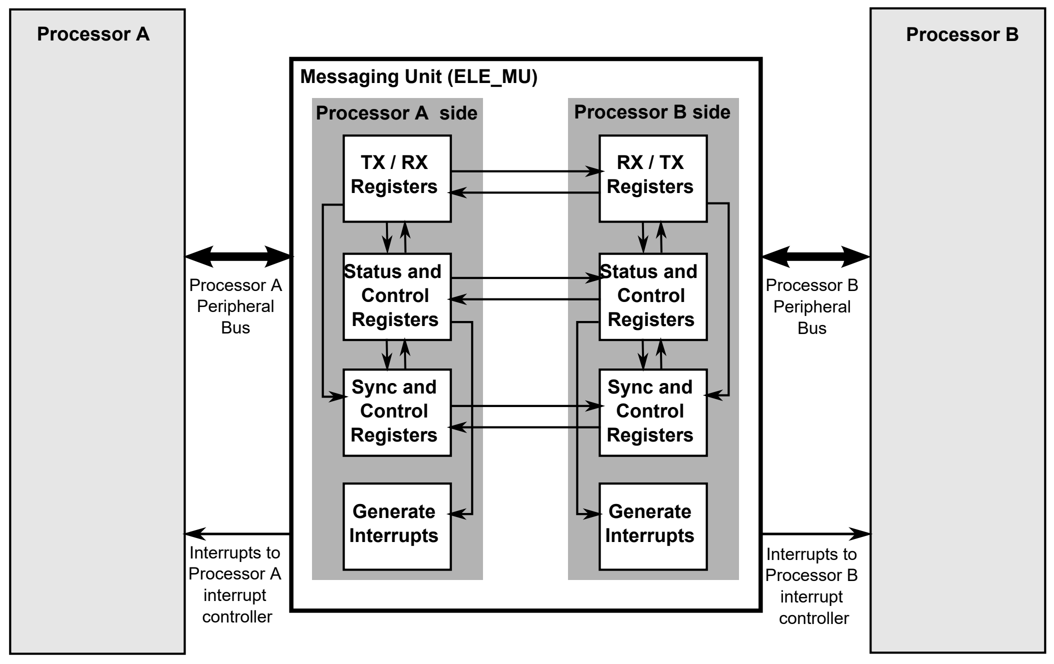

Processor A
Processor B
Messaging Unit (ELE_MU)
Processor A side
Processor B side
TX / RX Registers
RX / TX Registers
Status and Control Registers
Status and Control Registers
Sync and Control Registers
Sync and Control Registers
Generate Interrupts
Generate Interrupts
Processor A Peripheral Bus
Processor B Peripheral Bus
Interrupts to Processor A interrupt controller
Interrupts to Processor B interrupt controller

Figure 54. ELE_MU Block Diagram

9.2.2 Features

The ELE_MU includes the following features:

1. Memory-Mapped Registers
   - The ELE_MU is connected with separate peripheral buses on Processor A-side and Processor B-side.
2. Synchronous Message Transfers between Processing Elements
   - For sending data or messages between the processing elements, ELE_MUA provides 16 transmit registers and 2 receive registers, while ELE_MUB provides 2 transmit registers and 16 receive registers.
   - The transfer of data messages between processing elements uses transmit empty and receive full flags provided on both sides of the ELE_MU.
   - The update of these transmit and receive flags is accomplished using a fully synchronous mechanism. Upon a register read or write, the corresponding transmit or receive flags are updated at the completion of the register access cycle.
3. Secure Semaphore
   - The ELE_MUA port implements a semaphore function to support hardware enforcement of Edgelock enclave (ELE) serialization mechanisms in certain device configurations. This logic also provides read-only information on the ELE status.

9.3 Functional Description

The Messaging Unit (ELE_MU) enables two processing elements (Processor A and Processor B) to communicate with each other, by passing message/data information to each other, and by enabling one side to wake up the other side using data transmission interrupts.

The messaging, control, and status registers of the Processor A and Processor B sides for the ELE_MU are mapped to the processing element A and processing element B address spaces using a standard peripheral bridge memory slot. The peripheral data bus is 32 bits wide inside the ELE_MU module.

Most of the messaging mechanisms are symmetric. They are duplicated and are available on both the A-side and the B-side. The messaging mechanisms are:

- Sixteen 32-bit ELE_MUA transmit registers, which are each reflected in sixteen read-only ELE_MUB receive registers. These registers can be used to transfer 32-bit word messages or pass information for messages written to the shared memory (number of words, address pointers, and message type code).
- Two 32-bit ELE_MUB transmit registers, which are each reflected in two read-only ELE_MUA receive registers. These registers can be used to transfer 32-bit word messages or pass information for messages written to the shared memory (number of words, address pointers, and message type code).
- A write to a transmit register clears the corresponding “transmitter empty” bit in the Status Register on the transmitter side, and sets the appropriate “receiver full” bit in the Status Register on the receiver side. The setting of the bit at the receiver side can optionally trigger an interrupt at the receiver side (maskable receive interrupt).
- A read of one of the receive registers clears the corresponding “receiver full” bit in the Status Register at the receiver side, and sets the appropriate “transmitter empty” bit in the Status Register on the transmitter side. The setting of the “transmitter empty” bit can optionally trigger an interrupt at the transmitter side (maskable transmit interrupt).

A write to a transmit register signals the receiver side that data is ready for retrieval.

- Writing to the transmit register again without verifying that the data was retrieved is prohibited, because the transmitter side has no way of knowing the exact time that the receiver retrieves the data.
- Before attempting to write the transmit register again, the transmitter side waits for a “Transmitter Empty” interrupt, or polls the “Transmitter Empty” bit in the Transmit Status Register.

A read of a receive register signals the transmitter side that new data can be written to that register. In the same way, the receiver processor should not read a receive register before receiving a “Receiver Full” interrupt or polling the “Receiver Full” bit in the Receive Status Register.

- Reading the receive register again without verifying that the data was written is prohibited, because the receiver side has no way of knowing the exact time that the transmitter writes the data.
- Before attempting to read the receive register again, the receiver side waits for a “Receiver Full” interrupt, or polls the “Receiver Full” bit in the Receive Status Register.

9.3.1 Hardware Semaphore

The ELE_MU includes the following features:

To provide a mechanism to serialize ownership of the Edgelock enclave subsystem, ELE_MUA implements a hardware semaphore (SEMA4) register. Typically, and especially in multi-core host configurations, when a task wants to send crypto service requests to Edgelock enclave, it first establishes ownership of the subsystem by acquiring the semaphore. After the crypto task requests are completed, the host task relinquishes ownership of the semaphore, making the hardware available to other tasks.

Once the semaphore is acquired (locked), only writes to the ELE_MUA TR[0-14] registers from the semaphore owner are allowed; all other attempted writes are error terminated.

In addition to the physical 32-bit SEMA4 register, the ELE_MUA logic supports five read-only 32-bit virtual register programming model. The registers include a Edgelock enclave status register, the secure semaphore owner identity, operations to acquire and release the semaphore, plus a mechanism to force the release of the semaphore.

The physical register information includes the locked/unlocked semaphore state plus the domain identifier and the secure/nonsecure, privileged/user access mode attributes of the locking task.

Accesses to the five virtual registers can be classified into two groups:

1. Two simple register reads that return status (SEMA4_SR) and the secure 32-bit semaphore owner value (OWNR).

2. Three (atomic) compound W/R operations (ACQ, REL, FREL) that potentially update the semaphore state and then return the new secure owner value.

The three compound virtual register operations first perform a “write” and then return the standard 32-bit semaphore owner value. The owner value returns one of three possible values: 0x0000_0000 if the semaphore is unlocked, 0x0000_0055 if the semaphore is owned by the requesting host task (based on domain identifier and the 2-bit {secure/nonsecure, privileged/user} access mode attributes) and 0xFFFF_FFAA is the semaphore is locked but not by the requesting host task.

For the compound W/R operations, the (virtual) register IPS address encodes the write operation to be performed in the address. The actual bus transaction from the initiating host task is always a simple 32-bit data read.

9.4 Register Definition

The ELE_MU provides transmit and receive data registers for the communication between Processor A and Processor B. Some control and status registers to the Processor A and Processor B sides are for control operations (such as interrupts), and for status checking of the other ELE_MU-side. The following diagram shows the ELE_MU registers schematic.

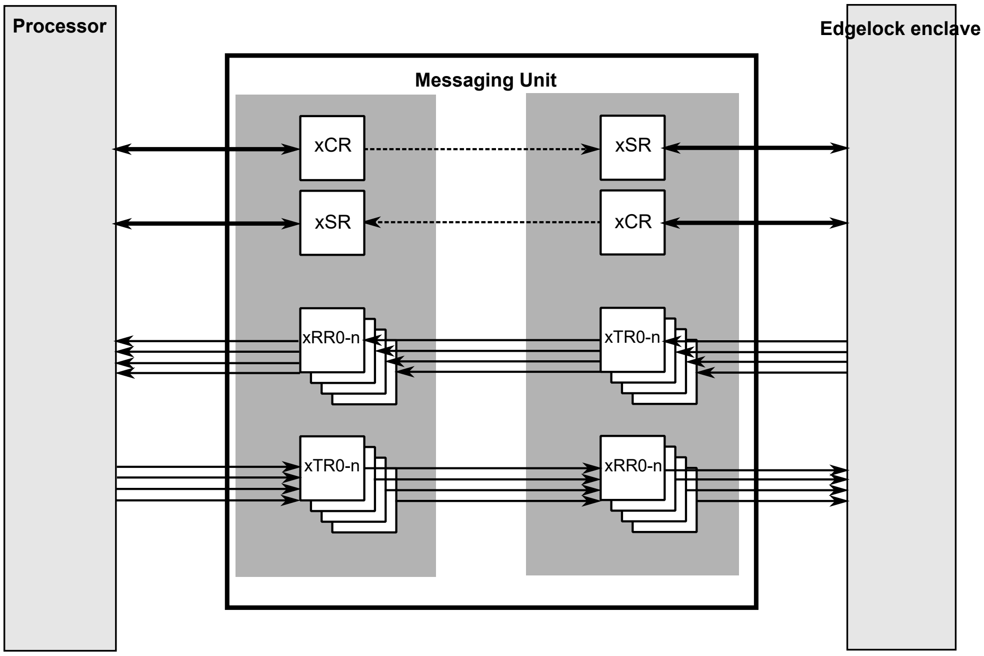


Figure 55. ELE_MU Registers

The detailed ELE_MU register definition can be found below.

NOTE

Read of a write-only, read zero (WORZ) register returns 0. Writes to a read-only (RO) register are ignored. Byte/halfword accesses are supported.

A read/write access to any illegal location of the ELE_MU generates a bus transfer error acknowledge to the Processor A or Processor B. Writes to "RSVD" registers are ignored and reads to "RSVD" registers return 0.

9.4.1 ELE_MUA register descriptions

This section contains the detailed register descriptions for the ELE_MUA registers.

9.4.1.1 ELE_MUA memory map

ELEMU.ELE_MUA base address: 4002_4000h

Offset	Register	Width (In bits)	Access	Reset value
0h	Version ID Register (VER)	32	R	0100_0000h
4h	Parameter Register (PAR)	32	R	0000_0210h
8h	Unused Register 0 (UNUSED0)	32	R	0000_0000h
Ch	Status Register (SR)	32	R	0000_0020h
120h	Transmit Control Register (TCR)	32	RW	0000_0000h
124h	Transmit Status Register (TSR)	32	R	0000_0000h
12Ch	Receive Status Register (RSR)	32	R	0000_0000h
1FCh	Unused Register 1 (UNUSED1)	32	RW	0000_0000h
200h - 23Ch	Transmit Register (TR0 - TR15)	32	RW	0000_0000h
280h - 284h	Receive Register (RR0 - RR1)	32	R	0000_0000h
400h	Semaphore Status Register (SEMA4_SR)	32	R	0000_0000h
474h	Semaphore Ownership Register (SEMA4_OWNR)	32	R	0000_0000h
998h	Semaphore Acquire Register (SEMA4_ACQ)	32	R	0000_0000h
ACCh	Semaphore Release Register (SEMA4_REL)	32	R	0000_0000h
BA4h	Semaphore Forced Release Register (SEMA4_FREL)	32	R	0000_0000h

9.4.1.2 Version ID Register (VER)

Offset

Register	Offset
VER	0h

Function

The VER register can be used to determine the version ID and feature set number of ELE_MUA.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	MAJOR								MINOR							
W																
Reset	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	FEATURE															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-24 MAJOR	Major Version Number (0x01)
23-16 MINOR	Minor Version Number (0x00)
15-0 FEATURE	Feature Set Number 0000_0000_0000_0000b - Standard features are implemented.

9.4.1.3 Parameter Register (PAR)

Offset

Register	Offset
PAR	4h

Function

The PAR register reports the number of Transmit (TR) registers and Receive (RR) registers.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	RR_NUM								TR_NUM							
W																
Reset	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0

Fields

Field	Function
31-16 —	Reserved
15-8 RR_NUM	Number of Receive (RRn) registers (8'd2)
7-0 TR_NUM	Number of Transmit (TRn) registers (8'd16)

9.4.1.4 Unused Register 0 (UNUSED0)

Offset

Register	Offset
UNUSED0	8h

Function

DO NOT WRITE

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0															Reserved
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-1 —	Reserved
0 —	DO NOT WRITE TO THIS BIT FIELD.

9.4.1.5 Status Register (SR)

Offset

Register	Offset
SR	Ch

Function

The SR register reports the status of the Transmit Empty Pending and Receive Full Pending flags.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0									RFP	TEP	0				
W																
Reset	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Fields

Field	Function
31-7 —	Reserved
6 RFP	Receive Full Pending Flag Indicates if any of the receive registers are ready to be read. When the RFP bit is set, then software should check the RSR[RFn] flags to determine which RRn register is ready to be read. 0b - No data is ready to be read. All RSR[RFn] bits are clear. 1b - Data is ready to be read. One or more RSR[RFn] bits are set.
5 TEP	Transmit Empty Pending • The TEP bit reads as "1" when any TSR[TEn] bit is set. • The TEP bit reads as "0" when all TSR[TEn] bits are clear. • When the TEP bit reads as "1", check the TSR[TEn] flags to determine which TRn register is ready to be written.
4-0 —	Reserved

9.4.1.6 Transmit Control Register (TCR)

Offset

Register	Offset
TCR	120h

Function
The TCR register is used to configure which Transmit Empty interrupts are generated when the corresponding TSR[TEn] bits are set.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	TEIEn															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-16 —	Reserved
15-0 TEIEn	Transmit Register n Empty Interrupt Enable When set, causes a Transmit Empty interrupt to be generated when the corresponding TSR[TEn] bit is set.

9.4.1.7 Transmit Status Register (TSR)

Offset

Register	Offset
TSR	124h

Function

The TSR register shows which TR registers are ready to be written.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	TEn															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-16 —	Reserved
15-0 TEn	Transmit Register n Empty • The TEn bit is used to indicate to the ELE_MUA processing element that the corresponding ELE_MUA TRn register is ready to be written.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	• The TEn bit is set after the corresponding ELE_MUB RRn register is read, indicating the ELE_MUA TRn register is empty. • The TEn bit is cleared after the ELE_MUA TRn register is written, indicating the ELE_MUA TRn register is full. • After the TEn bit is cleared, the Transmit Empty interrupt n (if the TCR[TEIEn] bit is set) is cleared on the ELE_MUA side.

9.4.1.8 Receive Status Register (RSR)

Offset

Register	Offset
RSR	12Ch

Function

The RSR register shows which RR registers are ready to be read.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0														RFn	
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-2 —	Reserved
1-0 RFn	Receive Register n Full • The RFn bit is used to indicate to the ELE_MUA processing element that the corresponding ELE_MUA RRn register is ready to be read.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	• The RFn bit is set after the corresponding ELE_MUB TRn register is written, indicating the ELE_MUA RRn register is full. • The RFn bit is cleared after the ELE_MUA RRn register is read, indicating the ELE_MUA RRn register is empty.

9.4.1.9 Unused Register 1 (UNUSED1)

Offset

Register	Offset
UNUSED1	1FCh

Function

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	DATA16															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-16 —	Reserved
15-0 DATA16	Unused 16-bit Register

9.4.1.10 Transmit Register (TR0 - TR15)

Offset

For n = 0 to 15:

Register	Offset
TRn	200h + (n × 4h)

Function

The TR registers contain Transmit Data.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W	TR_DATA															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0															
W	TR_DATA															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 TR_DATA	Transmit Data • Data written to the TRn register is reflected in the ELE_MUB Receive Register n (RRn). The TRn and RRn registers are not double-buffered; a write to the TRn register overrides the data readable at the RRn register. • A write to the transmit register clears the ELE_MUA TSR[TEn] bit on the transmitter side, and sets the ELE_MUB RSR[RFn] bit on the receiver side. • TRn register should be written only when the ELE_MUA TSR[TEn] bit is set to "1". • Reading the TRn register returns all zeros.

9.4.1.11 Receive Register (RR0 - RR1)

Offset

Register	Offset
RR0	280h
RR1	284h

Function

The RR registers contain Receive Data.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	RR_DATA															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	RR_DATA															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 RR_DATA	Receive Data • Reflects the data written to ELE_MUB Transmit Register (TRn). • Reading the RRn register clears the ELE_MUA RSR[RFn] bit on the receiver side, and sets the ELE_MUB TSR[TEn] bit on the transmitter side. • RRn register should be read only when the ELE_MUB RSR[RFn] bit is set to "1".

9.4.1.12 Semaphore Status Register (SEMA4_SR)

Offset

Register	Offset
SEMA4_SR	400h

Function

Reads to this register return Edgelock enclave status and a 16-bit version of semaphore ownership.

NOTE

Bit field names with prefix "SSS" refer to the Security SubSystem, Edgelock enclave (ELE).

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	SSS_ BSY	MISC_BSY						SSS_ LCK	0						SSS_ CIP1	SSS_ CIP2
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	OWNR16															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31 SSS_BSY	Security SubSystem (ELE) Busy This field returns whether or not Edgelock enclave is busy. 0b - Edgelock enclave is not busy 1b - Edgelock enclave CPU is busy
30-25 MISC_BSY	Miscellaneous ELE Busy Indicators This field is a multi-bit collection of individual internal ELE module busy status bits, where an all-zero value indicates no modules are busy and a non-zero value indicates one or more internal ELE modules are busy.
24 SSS_LCK	Security SubSystem (ELE) lockup This field returns whether or not the Edgelock enclave is locked up. 0b - Edgelock enclave is not locked up 1b - Edgelock enclave is locked up in an unrecoverable state
23-18 —	Reserved
17 SSS_CIP1	Security SubSystem (ELE) command group 1 in progress This field returns whether or not a service request is being processed. 0b - Service request group 1 not being processed by ELE 1b - Service request group 1 being processed by ELE
16 SSS_CIP2	Security SubSystem (ELE) command group 2 in progress This field returns whether or not a service request is being processed 0b - Service request group 2 not being processed by ELE 1b - Service request group 2 being processed by ELE

Table continues on the next page...

Table continued from the previous page...

Field	Function

15-0
OWNR16
	Semaphore Owner
	This field returns one of three values: 0x0000 if the semaphore is unlocked, 0x0055 if the semaphore is locked and owned by the requesting host task (based on domain identifier and the 2-bit {secure/nonsecure, privileged/user} access mode attributes) and 0xFFAA if the semaphore is locked but not by the requesting host task. These values support a 3-way signed conditional branch (> 0 (owner), = 0 (unlocked), < 0 (not owner)) based on the return value.
	`if (SEMA4 == IDLE)`
	`    then OWNR16 = 0x0000`
	`    else if (SEMA4 == requesting_host)`
	`        then OWNR16 = 0x0055`
	`        else OWNR16 = 0xFFAA`

9.4.1.13 Semaphore Ownership Register (SEMA4_OWNR)

Offset

Register	Offset
SEMA4_OWNR	474h

Function

Reads to this virtual register return the semaphore ownership state.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	OWNR32															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	OWNR32															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 OWNR32	Semaphore Owner This field returns one of three values: 0x0000_0000 if the semaphore is unlocked, 0x0000_0055 if the semaphore is locked and owned by the requesting host task (based on domain identifier and the 2-bit

Table continues on the next page...

Field	Function

	{secure/nonsecure, privileged/user} access mode attributes) and 0xFFFF_FFAA if the semaphore is locked but not by the requesting host task. These values support a 3-way signed conditional branch (> 0 (owner), = 0 (unlocked), < 0 (not owner)) based on the return value.
	`if (SEMA4 == IDLE)`
	`    then OWNR32 = 0x0000_0000`
	`    else if (SEMA4 == requesting_host)`
	`         then OWNR32 = 0x0000_0055`
	`         else OWNR32 = 0xFFFF_FFAA`

9.4.1.14 Semaphore Acquire Register (SEMA4_ACQ)

Offset

Register	Offset
SEMA4_ACQ	998h

Function

Reads to this virtual register attempt to acquire (lock) the semaphore for the requesting task.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	OWNR32															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	OWNR32															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 OWNR32	Semaphore Owner Reads of this virtual register perform a compound W/R operation. The first operation is an attempted lock of the semaphore. If unlocked, the semaphore state is changed to locked and the associated address attributes (domain identifier, secure/nonsecure and privileged/user attributes) are captured; else if already locked, then the attempted lock of the semaphore is ignored. The second operation is a secure read of the semaphore state returning one of two possible values: 0x0000_0055 if the semaphore is owned by the requesting host task (based on domain identifier and the 2-bit {secure/nonsecure, privileged/user} access mode attributes) and 0xFFFF_FFAA if the semaphore is locked but not by the requesting host task.

Table continues on the next page...

Field	Function

	if (SEMA4 == IDLE)
	    then {SEMA4 = LOCKED by requesting host; OWNR32 = 0x0000_0055}
	    else if (SEMA4 == requesting_host)
	         then OWNR32 = 0x0000_0055
	         else OWNR32 = 0xFFFF_FFAA

9.4.1.15 Semaphore Release Register (SEMA4_REL)

Offset

Register	Offset
SEMA4_REL	ACCh

Function

Reads to this virtual register attempt to release (unlock) the semaphore for the requesting task.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	OWNR32															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	OWNR32															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function

31-0
OWNR32
	Semaphore Owner
	Reads of this virtual register perform a compound W/R operation. The first operation is an attempted release (unlock) of the semaphore. If locked and the associated address attributes (domain identifier, secure/nonsecure and privileged/user attributes) of the transaction match the semaphore state, the state is changed to unlocked; else if the attributes do not match, or the semaphore is already unlocked, then the attempted unlock is ignored. The second operation is a secure read of the semaphore state returning one of two possible values: 0x0000_0000 if the semaphore is unlocked and 0xFFFF_FFAA is the semaphore is locked but not by the requesting host task.
	if (SEMA4 == LOCKED by requesting host)
	    then {SEMA4 = UNLOCKED; OWNR32 = 0x0000_0000}
	    else if (SEMA4 == LOCKED by a different host)
	         then OWNR32 = 0xFFFF_FFAA
	         else OWNR32 = 0x0000_0000

9.4.1.16 Semaphore Forced Release Register (SEMA4_FREL)

Offset

Register	Offset
SEMA4_FREL	BA4h

Function

Reads to this virtual register attempt to unconditionally release (unlock) the semaphore.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	OWNR32															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	OWNR32															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function

31-0
OWNR32
	Semaphore Owner
	Reads of this virtual register perform a compound W/R operation. The first operation is an attempted release (unlock) of the semaphore. If locked and the associated secure/nonsecure and privileged/user attributes of the transaction signals SecurePrivileged, the state is unconditionally forced to unlocked; else if the attributes define an access mode different from SecurePrivileged, or the semaphore is already unlocked, then the attempted forced unlock is ignored. The second operation is a secure read of the semaphore state returning one of two possible values: 0x0000_0000 if the semaphore is unlocked and 0xFFFF_FFAA if the host task requesting the forced release was not in the SecurePrivileged mode.
	`if ((SEMA4 == LOCKED) && (request == SecurePrivileged))`
	`    then {SEMA4 = UNLOCKED; OWNR32 = 0x0000_0000}`
	`    else if (SEMA4 == UNLOCKED)`
	`         then OWNR32 = 0x0000_0000`
	`         else OWNR32 = 0xFFFF_FFAA`

Chapter 10
Key Management

10.1 Overview

The ELE implements a key store, which can be used to store user keys for crypto operations executed by the ELE. The ELE key store supports the following key types:

- Symmetric keys
- Asymmetric private keys:
  - NIST P-192
  - NIST P-224
  - NIST P-256
  - NIST P-384
  - NIST P-521
  - Curve25519
  - Brainpool P192r1
  - Brainpool P224r1
  - Brainpool P256r1
  - Brainpool P320r1
  - Brainpool P384r1
  - Brainpool P512r1
  - Brainpool P192t1
  - Brainpool P224t1
  - Brainpool P256t1
  - Brainpool P320t1
  - Brainpool P384t1
  - Brainpool P512t1
  - Binary Field (Koblitz) Curve sect163k1 (K-163)
  - Binary Field (Koblitz) Curve sect1283k1 (K-163)

  The ELE key store allows to store either private or public part of the key or complete key pair.

The number of keys, which can be stored in the key store is not fixed but is limited by the following parameters:

- Total key store size: 15836 bytes
- Total key objects number: 128 objects

The real number of available key objects depends on which parameter first reaches its limit. The key store also supports memory defragmentation. After some keys are deleted from the key store, the key store memory can be defragmented and reused for new keys.

The ELE supports the following operations with keys:

- Set/get key/key pair
- Generate random key/key pair

- Import/export key blob
- Set/get key properties

All keys including their input/output buffers are stored in big endian format. See source code example below:

```
//KEY = 1f8e4973953f3fb0bd6b16662e9a3c17
uint8_t symKeyData[16] = {0x1f, 0x8e, 0x49, 0x73, 0x95, 0x3f, 0x3f, 0xb0, 0xbd, 0x6b, 0x16, 0x66,
0x2e, 0x9a, 0x3c, 0x17};
```

10.2 Key object properties

The key object properties define access rights for the key, restrict key usage for specific crypto operations or specific mode within crypto operation. The key object properties can be set during the KEY_OBJECT_ALLOCATE_HANDLE command or any time later using the KEY_OBJECT_SET_PROPERTIES command. Key object properties for specific key object can be obtained by KEY_OBJECT_GET_PROPERTIES command. Some key object properties as SEC, DID and TRUSTED_KEY are set internally by ELE during key generation and cannot be modified by an user. The SEC and DID properties are set during the KEY_OBJECT_INIT command and their values are inherited from bus master requesting the KEY_OBJECT_INIT command. For example if secure privilege master with DID=0 sends the KEY_OBJECT_INIT command into ELE, the created key object will have set key properties to SEC=11 and DID=0. The TRUSTED_KEY property indicates to the user that the key was securely generated inside ELE and never left the ELE. This property is set during KEY_STORE_GENERATE_KEY command if all conditions (plain read and write are disabled) are met.

Field	Description
31 LOCK	This field locks the key object. After the key object is locked, neither key data nor key properties can be modified. The locked key object (including key data) cannot be deleted except by using CLOSE_SESSION, KEY_STORE_FREE and MGMT_CLEAR_ALL_KEYS commands. NOTE This field is a sticky bit. After it is set, it remains set until the next RESET. 0b - Key object is not locked. 1b - Key object is locked.
30-29 SEC	This field indicates the security access rights for the key. The host must have the same or higher security level for any key usage or modification. NOTE This field is read only. The SEC field is set automatically during KEY_OBJECT_INIT operation and the value is derived from the host security level requesting the KEY_OBJECT_INIT command. If the user wants to create a key object with lower security access rights, a call to the MGMT_SET_HOST_ACCESS_PERMISSION command with proper lower level should be issued just before the KEY_OBJECT_INIT command. 00b - Non-secure user 01b - Non-secure privilege 10b - Secure user 11b - Secure privilege

Table continues on the next page...

Table continued from the previous page...

Field	Description
28 TRUSTED_KEY	This field indicates that the key was randomly generated by the ELE and never left the ELE. This field is set during KEY_STORE_GENERATE_KEY operation only if the target key object has already set the NO_PLAIN_READ and NO_PLAIN_WRITE fields. This field is read only. 0b - Key is not a trusted key (not generated inside ELE and/or key has left ELE) 1b - Key is a trusted key (generated inside ELE and has plaintext reads and writes disabled)
27	Reserved
26-25 DID	Domain ID, see the *TRDC* chapter in the Reference Manual for more details. NOTE This field is read only. The DID field is set automatically during KEY_OBJECT_INIT operation and the value is derived from the host domain ID requesting the KEY_OBJECT_INIT command.
24 DID_MATCH	This field forces DID match during key usage, which means that domain ID captured during key creation must match to the domain ID of the master using the key. NOTE This field is a sticky bit. After it is set, it remains set until the next RESET.
23-17	Reserved
16 NO_IMPORT/EXPORT	This field disables key import/export using all ELE key blob types except ELE die unique blob. ELE die unique blob is always enabled. NOTE This field is a sticky bit. After it is set, it remains set until the next RESET. 0b - The key blob import/export is allowed 1b - The key blob import/export is not allowed
15 NO_PLAIN_READ	This field disables key read in plaintext from a key store slot. NOTE This field is a sticky bit. After it is set, it remains set until the next RESET. 0b - Key can be read by host in plaintext from key store slot (symmetric key or private part of asymmetric key) 1b - Key cannot be read in plaintext from host side
14 NO_PLAIN_WRITE	This field disables the key write in plaintext into key store slot. NOTE This field is a sticky bit. After it is set, it remains set until the next RESET. 0b - Key can be written by host in plaintext into key store slot

Table continues on the next page...

Table continued from the previous page...

Field	Description
	1b - Key cannot be written in plaintext from host side
13 NO_VERIFY	This field disables verifying operation with the key. NOTE This field is a sticky bit. After it is set, it remains set until the next RESET. 0b - Key can be used for verifying operation 1b - key cannot be used for verifying operation
12 NO_SIGN	This field disables sign operation with the key. NOTE This field is a sticky bit. After it is set, it remains set until the next RESET. 0b - Key can be used for sign operation 1b - Key cannot be used for sign operation
11 NO_DECRYPT	This field disables decrypt operation with the key. NOTE This field is a sticky bit. After it is set, it remains set until the next RESET. 0b - Key can be used for decrypt operation 1b - Key cannot be used for decrypt operation
10 NO_ENCRYPT	This field disable encrypt operation. NOTE This field is a sticky bit. After it is set, it remains set until the next RESET. 0b - Key can be used for encrypt operation 1b - Key cannot be used for encrypt operation
9-0 CRYPTO_OPERATION	This field defines all cryptographic operations, where the key can be used. If bit[n] is set, then the associated cryptographic operation is allowed. NOTE This field is write once field. After any non-zero value is written, it cannot be modified until the next RESET. The reserved bits must be kept as 0 for future compatibility bit0: AES bit1: MAC bit2: AEAD bit3: Asymmetric sign/verify bit4: KDF

Table continues on the next page...

Table continued from the previous page...

Field	Description
	bit5: PKB (key can be sent over PKB only) bit6: Reserved bit7: Reserved bit8: Reserved bit9: Reserved

Table 182 shows the key properties used in each command. The following identifiers are used in this table:

- x: The command evaluates this key property.
- x^*: The command evaluates this key property, when DID bit is set.
- w^1: The command writes into this key property. The written value is determined by executed command.
- w^2: The command writes user-defined value into this key property.
- r: The command reads this key property.

Table 182. Command key properties usage

Command	Bit field																
	31	29-30	28	25-26	24	16	15	14	13	12	11	10	5	4	3	2	1	0
CIPHER_ONE_GO		x		x^*							x	x						x
AEAD_ONE_GO		x		x^*							x	x				x		
MAC_ONE_GO		x		x^*													x	
MAC_INIT		x		x^*														
MAC_UPDATE		x		x^*														
MAC_FINISH		x		x^*														
ASYMMETRIC_SIGN		x		x^*						x					x			
ASYMMETRIC_VERIFY		x		x^*					x						x			
DERIVE_KEY	x	x		x^*										x				
ASYMMETRIC_DH_DERIVE_KEY	x	x		x^*										x				
ASYMMETRIC_SPAKE2_DERIVE_KEY	x	x		x^*										x				
KEY_STORE_SET_KEY	x	x		x^*				x										
KEY_STORE_GET_KEY		x		x^*			x											
KEY_STORE_EXPORT_KEY		x		x^*		x												
KEY_STORE_IMPORT_KEY	x/w^1	x		x^*		x/w^1	w^1	w^1	w^1	w^1	w^1	w^1		w^1	w^1	w^1	w^1	w^1

Table continues on the next page...

Table 182. Command key properties usage (continued)

Command	Bit field																	
KEY_STORE_GENERATE_KEY	x	x	w^1	x^*			x	x										
KEY_STORE_OPEN_KEY													x					
KEY_STORE_ERASE_KEY	x	x		x^*														
KEY_OBJECT_INIT		w^1		w^1														
KEY_OBJECT_ALLOCATE_HANDLE	w^2	x		x^*	w^2	w^2	w^2	w^2	w^2	w^2	w^2	w^2		w^2	w^2	w^2	w^2	w^2
KEY_OBJECT_GET_HANDLE		x		x^*														
KEY_OBJECT_GET_PROPERTIES	r	x/r	r	x^*/r	r	r	r	r	r	r	r	r		r	r	r	r	r
KEY_OBJECT_SET_PROPERTIES	x/w^2	x		x^*	w^2	w^2	w^2	w^2	w^2	w^2	w^2	w^2		w^2	w^2	w^2	w^2	w^2
KEY_OBJECT_FREE	x	x		x^*														

NOTE

While xxx_CONTEXT_INIT functions can contain a key object as an input parameter, the key properties are not used and checked until an operation attempts to use the key. For example, SYMMETRIC_CONTEXT_INIT specifies a key, but the key properties are validated when a CIPHER_ONE_GO operation attempts to use the key.

10.3 Import/export keys

The ELE supports the following key blob types for import/export operation:

- Import/export using die unique key
- Bluetooth ELE key export
- ELE to ELE import/export

10.3.1 Import/export using die unique key

The ELE does not include any user non-volatile memory. If an application requires to store keys into non-volatile memory, the keys must be stored in the host non-volatile memory. To keep confidentiality of keys stored in the EdgeLock enclave key store, the application can export keys by the key blobs wrapped using the die unique key. The key can be exported by KEY_STORE_EXPORT_KEY command with blob type set to “ELE die unique blob”. This command returns blob, which contains encrypted key value, include key type and key properties. The size of the blob is key size + 24 bytes. Now this blob can be stored to any application non-volatile memory.

After power cycle, the stored keys can be imported back into ELE using KEY_STORE_IMPORT_KEY command. The imported blob is decrypted and key data include key type and key properties are written into selected key object. Now the key is ready for any allowed crypto operation. Because the blob is encrypted/decrypted by the die unique key, the exported key blobs cannot be reused on another device.

Table 183 and Table 184 show command sequence for secure key generation and export, and key blob import.

Table 183. Command sequence for key generation and export

Command	Description
KEY_OBJECT_INIT KEY_OBJECT_ALLOCATE_HANDLE	Creates key object and allocates key store space.
KEY_STORE_GENERATE_KEY	Generates new key value.
KEY_STORE_EXPORT_KEY	Exports key as key blob.

Table 184. Command sequence for key blob import

Command	Description
KEY_OBJECT_INIT KEY_OBJECT_ALLOCATE_HANDLE	Creates key object and allocates key store space.
KEY_IMPORT_EXPORT_KEY	Imports key blob.

10.3.2 Bluetooth ELE key export

The ELE supports Bluetooth LE protocol by calculation of Session Key (SK) and Identity Resolving Key (IRK). The SK can be set by host or generated randomly by ELE. The IRK generation is supported by DERIVE_KEY command. For more information see DERIVE_KEY command description and Bluetooth LE protocol specification. To keep confidentiality of SK and IRK these keys has to be securely transferred into NBU. For this purpose the NBU and ELE are connected via Private Key Bus (PKB). The PKB is not used to transfer SK and IRK keys from ELE to NBU but it is used to transfer wrapping keys for SK and IRK key blobs.

The wrapping keys (NBU_DKEY_SK, NBU_DKEY_IRK) are random keys generated using KEY_STORE_OPEN command. Once this command is executed with corresponding parameters, the wrapping keys are randomly generated and sent via PKB in to NBU. After wrapping keys are available, the SK and IRK keys can be exported using KEY_STORE_EXPORT_KEY command. Final SK and IRK blobs are sent into NBU and automatically decrypted by wrapping keys.

Repeated call of KEY_STORE_OPEN_KEY command generates new random NBU_DKEY_x keys. So all blobs created prior to this command call become invalid, and SK and IRK keys have to be exported again.

10.3.3 ELE to ELE import/export

The ELE to ELE import/export is identical to import/export using die unique key except for the key used to encrypt/decrypt the key blob. This import/export uses a special internal key, which is created using ASYMMETRIC_DH_DERIVE_KEY command and an algorithm configured to ELE to ELE blob key derivation. The exported key blob can be sent into a different device and imported into its ELE. For more details, refer to ELE to ELE enclave key exchange.

10.4 ELE to ELE enclave key exchange

Some applications require exchanging keys among different nodes/devices. The ELE supports a secure mechanism to exchange keys between two ELE subsystems without exposing plaintext data out of any ELE subsystem. This mechanism is supported by two ELE commands:

- ASYMMETRIC_DH_DERIVE_KEY command using ELE to ELE blob key derivation algorithm
- KEY_STORE_IMPORT_KEY/KEY_STORE_EXPORT_KEY with the key blob type parameter configured as ELE to ELE blob.

The ELE to ELE blob key derivation algorithm is the standard Diffie-Hellman key exchange algorithm using NIST P-256 elliptic curve. The resultant derived key is used as a salt data to the internal key derivation function. The output of this key derivation function is the key, which is used to encrypt/decrypt the ELE to ELE key blob. The complete key exchange between two ELE subsystems is shown in the following table:

ELE A	ELE B
Generate asymmetrical key pair for NIST P-256 curve	Generate asymmetrical key pair for NIST P-256 curve
Send public part of the key to ELE B	Send public part of the key to ELE A
Execute ELE to ELE key derivation algorithm	Execute ELE to ELE key derivation algorithm
Export selected key using ELE to ELE export command and send it to ELE B	Import received blob into ELE B via ELE to ELE import command
Now both ELE subsystems share the same key without exposing plaintext key data out of any ELE.	

Chapter 11
Debug Subsystem (DBGMB)

11.1 Chip-specific Debug Mailbox information

Table 185. Reference links to related information[1]

Topic	Related module	References
Full description	Debug Mailbox	Debug Mailbox (DBGMB)
System memory map		See the section "System memory map"
CLocking		See the chapter "Clocking"
Signal multiplexing	Port control	See the chapter "Signal Muxing and Pinout "

[1] For all the reference sections and chapters mentioned in this table, refer the Reference Manual.

11.1.1 Module instance

This device has one instance of the Debug Mailbox module.

11.2 Overview

This chapter outlines the debug capabilities implemented in this chip through DBGMB. Debugging uses the Arm Serial Wire Debug (SWD) interface. This chapter is for debug tool developers and assumes that you know the Arm SWD interface and CoreSight (TM) debug and trace technology.

DBGMB is accessible through two paths:

- By software running on the chip using the DBGMB register interface as the access method.
- By tools software running on a PC connected to a device over SWD pins. These tools access DBGMB using the Arm Debug Interface (ADIv5) specification where the individual nodes or ports are called Access Ports (AP). In this context, we are using the DBGMB AP (DM-AP).

11.2.1 Block diagram

This figure shows the top-level debug ports and connections.

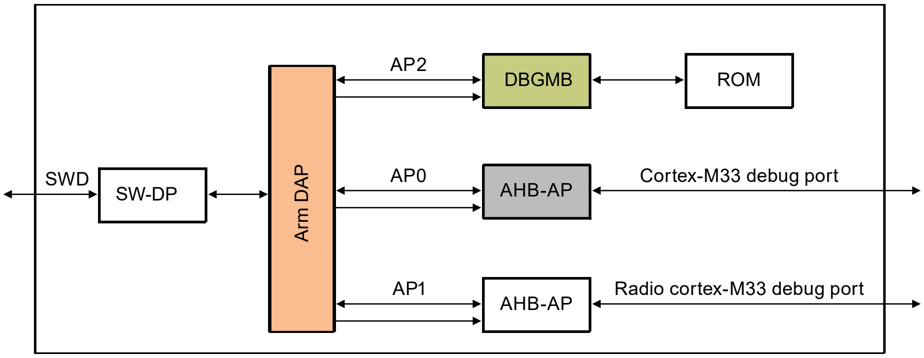


Figure 56. Serial Wire Debug (SWD) internal connections

11.2.2 Ports

Table 186. Ports

Port	Description
Arm DAP	Debug access port with a serial wire port (SW-DP) which interprets incoming data and routes it to the appropriate AP
AP0 (CPU0)	Debug access port for the Cortex-M33 core instantiated as CPU0
AP1 (Radio)	Debug access port for the Radio Cortex-M33 core
AP2 (DBGMB)	Debug access port for the debug mailbox. DBGMB: • Sends and receives messages from the code executing from the ROM. • Is always enabled. ROM can send and receive data externally. • Implements the NXP debug authentication protocol.

For JTAG identifier details, see the SMSCM chapter.

11.2.3 Features

- Support for Arm SWD mode
- Cortex-M33 CPU instruction trace capability via trace port, with output via a serial wire viewer
- Direct debug access to all memories, registers, and peripherals
- No target resource requirements for a debugging session
- Support for setting instruction breakpoints
- Support for setting data watchpoints, which you can use as triggers
- Optional additional software-controlled trace for the CPU via the instrumentation trace macrocell (ITM)

11.2.4 Glossary

Table 187. Glossary

Abbreviation	Term	Description
RoT	Root of Trust	Vendor-owned key pair that authorizes data assets through cryptographic signatures. The public part of the key is typically preconfigured in products so that data from untrusted sources can be cryptographically verified. The vendor public key used by the device to verify the signature of this DC. (The corresponding private key was used to sign the DC).
RoTpub	RoT Public Key	The vendor public key used by the device to verify the signature of this DC. (The corresponding private key was used to sign the DC.)
RoTID	RoT Identifier	RoTID allows the debugger to infer which RoT public keys are acceptable to the device. If the debugger does not provide such a credential, the authentication process fails.
RoTMETA	RoT metadata	The RoT metadata required by the device to corroborate: the ROTID sent in the DAC, the field in this DC, and any additional RoT state not stored within the device. This

Table continues on the next page...

Table 187. Glossary (continued)

Abbreviation	Term	Description
		metadata allows different RoT identification, management, and revocation solutions to be handled.
SoCC	SoC Class	A unique identifier for a set of SoCs that require no SoC-specific differentiation in their debug authentication. The main usage is to allow a different set of debug domains and options to be negotiated between the device configuration and credentials. If the granularity of debug control warrants it, a class can contain a single revision of a single SoC model.
DCK	Debug Credential Key	A user-owned key pair. The public part of the key is associated with a DC. The private part is held by the user and used to produce signatures during authentication.
DC	Debug Credential	A user public key and associated attributes, bound together and signed by a RoT, serves as an *identity*.
CC	Credential Constraint	In product configuration, CCs are limitations on the DCs that the device accepts for authentication. In DCs, CCs are vendor/RoT-authorized usages of the DC, as well as inputs to the desired debug behavior.
VU	Vendor Usage	A CC (constraint) value that is opaque to the debug authentication protocol itself but can be leveraged by vendors in product-specific ways.
SoCU	SoC Usage	A CC (constraint) value that is a bit mask, and whose bits are used in an SoC-specific manner. These bits are typically used for controlling which debug domains are accessed via the authentication protocol. Device-specific debug options can also be managed in this way.
CB AB	Credential Beacon Authentication beacon	A value that is passed through the authentication protocol, which is not interpreted by the protocol but is instead made accessible to the application being debugged. A credential beacon is associated with a DC and is therefore vendor/RoT-signed. An authentication beacon is provided and signed by the debugger during the authentication process.
DCFG_*	Debug Config	Refers to device configuration settings stored in OTP.
OTP	One-time programmable	One-time programmable storage (fuses).

11.3 Functional description

11.3.1 Basic configuration

This chip supports Arm's serial wire debug (SWD) interface. SWD is the default function for pins PTA1(SWCLK) and PTA0 (SWDIO) after a reset. If you use serial wire output (SWO), you must enable it in the application code by selecting the SWO function on PTA4 and enabling the trace clock (see External signals).

The ROM controls debug access via a remote host, and it is enabled only when permitted through the chip configuration and when you follow the correct protocol to initiate a debug session. If you have configured the chip for debug authentication, you must initiate a debug session following the correct authentication sequence. When a chip is in the development life-cycle state, use the debug session protocol described in Debug session protocol. When the chip is in deployed life-cycle state, use the debug authentication protocol described in Debug authentication.

11.3.2 Debug Access Port (DAP)

The DAP with a serial wire port (SWJ-DP) interprets incoming data and routes it to the appropriate AP. External signals describes the external I/O pins that interface with DAP. The DAP block is always enabled, but you can use the I/O pins that provide access to the SWD signals for other functions controlled by software.

11.3.3 Arm Cortex-M33 access port

The debug access port (DAP) for the Arm Cortex-M33 processor is disabled during power-on reset (POR) or during the assertion of the reset pin. The ROM enables the DAP when you follow the correct debug initiation procedures. If you are not using the DAP, you can use the debug enablement protocol to initiate a debug session.

The debug authentication process allows control of the DBGEN, NIDEN, SPIDEN, and SPNIDEN authentication signals connected to Cortex-M33 as described below.

Table 188. Authentication signals

Signal	Description
DBGEN	Invasive debugging • Breakpoints and watch points halt the processor on a specific activity. • A debug connection examines and modifies registers and memory, and it provides single-step execution.
NIDEN	Noninvasive debugging • Collects information on instruction execution and data transfers. • Delivers trace in real time to off-chip tools to merge data with source code on a development workstation for future analysis.

11.3.4 Debugger mailbox access port (DM-AP)

The DM-AP provides a register-based mailbox accessible by a chip's core and the device debug port (DP). This port is always enabled. An external host communicating through the SWD interface can exchange messages and data with the boot code executing from the ROM. This port implements the NXP debug authentication protocol. See Debug authentication for a description of the protocol used to initiate a debug session from a host debug system.

The boot ROM implements a debug mailbox protocol to interact with tools via the SWD interface.

The debug mailbox protocol has the following features:

- Request-and-response based, where the requests and responses use the same basic structure
- Support for relatively large command and response data
- 32-bit word alignment for all commands and responses
- Support for data above 32 bits via an ACK_TOKEN that moderates the transfer in 32-bit value chunks

11.3.5 Debug session protocol

When DBGMB fetches instructions from the ROM address range during boot, the DAP of CPU0 is disabled, regardless of device life-cycle state or DCFG_SOCU settings. This document refers to this mechanism as Boot-ROM protection. The method to initiate a debug session varies depending on the device state and intended debugging scenario. The scenarios described in the rest of these sections are:

Table 189. Debug session scenarios

Scenario	Description
Debug session with uninitialized or invalid image or ISP mode	If DBGMB detects no valid image, the ROM proceeds into In-System Programming (ISP) mode and waits to be booted via one of its serial interfaces. In ISP mode, the debug interface is disabled.
Debug session with valid application	Program control is in ISP mode, initiated because the ISP pin is asserted on the device at reset.
Debug session attaching to a running target	Connecting to a device with the intent to debug without writing an image (also called a debug attach).
Halting execution immediately following ROM execution	Connecting to a device running a valid application, with the intent to write a new application.

11.3.5.1 Debug session with uninitialized or invalid image or ISP mode

When the chip boots and the boot media does not contain a valid image, the ROM-based program control enters ISP mode. DBGMB disables debug access for security reasons. The chip also enters into ISP mode when the ISP has been asserted as the chip leaves reset. This section describes how to establish a debug session for these scenarios.

To ensure that the state machine controlling debug mailbox commands is in a known state, the debugger can reset this logic via the following steps:

1. Write 1 to CSW[RESYNCH_REQ] to request resynchronization.
2. Write 1 to CSW[CHIP_RESET_REQ] to reset the chip.
3. Read Command and Status Word (CSW).
4. Wait until the chip has completed resynchronization, identified by reading a value of 0 from the lower 16 bits of CSW.

To start a debug session and control the exchange of debug information, the debugger uses the DM-AP commands:

1. Following a successful initial resynchronization, communicate with the chip via 32-bit DM-AP commands to the REQUEST register in the debug mailbox.
2. Read results via the RETURN register. To ensure that the transactions have completed successfully, the debugger must poll the RETURN register as it polled the CSW register following a resynchronization request.
3. View the results in Response packet.

To initiate a debug session over SWD while debug is disabled:

1. The debug system must issue a Start Debug Session command to the ROM-based boot code, following the Debug session protocol.
2. Upon receiving the command, the boot code disables any unwanted peripheral and manages NXP secrets before enabling debug access.
3. After enabling debug access, the ROM enters a while(1) loop.

Once the Start Debug Session command and chip reset have executed successfully, the AP for CPU0 is accessible. You can use it to set breakpoints and perform other tasks, as with other Arm Cortex-M devices.

Following a successful debug connection, a flashloader is loaded into RAM to program the application to be debugged and to set required breakpoints in the code. After this process, a SYSRESET_REQ command must be issued to verify that the ROM fully executes (similar to a deployed end application) before reaching the downloaded application.

The code sample below shows how to initiate a debug session for the scenarios described above:

```
// Pseudo Code Syntax
// -----------------------------------------------------------------------
// WriteDP "register" "value"
// value = ReadDP "register"
// AP transactions presume the DM AP is selected
// WriteAP "register" "value"
// value = ReadAP "register" "value"
// ----------------------------------------------------------------------
```

```
// Read AP ID register to identify DM AP at index 2
WriteDP 2 0x020000F0
// The returned AP ID should be 0x002A0000
value = ReadAP 3
print "AP ID: ", value
```

```
// Select DM AP index 2
WriteDP 2 0x02000000
// Write DM RESYNC_REQ + CHIP_RESET_REQ
WriteAP 0 0x21
// Poll CSW register (0) for zero return, indicating success
value = -1
while value != 0 {
value = ReadAP 0
}
print "RESYNC_REQ + CHIP_RESET_REQ: ", value
// Write DM START_DBG_SESSION to REQUEST register (1)
WriteAP 1 7
// Poll RETURN register (2) for zero return
value = -1
while value != 0 {
value = (ReadAP 2 & 0xFFFF)
}
print "DEBUG_SESSION_REQ: ", value
```

11.3.5.2 Debug session with valid application

In this case, the application has not disabled debug. You can access the CPU0 AP and set breakpoints without resynchronizing the mailbox hardware or issuing a Debug Session Request. You can use the methods described in Debug session with uninitialized or invalid image or ISP mode to simplify debug support implementations.

11.3.5.3 Debug session attaching to a running target

In this scenario, the device has booted and is running an application that has not disabled debug. The host system is attempting to connect to the device without resetting it and without updating the application. In this case, the CPU0 AP should be accessible and you can set breakpoints without resynchronizing the mailbox hardware.

11.3.5.4 Halting execution immediately following ROM execution

Traditionally, debug systems can set a vector catch at the reset vector to break code execution, but the chip ROM prevents this method.

The debugger must use this reset sequence:

Table 190. Reset procedure

Step	Purpose	Programming	Notes
1	Allow the debug system to halt execution immediately after the ROM	Set the data watchpoint on a read to address location E000ED08h.	—

Table continues on the next page...

Table 190. Reset procedure (continued)

Step	Purpose	Programming	Notes
	completes the preparations associated with a debug session request.		
2	—	If all data watchpoint comparators are occupied, backup one of the watchpoint settings and replace it with the above watchpoint location.	—
3	Reset the core and peripherals.	Write 1 to the Arm Cortex-M33 field AIRCR[SYSRESETREQ].	—
4	Allow the ROM to reenable debug access.	Wait for 100 ms.	—
5	Verify that the core has halted at the watchpoint.	Read the Arm Cortex-M33 register DHCSR.	If the DHCSR read times out or returns an error response, the ROM has entered an ISP command-handling loop due to an invalid image in boot media. Proceed to Step 6, otherwise, proceed to Step 10.
6	—	Execute the start debug session sequence described in Debug session with uninitialized or invalid image or ISP mode.	—
7	—	Wait for 10 ms.	—
8	—	Enable the Cortex-M33 AP.	—
9	—	Read DHCSR and issue a HALT command to Cortex-M33 AP.	—
10	—	Clear data watchpoint added in step 1. If you set the watchpoint as described in step 2, restore the watchpoint configuration.	—

11.3.6 Reset handling

The debug domain (DP, Cortex-M33 AP, DM-AP) is reset upon POR or pin reset (assertion of external nRESET). On other resets, the debug domain retains its state. The defined breakpoints and watchpoints persist even when the debugging tool issues a reset to the device.

11.3.7 Mailbox commands

This section describes the request and response message formats and available mailbox commands.

11.3.7.1 Request packet layout

The first word transmitted in a request is a header word containing the command ID and the number of following data words. The command packet is sent to the device by writing 32 bits at a time to the REQUEST register. When sending command packets greater than 32 bits, the debugger must read an ACK_TOKEN in the RETURN register before writing the next 32 bits.

The 32-bit words quantified by the header follow the header itself.

Table 191. Request register byte description

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	commandID[7:0]	commandID[15:8]	dataWordCount[7:0]	dataWordCount[15:8]
1	*data*	—	—	—

The C structure definition for a request is:

```
struct dm_request {
        uint16_t commandID;
        uint16_t dataWordCount;
        uint32_t data[ ];
        };
```

11.3.7.1.1 DM-AP commands

DM-AP commands are written to the REQUEST register. An Exit DM_AP command follows one or more of the DM-AP commands in the table below to resume the normal device boot flow. This sequence does not occur after the Enter ISP Mode and Start Debug Session commands.

Table 192. DM-AP commands

Name	Command code	Parameter and response	Description
Start DM-AP (legacy command)	01h	Parameters: None Response: 32-bit status	Causes the device to enter DM-AP command mode. If used, this command must occur before sending other commands. This command is provided for backward compatibility and is not required.
Mass Erase	03h	Parameters: None Response: 32-bit status	Erase the entire CM33 Program Flash (IFR0 not included). You can restrict support of this command through the DCFG_SOCU field as described in Debug authentication. See also Credential Constraints (DCFG_CC_SOCU).
Exit DM_AP (legacy command)	04h	Parameters: None Response: 32-bit status	Causes the device to exit DM-AP command mode. The device returns to normal boot path.
Enter ISP Mode	05h	Parameters: • dataWordCount: 1h • data[0]: ISP mode enum • — FFFF_FFFFh - Auto detection — 1h - LPUART — 2h - LPI2C	Enters the specified ISP mode. By default, the state of the ISP boot selection pins at the time of reset determines the ISP mode entry. The protected flash region (called either IFR or PFR) configuration usually disables this functionality prior to field deployment. You can use this command to enter ISP mode in those situations or when a different board function uses the ISP boot selection pins. You can restrict support of this command through the DCFG_SOCU field as described

Table continues on the next page...

Table 192. DM-AP commands (continued)

Name	Command code	Parameter and response	Description
		— 4h - LPSPI — 8h - CAN — Others - Reserved Response: 32-bit status	in Debug authentication. See also Credential Constraints (DCFG_CC_SOCU).
Set FA Mode	06h	Parameters: data[]: FA request Response: 32-bit status	Sets the part permanently in Fault Analysis (FA or RMA) mode bit in PFR field, and is ready to return the part to NXP for FA/RMA testing. Upon receiving this command boot code in ROM, customer-sensitive assets (key codes), stored in the protected flash region (called either IFR or PFR), are erased. Also, the FA or RMA mode bit in the protected flash region becomes 1, so suspect parts can be sent to NXP for FA/RMA testing. You can restrict support of this command through the DCFG_SOCU field as described in Debug authentication. See also Credential Constraints (DCFG_CC_SOCU).
Start DBG Session	07h	Parameters: None Response: 32-bit status	Starts debug session. Used to indicate to ROM the intention of connecting the debugger. ROM bootloader enables debug access (if no debug authentication required) and enters while(1) loop.
Debug Auth. Start	10h	Parameters: dataWordCount: 0h Response: data[]: DAC	Start the Debug Authentication Protocol. ROM responds to the debugger with a debug authentication challenge (DAC) message.
Debug Auth. Response	11h	Parameters: data[]: DAR Response: 32-bit status	Debug authentication response

11.3.7.2 Response packet layout

The first word transmitted in a response is a header word containing the command status and the number of following data words. The command response can be read 32 bits at a time through Return Value (RETURN). The initial 32 bits contains the response header, as shown in the table below. When reading a response longer than 32 bits, the debugger writes the ACK_TOKEN to REQUEST after every read until the full response packet is received.

NOTE

To support legacy LPC command and response values, Bit_31 in the header indicates that the response follows the new protocol structure (see Table 194). This bit is 1 when using the new protocol.

Table 193. Response register byte description

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	bits[7:0]:commandStatus[7:0]	bits[7:0]:commandStatus[15:8]	bits[7:0]:dataWordCount[7:0]	bits[6:0]:dataWordCount[14:8] bits[7]:new_protocol[15]
1	*data*	—	—	—

The C structure definition for a response is:

```
struct dm_response {
        uint16_t commandStatus;
        uint16_t dataWordCount;
        uint32_t data[];
        };
```

11.3.7.2.1 Response packet

You can read the command response 32 bits at a time through Return Value (RETURN). The initial 32 bits contain the response header, as shown in the table below. When reading a response greater than 32 bits, the debugger writes the ACK_TOKEN to Request Value (REQUEST) after every read of RETURN until the full response packet is received.

Table 194. Response Packet

Bits	Field	Description
31	LONG_RESP	Long response packet indicator; also called the new protocol indicator
[30:16]	REMAIN_TRANS	Number of times the debugger must read RETURN to receive remaining response data. The debugger writes ACK_TOKEN to REQUEST after every read of RETURN until the full response packet is received. NOTE REMAIN_TRANS is valid only when ERROR_SRESP is 0 and LONG_RESP is 1.
[15:0]	ERROR_SRESP	Short response data or error code If bit_20 is not 1, this field is interpreted as short response data. For example, this field returns the CRP level for a GET_CRP_LEVEL command. Error codes • 0000h: Command succeeded. • 0001h: Debug mode not entered. This value is returned when other commands are sent prior to the Enter DM-AP command. • 0002h: Command is not supported. • 0003h: Communication failure. ACK_TOKEN is missing during data transactions.

11.3.7.3 ACK_TOKEN

The ACK_TOKEN provides an acknowledgment to the sender during debug mailbox data transactions. DBGMB uses the acknowledgment in the following ways:

- When the debugger issues a command with parameter data, it waits for the ACK_TOKEN (read through Return Value (RETURN)) before writing the next 32-bit value to Request Value (REQUEST).
- When the debugger receives a long response packet, it writes the ACK_TOKEN to REQUEST before reading the next 32-bit value RETURN.

Table 195. ACK_TOKEN

Bits	Field	Description
[31:16]	REMAIN_TRANS	Number of remaining data transactions
[15:0]	ACK_MARKER	Acknowledgment marker; must always be A5A5h

The C structure definition for the ACK_TOKEN is:

```
struct dm_ack_token {
        uint16_t token; /* always set to A5A5h */
        uint16_t remainCount; /* count of remaining word */
        };
```

11.3.7.4 Error handling

When an overrun occurs from either side of the communication, DBGMB sets the appropriate error flag in Command and Status Word (CSW). The state machine hardware prevents further communication in either direction. The debugger must start with a new resynchronization request to clear the error flag.

11.3.8 Debug authentication

The fundamental principles of debugging, which require access to the system state and system information, conflict with the principles of security, which require the restriction of access to assets. Therefore, many products disable debug access completely before deploying the product. This causes challenges for product design teams to do proper Return Material Analysis (RMA). To address these challenges, the chip offers a debug authentication protocol as a mechanism to authenticate the debugger (an external entity) has the credentials approved by the product manufacturer before granting debug access to the device.

The debug authentication is a challenge-response scheme and assures that only the debugger in possession of the required debug credentials can successfully authenticate over the debug interface and access-restricted parts of the device. This protocol is divided into steps as described below:

1. The debugger initiates the Debug Mailbox message exchange by setting the CSW[RESYNCH_REQ] bit and CSW[CHIP_RESET_REQ] bit of DM-AP.
2. The debugger waits (minimum 30 ms) for the devices to restart and enter debug mailbox request handling loop.
3. The debugger sends the Debug Authentication Start command (command code 10h) to the device.
4. The device responds back with Debug Authentication Challenge (DAC) packet based on the , which are collectively referred as Device Credential Constraints Configuration (DCFG_CC). The response packet also contains a 32 bytes random challenge vector.
5. The debugger responds to the challenge with a Debug Authentication Response (DAR) message by using an appropriate debug certificate, matching the device identifier in the DAC. The DAR packet contains the debug access permission certificate, also referred as Debug Credential (DC), and a cryptographic signature binding the DC and the challenge vector provided in the DAC.
6. The device on receiving the DAR, validates the contents by verifying the cryptographic signature of the message using the debugger's public key present in the embedded the Debug Credential (DC). On successful validation of DAR, the device enables access to the debug domains permitted in the DC.

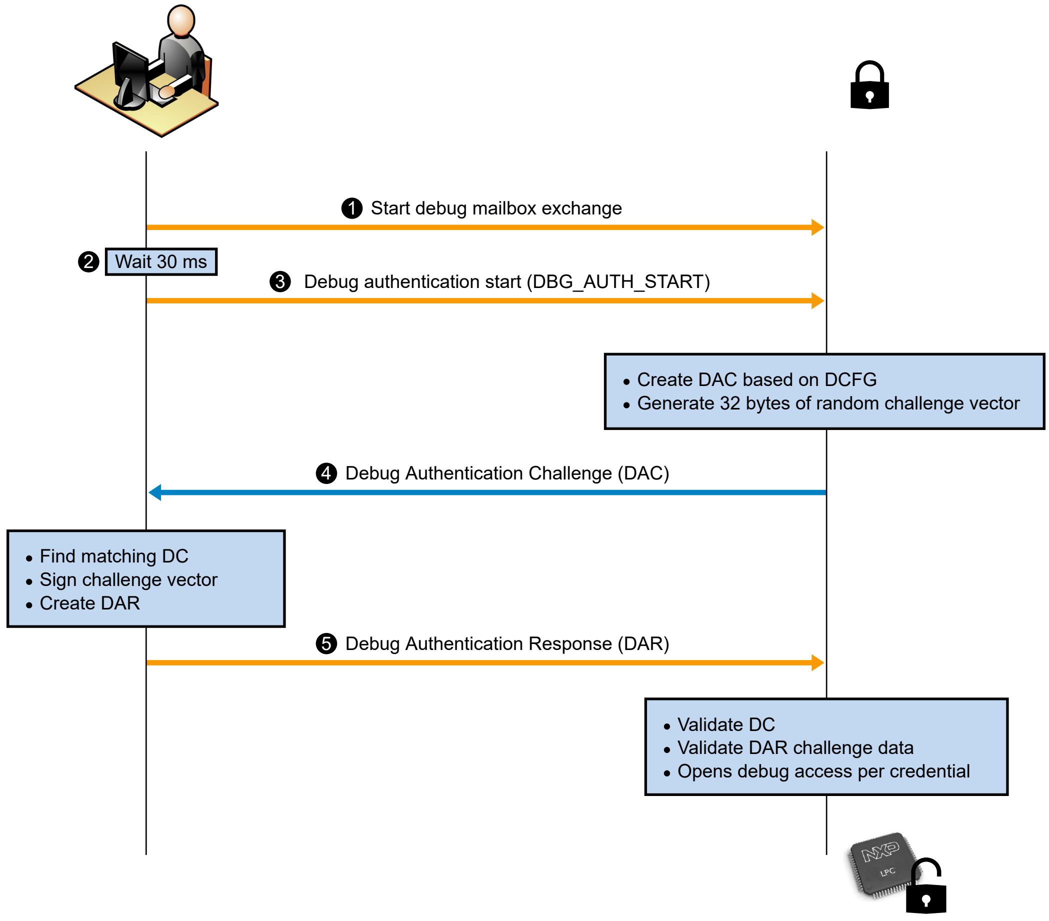

1 Start debug mailbox exchange
2 Wait 30 ms
3 Debug authentication start (DBG_AUTH_START)
• Create DAC based on DCFG
• Generate 32 bytes of random challenge vector
4 Debug Authentication Challenge (DAC)
• Find matching DC
• Sign challenge vector
• Create DAR
5 Debug Authentication Response (DAR)
• Validate DC
• Validate DAR challenge data
• Opens debug access per credential
NXP
LPC

Figure 57. Debug authentication flow

11.3.8.1 Debug Access Control Configuration

The boot code that is present in the device ROM handles the device side of the Debug authentication process. However, the debug access control rights and security policies can be configured by programming the following configuration fields, which are collectively referred to as Device Configuration for Credential Constraints (DCFG_CC) .

- **DCFG_VER:** This field controls the cryptographic primitives used during authentication.
- **DCFG_ROTID:** This field defines the Root of trust identifier (ROTID). The ROTID field is used to bind the devices to specific Certificate Authority (CA) keys issuing the debug credentials. These CA keys are also referred as Root of Trust (RoTK) keys.
- **DCFG_UUID:** Controls whether to enforce UUID check during Debug Credential (DC) validation. If this field is set only DC with matching device UUID can unlock the debug access.
- **DCFG_CC_SOCU:** This configuration field specifies access rights to various debug domains.
- : This field can be used to define vendor specific debug policy use case such as DC revocations or department identifier. It is recommended to use the field for revocation of already issued debug certificates.

These fields should be programmed as part of the OEM provisioning process.

11.3.8.1.1 Protocol Version (DCFG_VER)

The device supports multiple instantiations of Debug authentication protocol versions, which are defined based on cryptographic algorithms and key sizes.

- Version 2.0: Uses ECDSA P-256 signature verification using ECC keys.
- Version 2.1: Uses ECDSA P-384 signature verification using ECC keys.

Note, both debug authentication certificates and image signing certificates use the same Root of Trust keys (RoTK).

By default it is expected that Root of Trust keys (RoTK) are based on P-384, if RoTK are based on the P-256 curve, then the DCFG_SOCU_L1 fuse bits [20:19] must be set to value 10b, which indicates that RoTK is based on P-256. All other values indicate P-384.

11.3.8.1.2 Root of Trust Identifier (DCFG_ROTID)

The Root of Trust Identifier used in debug authentication protocol is composed of two elements.

1. A 256-bit cryptographic hash (SHA256) for version 2.0 or 384-bit cryptographic hash (SHA384) for version 2.1 over the Root of Trust Keys Table. This is the same as the Root Keys Table Hash (RKTH) or Root of Trust Keys Hash (ROTKH) field in the Secure boot ROM chapter. RKTH (stored in the CUST_PROD_OEMFW_AUTH_PUK fuse) is a 32-byte (SHA-384 uses only 1st 32 bytes) SHA-256/SHA-384 of SHA-256/SHA-384 hashes of up to four root public keys.
2. The CUST_PROD_OEMFW_AUTH_P UK_REVOKE fuse contains 4 bits, each associated with one of the four RoT keys in the table. These bits are used for revoking the keys.

11.3.8.1.3 Enforce UUID checking (DCFG_UUID)

Controls whether to enforce UUID check during Debug Credential (DC) validation. If this field is set, then only DC containing a UUID attribute that is an exact match to the device can unlock the debug access.

This field can be set by programming UUID_CHECK (bit 18) in DCFG_CC_SOCU_L1 fuse or same bit position 18 in DCFG_CC_SOCU_L2 fuse pages, see Table 198.

This device-specific constraint, if enabled, is in addition to all the other constraints defined and enforced by the authentication protocol.

11.3.8.1.4 Credential Constraints (DCFG_CC_SOCU)

The DCFG_CC_SOCU is a configuration that specifies debug access restrictions per debug domain. These access restrictions are also referred as constraint attributes in this section. The debug subsystem on this device is sub-divided into multiple debug domains to allow finer access control. Table 196 shows the debug domains and their corresponding control bit position in the CC_SOCU field used in the Debug certificate (that is, the Debug Credential (DC)) and the OTP fuse location (DCFG_SOCU_L1 fuse and DCFG_SOCU_L2 fuse). The DCFG_CC_SOCU is logically composed of two components:

- **SOCU_PIN:** A bitmask that specifies which debug domains are predetermined by device configuration.
- **SOCU_DFLT:** Provides the final access level for those bits that the SOCU_PIN field indicated are predetermined by device configuration.

The following table shows the restriction levels:

Table 196. Access restriction levels

Restriction Level	SOCU_PIN [n]	SOCU_DFLT [n]	Description
0	1	1	Access to the subdomain is always enabled. This setting is provided for the module use case scenario where DCFG_CC_SOCU_L2 would be used to define further access restrictions before final deployment of the product.

Table continues on the next page...

Table 196. Access restriction levels (continued)

Restriction Level	SOCU_PIN [n]	SOCU_DFLT [n]	Description
1	0	0	Access to the subdomain is disabled at startup. But the access can be enabled through the debug authentication process by providing an appropriate Debug Credential (DC) certificate.
-	0	1	Illegal setting. Part may lock-up if this setting is selected.
2	1	0	Access to the subdomain is permanently disabled and cannot be reversed. This setting offers the highest level of restriction.

Debug domains supported by the device are shown below:

Table 197. CC_LIST_Table

Bit	Symbol	Description
0	NIDEN	Controls non-Invasive debugging of TrustZone for Arm8-M defined non-secure domain of CPU0 (AP0).
1	DBGEN	Controls invasive debugging of TrustZone for Arm8-M defined non-secure domain of CPU0 (AP0).
2	SPNIDEN	Controls non-Invasive debugging of TrustZone for Arm8-M defined secure domain of CPU0 (AP0).
3	SPIDEN	Controls invasive debugging of TrustZone for Arm8-M defined secure domain of CPU0 (AP0).
4	NBU_DBG_EN	Controls debugging of NBU (radio) core.
5	DSP_DBG_EN	Controls debugging of DSP core.
6	ISP_CMD_EN	Controls whether ISP boot flow DM-AP command (command code: 05h) can be issued after authentication.
7	FA_CMD_EN	Controls whether DM-AP Set FA Mode command (command code: 06h) can be issued after authentication.
8	ME_CMD_EN	Controls whether DM-AP Mass Erase command (command code: 03h) can be issued after authentication.
17:9	Reserved	Reserved
18	UUID_CHECK	Controls whether to enforce UUID check during Debug Credential (DC) validation. If this bit is set, then the device will only accept a Debug Credential (DC) certificate containing a UUID attribute that is an exact match to the device's UUID.
20:19	FORCE_P256	Valid only for DCFG_SOCU_L1. DCFG_SOCU_L1 fuse bits [20:19] must be set to value 10b, when RoTK is based on P-256. All other values (00b, 01b, 11b) indicate P-384. See Protocol Version (DCFG_VER).
31:21	Reserved	Reserved

Table 198. Layout of CC_SOCU_PIN (DCFG_SOCU_L1) and CC_SOCU_PIN_NS (DCFG_SOCU_L2)

Bits	Symbol	Description
8:0	SOCU_PIN[n]	Defines whether the restriction level for the subdomains is fixed or controlled by the debug authentication process. The bit encoding of this field is defined as per the CC_LIST_Table.
17:9	Reserved	Reserved for SOCU_DFLT.

Table 199. Layout of CC_SOCU_DFLT (DCFG_SOCU_L1) and DCFG_SOCU_DFLT_NS (DCFG_SOCU_L2)

Bits	Symbol	Description
8:0	Reserved	Reserved for SOCU_PIN.
17:9	SOCU_DFLT[n]	Defines the restriction level for the subdomains that are configured as predetermined in the SOCU_PIN[n] field. The bit encoding of this field is defined as per the CC_LIST_Table.

11.3.8.1.5 DCFG_CC_SOCU authentication and life cycle dependency

Following table shows the behavior of debug domains based on current life cycle state and debug authentication status. Domain can be either enabled, disabled, or DCFG_SOCU & DCFG_SOCU_NS configuration determines the settings for given life cycle and debug authentication status.

Table 200. Life cycle status for NIDEN/DBGEN and SPNIDEN/SPIDEN

Life cycle	NIDEN/DBGEN		SPNIDEN/SPIDEN	
	not authenticated	authenticated	not authenticated	authenticated
OEM-Open	enabled	enabled	enabled	enabled
OEM-SWC	enabled	enabled	disabled	DCFG_SOCU & DCFG_CC_SOCU_L1
OEM-Closed	disabled	DCFG_SOCU & DCFG_CC_SOCU_L2 & DCFG_CC_SOCU_L1	disabled	DCFG_SOCU & DCFG_CC_SOCU_L2 & DCFG_CC_SOCU_L1
OEM-Locked	disabled	disabled	disabled	disabled
Field Return OEM	enabled	enabled	enabled	enabled
Failure Analysis (FA)	enabled	enabled	enabled	enabled
Bricked	disabled	disabled	disabled	disabled

Table 201. Life cycle status for ISP_CMD_EN

Life cycle	ISP_CMD_EN	
	not authenticated	authenticated
OEM-Open	enabled	enabled
OEM-SWC	disabled	DCFG_SOCU & DCFG_CC_SOCU_L1
OEM-Closed	disabled	DCFG_SOCU & DCFG_CC_SOCU_L2 & DCFG_CC_SOCU_L1
OEM-Locked	disabled	disabled
Field Return OEM	disabled	disabled
Failure Analysis (FA)	disabled	disabled
Bricked	disabled	disabled

Table 202. Life cycle status for FA_CMD_EN and ME_CMD_EN

Life cycle	FA_CMD_EN		ME_CMD_EN	
	not authenticated	authenticated	not authenticated	authenticated
OEM-Open	enabled	enabled	enabled	enabled
OEM-SWC	disabled	DCFG_SOCU & DCFG_CC_SOCU_L1	disabled	DCFG_SOCU & DCFG_CC_SOCU_L1
OEM-Closed	disabled	DCFG_SOCU & DCFG_CC_SOCU_L2 & DCFG_CC_SOCU_L1	disabled	DCFG_SOCU & DCFG_CC_SOCU_L2 & DCFG_CC_SOCU_L1
OEM-Locked	disabled	disabled	disabled	disabled
Field Return OEM	disabled	disabled	disabled	disabled
Failure Analysis (FA)	disabled	disabled	disabled	disabled
Bricked	disabled	disabled	disabled	disabled

11.3.8.1.6 DCFG_CC_VU

This field can be used to define a vendor-specific debug policy use case such as Debug Credential (DC) certificate revocations or department identifier or model identifier. It is recommended to use the field for revocation of already issued debug certificates. During Debug Authentication Response (DAR) processing the device checks that the value specified in the Vendor Usage field of DC matches exactly the value programmed in DBG_AUTH_VU fuse of device configurations.

The device provides a 2-byte fuse DBG_AUTH_VU field.

- Recommended using this field to identify the department or model number. So that Debug Credential (DC) Certificates can be issued on a class basis instead of issuing UUID-specific certificates.

11.3.9 Debug Credential Certificate (DC)

By prior construction, the debugger should already have a Debug Credential Key (DCK). The public key part of this key pair represents the identity of the debugger through the creation of a DC. It binds that public key to usage attributes, and is then authorized or signed by the vendor's RoT key.

Combining keypair types is not allowed.

If ROTK is based on secp384r1, then DCK must also be based on secp384r1.

If ROTK is based on secp256r1, then DCK must also be based on secp256r1.

The total DC size depends on the protocol version and number of used root keys.

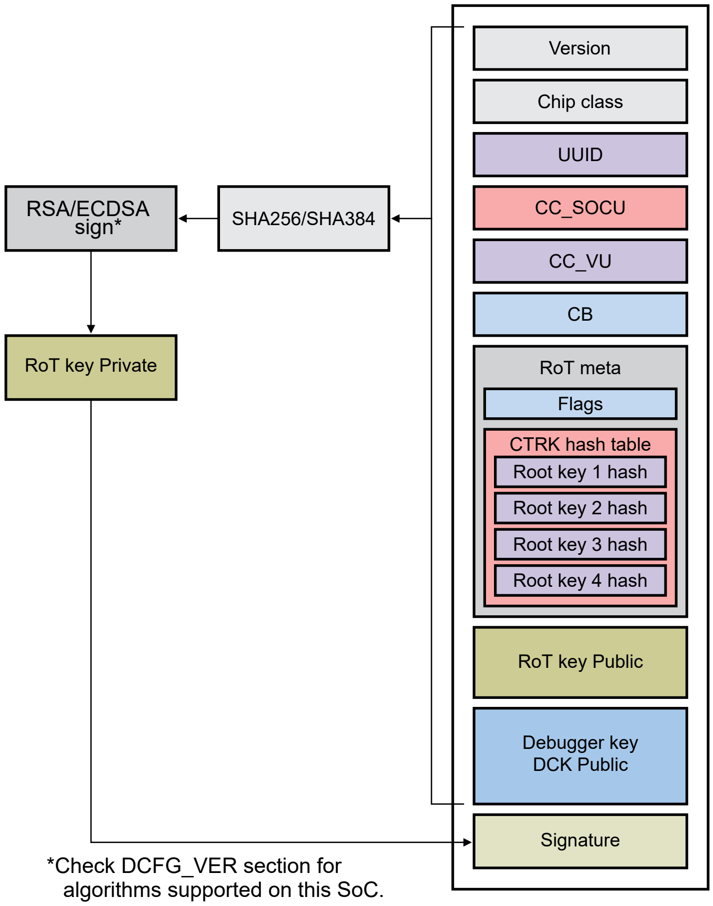


Figure 58. Debug Credential certificate fields

The data structure is represented as a packed binary concatenation of its component fields as shown in the list below.

Table 203. Debug Credential Certificate fields

Name	Offset	Description	Size in bytes
VERSION	0000h	Identifies the Debug Authentication protocol version	4

Table continues on the next page...

Table 203. Debug Credential Certificate fields (continued)

Name	Offset	Description	Size in bytes
		Version determined based on DCFG_SOCU_L1[20:19] fuse. • 00020000h for v2.0, which uses ECDSA P-256 (DCFG_SOCU_L1[20:19] - 10b) • 00020001h for v2.1, which uses ECDSA P-384 (DCFG_SOCU_L1[20:19] - 00b, 01b, 11b)	
SOCC	0004h	SoC class specifier Always set this field to 9h.	4
UUID	0008h	Unique device identifier, if this certificate is used on a device configured for UUID-matching. If UUID matching is enabled, the certificate is restricted to a specific device, otherwise the certificate is enabled for the whole SoC Class.	16
CC_SOCU	0018h	SoC specific Credential Constraints Specifies the debug access rights allowed for this certificate holder.	4
CC_VU	001Ch	Vendor Usage Should match the field in Device Configuration for Credential Constraints (DCFG_CC). See DCFG_CC_VU. It can be used to revoke Debug Certificates.	4
CB	0020h	Credential beacon that the vendor has associated with this DC. Only the lower 16 bits of this field are effective. This field can extend the Debug Authentication process. When a nonzero value is used in this field, ROM defers opening debug access to the user application. The result of the authentication process is written to the DBG_FEATURES register. The user application, after performing extended processing such as cleaning up critical keys and secrets, should copy the value to DBG_FEATURES_DP register to enable debug access. To aid the user application, ROM stores the beacon values in the DEBUG_AUTH_BEACON register.	4
ROTMETA_FLAGS	0024h	See Table 204	4
ROTMETA_CTRK_HASH_TABLE	0028h	SHA-256 or SHA-384 of root public key calculated based on root certificate EC size. Between two and four root certificates can be specified. In the case where only one root certificate is used, ctrkHashTable is not present.	variable (0 - 192)
ROT_KEY_PUB	variable	X and Y coordinates of Root of Trust vendor public key used for signing this certificate.	64 / 96
DCK_KEY_PUB	variable	X and Y coordinates of Debugger public key	64 / 96

Table continues on the next page...

Table 203. Debug Credential Certificate fields (continued)

Name	Offset	Description	Size in bytes
SIGNATURE	variable	R and S portion of the ECDSA cryptographic signature (P-256 or P-384) by the RoT over the previous fields. This signature ensures that the DC is approved by the vendor for use by the debugger.	64 / 96

Table 204. ROTMETA_FLAGS

Field	Description
3:0	Reserved
7:4	Number of records in ctrkHashTable [1-4]. When equal to 1, ctrkHashTable is not present and the device computes the hash from ROT_KEY_PUB and compares it to RKTH stored in CUST_PROD_OEMFW_AUTH_PUK fuse.
11:8	Determine which root cert number [0-3] was used for signing. Used only when more than one root certificate is specified.
30:12	Reserved
31	CA flag Should be always set to 1b in DC

11.3.9.1 Debug Authentication Challenge (DAC)

The debug authentication protocol begins with a DAC message issued by the device to the debugger. The total message size is 104 bytes.

From a protocol perspective, the debugger must select a Debug Credential (DC) certificate that successfully authenticates based on the constraints provided in the DAC. This DC provides the required debug behavior post-authentication (for example, whether to debug secure world, with the desired credential beacon). If such a credential cannot be found, the debugger should report an error to the user.

NOTE

The debugger must also be able to produce signatures using the private key corresponding to the selected DC. This requirement exists so that any credential store can manage this association between credentials and corresponding private keys.

The named elements of this message are shown below.

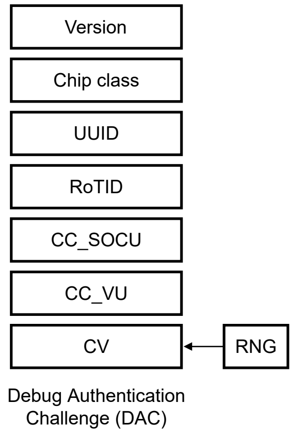


Figure 59. DAC fields

Table 205. DAC fields

Name	Offset	Description	Size in bytes
VERSION	000h	Identifies the Debug Authentication protocol version. Version determined based on DCFG_SOCU_L1[20:19] fuse. • 00020000h for v2.0, which uses ECDSA P-256 (DCFG_SOCU_L1[20:19] - 10b) • 00020001h for v2.1, which uses ECDSA P-384 (DCFG_SOCU_L1[20:19] - 00b, 01b, 11b)	4
SOCC	004h	SoC class specifier This field is always set to 9h.	4
UUID	008h	Unique device identifier If UUID matching is enabled in DCFG_CC_SoCU, then the device includes its UUID in the challenge packet. Otherwise, this field is set to zeroes.	16
ROTID	018h	Metadata used to authenticate Certificate Authority key (RoT key) for signing DC certificate. SHA256 or SHA384 hashes of four RoT public keys are set in this field. See Root of Trust Identifier (DCFG_ROTID) for details. NOTE On this device, the same RoT keys are used for certifying image signing keys and debug keys. 32-byte value (in the case of SHA384, only the 1st 32 bytes are provided) 4 bytes containing revocation flags. Only the least significant 4 bits are used.	36
CC	3Ch or 4Ch	Credential Constraints Specifies the debug access rights for this certificate holder.	12

Table continues on the next page...

Table 205. DAC fields (continued)

Name	Offset	Description	Size in bytes
		A 32-bit SOCU_PIN value. See the CC_LIST_Table in Credential Constraints (DCFG_CC_SOCU) for bit encoding of this field. A 32-bit SOCU_DFLT value. See the CC_LIST_Table in Credential Constraints (DCFG_CC_SOCU) for bit encoding of this field. A 32-bit DCFG_CC_VU value.	
CV	48h or 58h	Challenge Vector generated by the device. Device provides a random 32-byte value generated using the TRNG block.	32

11.3.9.2 Debug Authentication Response (DAR)

Before the debugger can formulate a response to the challenge, it should perform some checks to confirm the correctness of VER, SoCC, UUID, RoTID, and CC. It should find a matching DC.

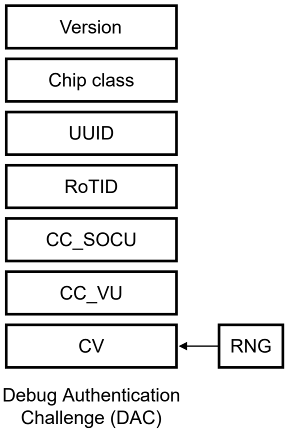


Figure 60. DAR fields

Table 206. DAR fields

Name	Description	Size in bytes
DC	Provides the credential, RoT public key, and more for the debugger as described in Debug Credential Certificate (DC).	variable
AB	The Authentication Beacon is provided and signed by the debugger during the authentication process. See the Credential Beacon (CB) field in Debug Credential Certificate (DC) structure for details.	4
SIG	A cryptographic signature by the debugger that binds the previous two fields with the challenge vector from the DAC. SIG = ECDSA_SIGN (DCKpriv,SHA(DAR::DC II DAR::AB II DAR::CV)) • Uses the private key corresponding to the public key (DCK) of the selected DC, proving that the debugger has possession of the debugger private key.	64 or 96

Table continues on the next page...

Table 206. DAR fields (continued)

Name	Description	Size in bytes
	• Depending on which Debug Authentication protocol is used, ECDSA secp256r1 or secp384r1 is used for SIG function. SHA is either SHA256 or SHA384 based on the ECDSA curve type.	

11.3.9.3 Device processing the DAR

The device Boot ROM processes the DAR received from the debugger. As part of the validation process, the device will:

- Verify the DC by validating the DC version, SoCC, UUID, RoT, VU, and DC signature.
- Verify that the DAR has a valid signature that binds it to the CV from the DAC.

If valid, it means that:

- The debugger possesses the private key corresponding to the vendor or RoT-signed credential.
- The credential satisfies the constraints specified in the device configuration.
- The response of the debugger to the challenge is produced and signed in response to the challenge (because of its cryptographic dependency on the challenge vector). The response is not replayed from a previous authentication where a different challenge vector is used.

After completion of processing the DAR:

- If authentication succeeds, debug access is granted.
- If authentication fails, no special response is issued. Further debug access requests are ignored and the device enters a failure loop (an infinite loop waiting for debug attach).

11.3.9.3.1 Successful authentication

The ROM executes the following steps after successful debug authentication:

1. The ROM determines the final enable states of the debug ports based on pinned state from DCFG_CC_SOCU and the DC::CC fields.
2. The ROM evaluates part enables using the following logic:
    - Uses pinned states based on DCFG_CC_SOCU_L1 and DCFG_CC_SOCU_L2.
    - Evaluates socu_pinned and socu_default.
    - Evaluates debug port enables for ports that are not pinned using authentication protocol.
    - Debug_State = (SOCU_PIN & SOCU_DFLT) | (!SOCU_PIN & DC:: CC_SOCU)
    - Enables debug ports for bits set in the above evaluation.
3. The Debug Mailbox handler allows the following commands only if the enable bit is set in the final evaluation of DCFG_CC_SOCU.
    - ENTER_ISP_MODE command, only if the default ISP_CMD_EN bit is set in DCFG_CC_SOCU.
    - SET_FA_MODE commands, only if the default FA_CMD_EN bit is set in DCFG_CC_SOCU.
4. The ROM stores the beacons in the write-lockable register DBG_AUTH_SCRATCH.
    - DBG_AUTH_SCRATCH [15:0] = DAR::DC::CB[15:0]
    - DBG_AUTH_SCRATCH [31:16] = DAR::AB[15:0]
5. On receiving an EXIT_DBG_MB command, the ROM exits the debug mailbox handler loop and continues normal boot flow.

11.3.9.4 Debug authentication use cases

This section describes use cases for debug authentication.

11.3.9.4.1 Return Material Analysis (RMA) use case

The diagram shows the RMA flow using debug authentication, where a debug credential certificate is issued for each field technician.

1. Vendor generates RoT key pairs and programs the device with a hash of RoT public key hashes before shipping.
2. Field technician generates own key pair and provides public key to vendor for authorization.
3. Vendor attests public key for field technician. In the debug credential certificate, vendor assigns access rights.
4. End customer with a locked product takes it to field technician.
5. Field technician uses credentials to authenticate with device and unlocks the product for debugging.

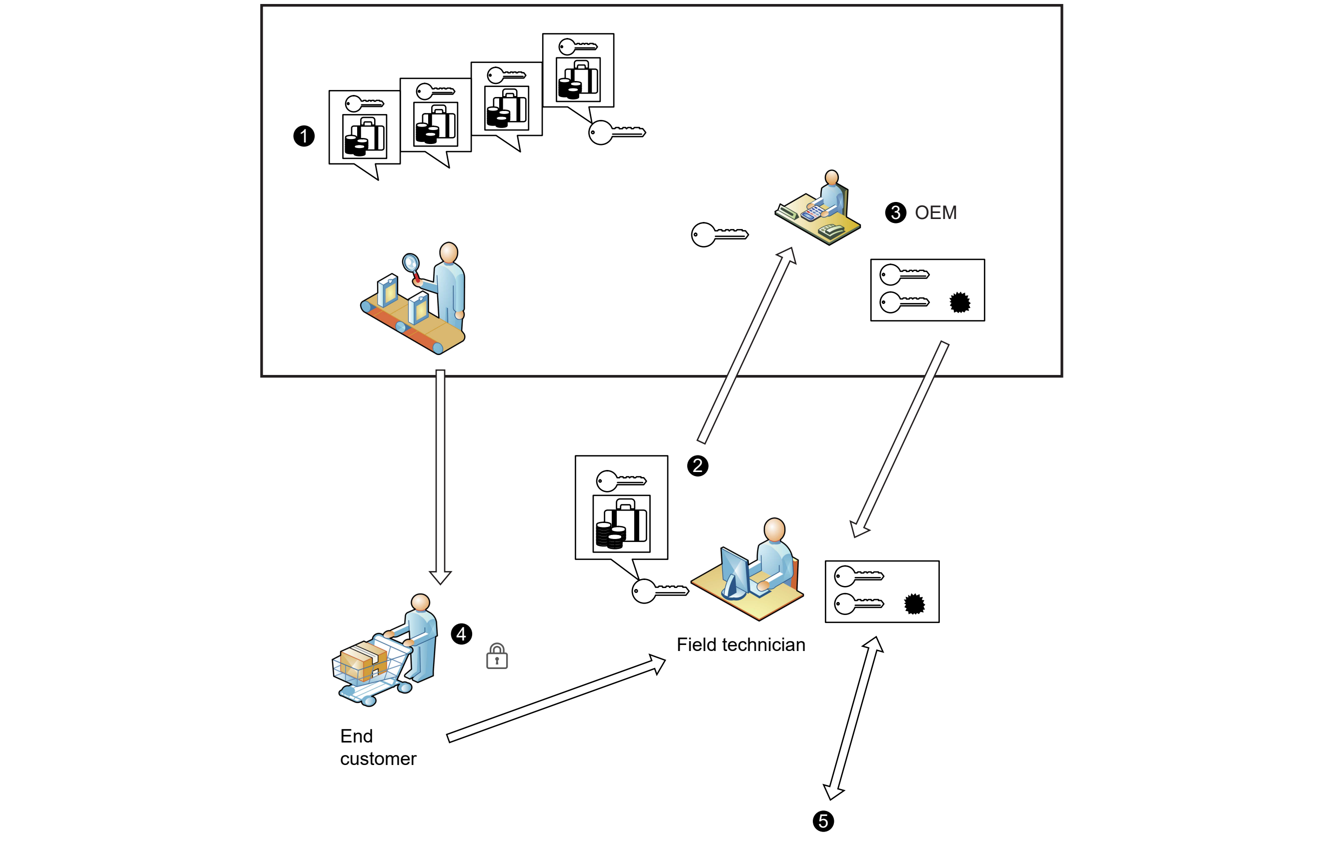


Figure 61. Debug Authentication protocol usage example

11.3.9.4.2 Module use case with Develop and Develop 2 Lifecycle states

DCFG_CC_SOCU_L2 allows module makers and OEMs to implement a tiered protection approach.

- The module maker, a tier-1 developer, develops secure code and define access rights to the module using DCFG_CC_SOCU_L1.
- The code is configured to block access to the secure module but allow access to non-secure debug.
- Once the module is ready, the tier-1 developer releases the module to a tier-2 developer (for example, an OEM). The developer blocks debug access to secure mode and enables debug access to non-secure mode.

- A tier-2 developer develops a non-secure module and extends access rights configuration to that module using DCFG_CC_SOCU_L2.

11.3.10 Fault Analysis (FA) mode

The ROM on this chip has an FA mode command (SET_FA_MODE) to enable the deletion of sensitive information (such as keys) before giving the device to NXP for fault analysis. ROM allows Set FA Mode command only when the corresponding flag, FA_CMD_EN, is set and a valid signed image is sent with the command. Users must pass an FA request message to Set_FA_Mode command to enter FA mode.

When activated by the boot ROM, the FA_MODE sequence is:

1. Erase the internal flash.
2. Erase all OEM Keys in fuses.
3. Advance lifecycle to OEM return.
4. Reset the device
5. Bootloader starts in OEM return ROM mode.

11.4 External signals

Table 207 shows the signals related to the debug process. A trace using the serial wire output has limited bandwidth.

Table 207. Serial wire debug signals

Signal	I/O	Description
SWCLK	I	Serial wire clock. Provides the clock for the SWD debug logic in Serial Wire Debug (SWD) mode. SWCLK is the default signal for its pin. At the release of reset, the pin is pulled down internally.
SWDIO	I/O	Serial wire debug data input and output. Used by an external debug tool to communicate with and control the part. SWDIO is the default signal of its pin. At the release of reset, the pin is pulled up internally.
SWO	O	Serial wire output. Optionally provides data from the Instrumentation Trace Macrocell (ITM) for an external debug tool to evaluate. See the chip-specific DBGMB information for the clocking required to enable SWO.

11.5 Memory map and register definition

11.5.1 DBGMB register descriptions

11.5.1.1 DBGMB memory map

DBGMB base address: 4800_0000h

Offset	Register	Width (In bits)	Access	Reset value
0h	Command and Status Word (CSW)	32	RW	0000_0000h
4h	Request Value (REQUEST)	32	RW	0000_0000h
8h	Return Value (RETURN)	32	RW	0000_0000h
FCh	Identification (ID)	32	R	002A_0000h

11.5.1.2 Command and Status Word (CSW)

Offset

Register	Offset
CSW	0h

Function

Contains command and status bits to facilitate communication between DBGMB and the chip.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	Reserved															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	Reserved										CHIP_RE...	SOFT_RE...	AHB_OR_...	DBG_OR_...	REQ_PEN...	RESYNCH...
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-6 —	Reserved
5 CHIP_RESET_REQ	Chip Reset Request Causes the chip (but not the DM-AP) to be reset by generating SYSRESET_REQ. This field is write only. 0b - No effect 1b - Reset
4 SOFT_RESET	Soft Reset Resets the DM-AP. This field is write only by the chip. 0b - No effect 1b - Reset
3 AHB_OR_ERR	AHB Overrun Error Indicates whether an AHB overrun has occurred: the chip has overwritten a RETURN value before DBGMB read the RETURN value. 0b - No overrun

Table continues on the next page...

Table continued from the previous page...

Field	Function
	1b - Overrun occurred
2 DBG_OR_ERR	DBGMB Overrun Error Indicates whether a DBGMB overrun has occurred: DBGMB has overwritten a REQUEST value before the chip read the REQUEST value. 0b - No overrun 1b - Overrun occurred
1 REQ_PENDING	Request Pending Indicates a pending request for DBGMB: a value is waiting to be read from Request Value (REQUEST). 0b - No request pending 1b - Request for resynchronization pending
0 RESYNCH_REQ	Resynchronization Request Requests a resynchronization. 0b - No request 1b - Request for resynchronization

11.5.1.3 Request Value (REQUEST)

Offset

Register	Offset
REQUEST	4h

Function
Used by DBGMB to send action requests to the chip.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	REQUEST															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	REQUEST															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 REQUEST	Request Value Indicates the request value. This field reads 0 when no new request is present. The chip clears this field. DBGMB can read back this field to confirm communication.

11.5.1.4 Return Value (RETURN)

Offset

Register	Offset
RETURN	8h

Function
Provides the responses from the chip to DBGMB.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	RET															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	RET															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 RET	Return Value Indicates the return value, which is a response from the chip to DBGMB. If no new data is present, DBGMB reading is stalled until new data is available.

11.5.1.5 Identification (ID)

Offset

Register	Offset
ID	FCh

Function

Provides an identification of the DM-AP interface.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	ID															
W																
Reset	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	ID															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 ID	Identification Value Provides an identification of the DM-AP interface. The Arm Debug Interface Specification version 5.0 (ADIv5) requires every AP to implement an AP identification register, at offset FCh. This register is the last register in the AP register space. This ID register is only readable by external tools (a debug dongle) when identifying the Access Port (AP). For the debug mailbox AP, the identification value is 002A_0000h. This register is not readable from on-chip software. If you attempt a read, you receive a value of 0000_0000h.

Chapter 12 Flash Memory Controller (FMC) with NVM PRINCE Encryption and Decryption (NPX)

12.1 Chip-specific Flash Memory Controller with NVM PRINCE Encryption and Decryption information

Table 208. Reference links to related information[1]

Topic	Related module	References
Full description	Flash Memory Controller (FMC) with NVM PRINCE Encryption and Decryption (NPX)	FMC NPX
System memory map		See the section "System memory map"
CLocking		See the chapter "Clocking"
Signal multiplexing	Port control	See the chapter "Signal Muxing and Pinout "

[1] For all the reference sections and chapters mentioned in this table, refer the Reference Manual.

12.1.1 Module instance

This device has one instance of the FMC NPX module.

12.1.2 Speculative reads control

On this device, speculative reads are controlled via the SMSCM_OCMDR0 register, using the OCMCF1[1:0] bits to enable or disable speculative accesses. See Speculative reads for more details.

12.2 Overview

The Flash Memory Controller (FMC) is a memory interface and acceleration unit providing:

- An interface between the device and the nonvolatile memory
- Buffers that can accelerate flash memory transfers
  - — A flash-phrase-sized(128 bits) single-entry buffer holds the most recently accessed flash phrase.
  - — An optional flash-phrase-sized(128 bits) speculation buffer can prefetch the next flash phrase.
  - — An optional 4-way,1-set, 128-bit line size flash cache can store previously accessed flash phrases.
- An optional NPX (NVM Prince XEX) module to perform On-The-Fly (OTF) read decryption and write encryption of flash memory contents

The FMC manages the interface between the device and the flash memory. The FMC receives status information describing the configuration of the memory and uses this information to ensure a proper interface. The following table shows the supported read and write operations.

Flash memory type	Read	Write
Program, IFR0, IFR1 flash memory	8-bit, 16-bit, and 32-bit reads	16-byte and 128-byte writes

The FMC is controlled by a programmer's model external to the FMC module; see the chip-specific FMC information for details. The FMC's programming model provides a very configurable, high performance flexible memory controller, which can be optimized for the runtime characteristics of specific applications.

> NOTE
>
> Program the FMC's controls only while the flash controller is idle. Changing the configuration settings while a flash access is in progress can cause non-deterministic, unpredictable behavior.

12.2.1 Features

- Interface between the device and the flash memory:
  - The FMC's input bus supports 8-bit, 16-bit, and 32-bit read operations to flash memory.
  - The FMC's flash memory interface fetches a 128-bit flash phrase.
  - For input read requests, the FMC fetches a flash phrase with the desired read data from flash memory.
  - The flash memory interface can write aligned 16-byte flash write phrases or aligned 128-byte flash write pages to flash memory.
  - The FMC has a 16-byte aligned write buffer. This buffer is used once for aligned 16-byte flash write phrases or eight times for aligned 128-byte flash write pages.
  - The FMC's input bus supports 32-bit write operations for flash memory writes to fill the FMC's write buffer.
  - For input write requests, the FMC must receive the 4-longword(16-byte) write of an aligned phrase in order.
  - For input write requests, the FMC supports write cancel.
  - Program flash and IFR0 support read and write requests, IFR1 only supports read requests.
- Acceleration of data transfer from flash memory to the device:
  - A flash-phrase-sized single-entry buffer that holds the current decrypted flash phrase fetched due to a FMC read request. Subsequent FMC read requests *that hit in the single-entry buffer* return data with no wait states.
  - A flash-phrase-sized prefetch speculation buffer with controls for prefetching on instructions and/or data reads. When prefetching is enabled, idle FMC-to-flash interface cycles are used to fetch the next sequential flash phrase and hold it in the prefetch buffer. Subsequent FMC read requests *that hit in the speculation buffer* return data with no wait states.
  - Input controls:
    - To disable data type speculation
    - To disable all speculation
    - To clear the flash cache,single-entry buffer and speculation buffer
    - To disable the flash cache,single-entry buffer and speculation buffer
    - To enable AHB bus stall on flash busy
    - To lock access to IFR1
    - To disable AHB bus error on flash multibit ECC error for data
    - To disable AHB bus error on flash multibit ECC error for instruction

    To enable flash speculation, enable AHB bus error on flash multibit ECC error for data and instruction. See the chip-specific section for details about controls.
  - Flash cache
    - Only Program flash is cacheable, IFR0 and IFR1 are not cacheable.
  - The flash cache has input controls:
    - To disable instruction caching(disable cache loads)

  - ◦ To disable data caching(disable cache loads)
  - ◦ To disable all caching(disable cache loads and hits)
  - ◦ To clear the cache(tag, data and valid)

  See the chip-specific section for details about controls.

- — The size of the flash cache in bytes is calculated as follows:

  flash cache size = [number of ways] × [number of sets] × [flash phrase size (in bytes)]

  For example, a flash cache with 4 ways, 1 set, and a 128-bit flash phrase (= 16 bytes) has a total flash cache size = 64 bytes

  (4 ways) × (1 set) × (16 bytes per flash phrase) = 64 bytes, the size of the flash cache

NOTE

Clear the speculation buffer and flash cache before accessing recently modified flash addresses. The flash cache has a specific clear control bit. To clear the speculation buffer, first disable then re-enable the speculation via other modules.

12.3 Functional description

The FMC is a flash interface and acceleration unit, with flexible buffers for user configuration.

- The FMC's input bus can operate faster than the flash memory.
- The FMC-to-flash interface has flow control to add wait states as needed (for input bus reads that need flash accesses).
- The FMC also contains various configurable buffers that hold recent flash accesses. If an input bus read hits a valid buffer, then that access will complete with no wait states.

12.3.1 Modes of operation

The FMC only operates when a bus master accesses the flash memory.

For any device power mode where the flash memory cannot be accessed, FMC is disabled automatically.

12.3.2 Default configuration

After system reset, the FMC is configured to provide a significant level of buffering for transfers from the flash memory. For all banks:

- The single-entry and speculation buffers are cleared by reset.
- Prefetch support by speculation buffer for data and instructions is enabled.
- The flash cache is cleared by reset.
- The flash cache is configured for data or instruction replacement.

12.3.3 Configuration options

The default configuration provides a high degree of flash acceleration, but advanced users may want to customize the FMC buffer configurations to maximize throughput for their use cases. When reconfiguring the FMC for custom use cases, do not program the FMC's control registers while the flash memory is being accessed. Instead, change the control registers with a routine executing from RAM in Supervisor mode.

The FMC's flash cache and buffering controls allow other modules to tune resources to suit specific application requirements. The flash cache and buffer are each controlled individually. The controls enable caching and prefetching per access type (instruction fetch or data reference).

As an application example: if both instruction fetches and data references are accessing flash memory, then control is available to send instruction fetches, data references, or both to the flash cache or the single-entry buffer. Likewise, speculation can be enabled or disabled for either type of access.

For best performance, the FMC flash cache and speculation buffering should be enabled for instruction, data fetching, or both. The following is recommended for best performance:

- If the speculation buffer is enabled for both instruction and data speculation, enable the flash cache for both instruction and data caching.
- If the speculation buffer is enabled for instruction speculation only, enable the flash cache for at least instruction caching.

12.3.4 Wait states

Because the core, crossbar switch, and bus masters can be clocked at a higher frequency than the flash clock, flash memory accesses that do not hit in the speculation buffer or cache usually require wait states.

FMC does not allow the configuration of wait states directly. Wait states can be controlled via FMU FCTRL[RWSC].

12.3.5 Speculative reads

The FMC has a speculation buffer that reads ahead to the next phrase in the flash memory if there is an idle cycle. Speculative prefetching is programmable for instruction and data accesses. Because many code accesses are sequential, using the speculative prefetch buffer improves performance in most cases.

Program flash and IFR0 support speculation reads, IFR1 doesn't support speculation reads. See the chip-specific section for information about controlling speculative reads.

When speculative reads are enabled, the FMC immediately requests the next sequential phrase address after a read completes. By requesting the next phrase immediately, speculative reads can help to reduce or even eliminate wait states when accessing sequential code and/or data.

12.3.6 NPX submodule

NPX is the FMC NVM On-The-Fly PRINCE Encryption and Decryption submodule. A trend in embedded processor design is an increasing need for hardware to support cryptographic calculations that are required for system security. There are emerging customer requirements to protect application code and data stored in flash memories in an encrypted form, and to have the embedded processor provide hardware support for "on-the-fly" decryption (meaning that the data is automatically decrypted as the data is loaded or saved). The flash memory image of the code and data is always stored in an encrypted format, and in response to processor references to the address space, the memory image is decrypted on the fly, returning the original value to the requesting bus master.

The FMC Controller's NPX (NVM PRINCE Encryption and Decryption) submodule provides both "on-the-fly" decryption and encryption. The "on-the-fly" encryption is used to program the flash with encrypted data, without needing external support to pre-encrypt the program data. Using this method, there is no need for external knowledge of keys for the system to benefit from the additional security of encrypted flash memory.

A cryptographic symmetric key block cipher algorithm is used for the encryption and decryption. The specific algorithm used is PRINCE. PRINCE is a symmetric key cipher, originally developed by NXP-Leuven and 3 European universities, that provides a lightweight hardware implementation cost, but still provides strong cryptographic protection. PRINCE operates on 64-bit (8-byte) data blocks with 128-bit secret keys.

The On-the-Fly (OTF) PRINCE engine operates in conjunction with the internal flash memory controller.

The NPX OTF encrypt/decrypt engine provides superior cryptographic capabilities without compromising system performance for the embedded processor applications that require enhanced security.

12.3.6.1 NPX features

- PRINCE On-the-Fly Encryption and Decryption
    - 128-bit keys and 64-bit data block sizes

- — Adds one cycle of incremental latency for encryption or decryption
- Hardware support for 4 independent encryption/decryption blocks, known as memory "regions." Each memory region defines a unique address space and has a unique, unreadable 128-bit key.
- Functionally acts as a slave submodule to the Flash Memory Controller (FMC)
  - — Logically connected between the FMC flash cache and the FMC flash interface
  - — Programming model is mapped to the FMC peripheral address space
- Hardware microarchitecture
  - — 128-bit (16-byte) input/output buses match flash read/write width
  - — Single cycle encryption or decryption engine using 2 parallel PRINCE instantiations
  - — Optimized for 32-bit WRAP4 bursts (CPU cache miss fetch size, typical DMA fetch size)

12.3.6.2 NPX block diagram

The NPX is directly connected to the Flash Memory Controller (FMC).

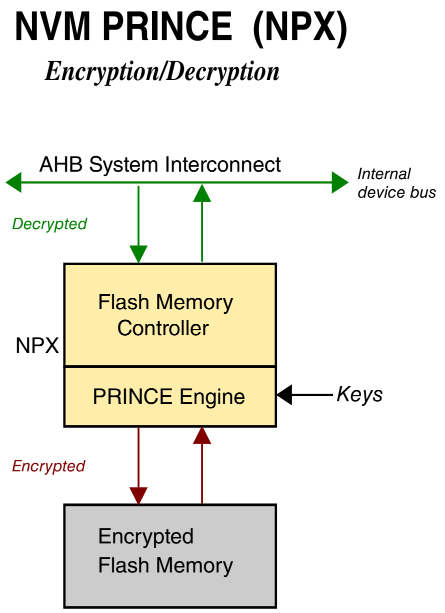


Figure 62. NPX block diagram

12.3.6.3 NPX modes of operation

12.3.6.4 Obfuscating cache data

When the flash cache is enabled, each 32-bit word of data is automatically exclusive OR’ed with the cache obfuscation mask () before being stored in the cache. Cache data is unmasked when read. To modify CACMSK:

1. Disable the flash cache.
2. Modify .
3. Clear the cache.
4. Re-enable the cache.

12.3.6.5 Remapping flash addresses

An address remap mechanism allows for the swapping (both ways) of a specified flash address range between flash bank 0 and flash bank 1. This remapping can facilitate software updates in the field without the need for address modifications in software. For example, to use the remap mechanism to update software in the lower portion of bank 0, follow these steps:

1. Load the updated software into the lower portion of flash bank1.
2. Enable remapping to automatically use the updated software.

In , remapping is enabled when LIM[20:16] = LIMDP[28:24], and LIM[20:16] is nonzero. LIM[20:16] and LIMDP[28:24] define the *remap_address*[17:13] for the remapping, providing an address range granularity of 8 KB. For a remapped FMC access to address *access_address*, see Table 209.

Table 209. Remapping *access_address*

When *access_address* originates in...	And is less than or equal to...	The access remaps to...	In...
flash bank 0	*remap_address*[17:13]	bank1_base + access_address	bank 1
flash bank 1	bank1_base + *remap_address*[17:13]	*access_address* – bank1_base	bank 0

The address ranges that may be swapped depends on the total flash size available. For non-power-of-2 total flash sizes, the upper half of flash bank 0 cannot be swapped, as shown in Table 210.

Table 210. Remapping address ranges

For a flash size of...	This bank 0 range...	Remaps to this bank 1 range...	This bank 0 range cannot be remapped...
2 MB	0x00_0000 – 0x0F_FFFF	0x10_0000 – 0x1F_FFFF	
1536 KB	0x00_0000 – 0x07_FFFF	0x10_0000 – 0x17_FFFF	0x08_0000 – 0x0F_FFFF
1 MB	0x00_0000 – 0x07_FFFF	0x08_0000 – 0x0F_FFFF	
768 KB	0x00_0000 – 0x03_FFFF	0x08_0000 – 0x0B_FFFF	0x04_0000 – 0x07_FFFF
512 KB	0x00_0000 – 0x03_FFFF	0x04_0000 – 0x07_FFFF	
384 KB	0x0_0000 – 0x1_FFFF	0x4_0000 – 0x5_FFFF	0x2_0000 – 0x3_FFFF
256 KB	0x0_0000 – 0x1_FFFF	0x2_0000 – 0x3_FFFF	—
192 KB	0x0_0000 – 0x0_FFFF	0x2_0000 – 0x2_FFFF	0x1_0000 – 0x1_FFFF
128 KB	0x0_0000 – 0x0_FFFF	0x1_0000 – 0x1_FFFF	—
96 KB	0x0_0000 – 0x0_7FFF	0x1_0000 – 0x1_7FFF	0x0_8000 – 0x0_FFFF
64 KB	0x0_0000 – 0x0_7FFF	0x0_8000 – 0x0_FFFF	—
48 KB	0x0_0000 – 0x0_3FFF	0x0_8000 – 0x0_BFFF	0x0_4000 – 0x0_7FFF
32 KB	0x0_0000 – 0x0_3FFF	0x0_4000 – 0x0_7FFF	—

When LIM = LIMDP = 0, FMC disables the remap function. When LIM = LIMDP = a nonzero value, the remapping address range is (LIM + 1) × 8 KB. For example:

- When LIM = LIMDP = 1, the range is ≤ 16 KB.
- When LIM = LIMDP = 2, the range is ≤ 24 KB.

12.3.6.6 NPX initialization

The operating configuration of the NPX is controlled by programmable bits in .

12.3.6.7 NPX memory regions

Memory regions are defined as the address range from equal-to-or-greater-than the start address (such as Memory Region 0 Start Address (MR0STARTADDR)) to less-than-or-equal-to the end address (such as Memory Region 0 End Address (MR0ENDADDR)).

NOTE

Hardware does not check to verify that a memory region's end address is greater than its start address. Software must ensure appropriate values are loaded into these fields (start and end addresses) of the memory region descriptor.

Each memory region has its own PRINCE key generated from the region's masked key and its corresponding mask:

- 128-bit PRINCE key = (128-bit masked key) XOR (128-bit key mask)

For system accesses that hit in multiple regions (or no regions), the fetched data is simply bypassed. NPX does not support memory region overlap, because each region has a unique key.

12.3.6.8 Interrupts

This module has no interrupts.

12.4 External signals

The FMC has no external signals.

12.5 Initialization and application information

The FMC does not require user initialization. Flash acceleration features are enabled by default.

The FMC has no visibility into flash memory erase and program cycles because the Flash Memory module manages them directly. To prevent the possibility of returning stale data after an application executes flash memory commands, disable and/or clear FMC's single-entry buffer, speculation buffer, and cache. If you want to disable the speculation buffer and cache in certain specific cases, we recommend disabling them all.

See the chip-specific section for details about the registers used to disable and/or clear flash cache, speculation buffer and single-entry buffer.

12.6 Register descriptions

The NPX (NVM Prince XEX Module) register set has a control register and a status register, plus information for 4 memory regions. Each memory region has:

- 4 masked key words (128 bits total masked key size)
- 4 key mask words (128 bits total mask size)
- a memory region start address
- a memory region end address

NOTE

See the chip-specific information for FMC controls.

3-bit pole/anti-pole control fields:

For critical control and configuration functions, a 3-bit pole/anti-pole scheme improves reliability by using the middle bit as the decisive value. When reading the pole/anti-pole field:

- 101b is the negated state
- 010b is the asserted state

When writing the pole/anti-pole field:

- 101b negates the state (W5C: "write 5 to clear")
- 010b asserts the state (W2S: "write 2 to set")

If the state changes to a value different from 010b or 101b, an internal security violation is signaled.

NOTE

Any access to an undefined memory area results in a bus error.

12.6.1 FMC register descriptions

12.6.1.1 FMC memory map

NPX base address: 4002_5000h

Offset	Register	Width (In bits)	Access	Reset value
0h	NPX Control Register (NPXCR)	32	RW	0000_0000h
4h	NPX Status Register (NPXSR)	32	R	0000_0004h
100h	Memory Region 0, Masked Key Word 0 (MR0MASKEDKEYWORD0)	32	W	0000_0000h
104h	Memory Region 0, Masked Key Word 1 (MR0MASKEDKEYWORD1)	32	W	0000_0000h
108h	Memory Region 0, Masked Key Word 2 (MR0MASKEDKEYWORD2)	32	W	0000_0000h
10Ch	Memory Region 0, Masked Key Word 3 (MR0MASKEDKEYWORD3)	32	W	0000_0000h
110h	Memory Region 0, Mask for Key Word 0 (MR0MASKFORKEYWORD0)	32	W	0000_0000h
114h	Memory Region 0, Mask for Key Word 1 (MR0MASKFORKEYWORD1)	32	W	0000_0000h
118h	Memory Region 0, Mask for Key Word 2 (MR0MASKFORKEYWORD2)	32	W	0000_0000h
11Ch	Memory Region 0, Mask for Key Word 3 (MR0MASKFORKEYWORD3)	32	W	0000_0000h
120h	Memory Region 0 Start Address (MR0STARTADDR)	32	W	0000_0000h
124h	Memory Region 0 End Address (MR0ENDADDR)	32	W	0000_01FDh
140h	Memory Region 1, Masked Key Word 0 (MR1MASKEDKEYWORD0)	32	W	0000_0000h
144h	Memory Region 1, Masked Key Word 1 (MR1MASKEDKEYWORD1)	32	W	0000_0000h
148h	Memory Region 1, Masked Key Word 2 (MR1MASKEDKEYWORD2)	32	W	0000_0000h
14Ch	Memory Region 1, Masked Key Word 3 (MR1MASKEDKEYWORD3)	32	W	0000_0000h
150h	Memory Region 1, Mask for Key Word 0 (MR1MASKFORKEYWORD0)	32	W	0000_0000h

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
154h	Memory Region 1, Mask for Key Word 1 (MR1MASKFORKEYWORD1)	32	W	0000_0000h
158h	Memory Region 1, Mask for Key Word 2 (MR1MASKFORKEYWORD2)	32	W	0000_0000h
15Ch	Memory Region 1, Mask for Key Word 3 (MR1MASKFORKEYWORD3)	32	W	0000_0000h
160h	Memory Region 1 Start Address (MR1STARTADDR)	32	W	0000_0000h
164h	Memory Region 1 End Address (MR1ENDADDR)	32	W	0000_01FDh
180h	Memory Region 2, Masked Key Word 0 (MR2MASKEDKEYWORD0)	32	W	0000_0000h
184h	Memory Region 2, Masked Key Word 1 (MR2MASKEDKEYWORD1)	32	W	0000_0000h
188h	Memory Region 2, Masked Key Word 2 (MR2MASKEDKEYWORD2)	32	W	0000_0000h
18Ch	Memory Region 2, Masked Key Word 3 (MR2MASKEDKEYWORD3)	32	W	0000_0000h
190h	Memory Region 2, Mask for Key Word 0 (MR2MASKFORKEYWORD0)	32	W	0000_0000h
194h	Memory Region 2, Mask for Key Word 1 (MR2MASKFORKEYWORD1)	32	W	0000_0000h
198h	Memory Region 2, Mask for Key Word 2 (MR2MASKFORKEYWORD2)	32	W	0000_0000h
19Ch	Memory Region 2, Mask for Key Word 3 (MR2MASKFORKEYWORD3)	32	W	0000_0000h
1A0h	Memory Region 2 Start Address (MR2STARTADDR)	32	W	0000_0000h
1A4h	Memory Region 2 End Address (MR2ENDADDR)	32	W	0000_01FDh
1C0h	Memory Region 3, Masked Key Word 0 (MR3MASKEDKEYWORD0)	32	W	0000_0000h
1C4h	Memory Region 3, Masked Key Word 1 (MR3MASKEDKEYWORD1)	32	W	0000_0000h
1C8h	Memory Region 3, Masked Key Word 2 (MR3MASKEDKEYWORD2)	32	W	0000_0000h
1CCh	Memory Region 3, Masked Key Word 3 (MR3MASKEDKEYWORD3)	32	W	0000_0000h
1D0h	Memory Region 3, Mask for Key Word 0 (MR3MASKFORKEYWORD0)	32	W	0000_0000h
1D4h	Memory Region 3, Mask for Key Word 1 (MR3MASKFORKEYWORD1)	32	W	0000_0000h
1D8h	Memory Region 3, Mask for Key Word 2 (MR3MASKFORKEYWORD2)	32	W	0000_0000h
1DCh	Memory Region 3, Mask for Key Word 3 (MR3MASKFORKEYWORD3)	32	W	0000_0000h

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
1E0h	Memory Region 3 Start Address (MR3STARTADDR)	32	W	0000_0000h
1E4h	Memory Region 3 End Address (MR3ENDADDR)	32	W	0000_01FDh

12.6.1.2 NPX Control Register (NPXCR)

Offset

Register	Offset
NPXCR	0h

Function

The NPX Control Register (NPXCR) contains enables for Global Lock, System Lock, Global Decryption and Global Encryption.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	GLK			0	SLK			0	GDE			0	GEE		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-15 —	This read-only field is reserved and always has the value 0
14-12 GLK	Global Lock Enable Set by software (W2S, sticky); cleared only by a reset. Attempted writes with values other than 010b or 101b do not change the register state. 010b - Lock enabled: cannot write to keys, masks, regions, NPXCR (NPX Control Register), NPXSR (NPX Status Register). Subsequent reads return 010b. 101b - Lock disabled. Subsequent reads return 000b.

Table continues on the next page...

Table continued from the previous page...

Field	Function
11 —	This read-only field is reserved and always has the value 0
10-8 SLK	System Lock Enable Set by software (W2S, sticky); cleared only by a reset. Attempted writes with values other than 010b or 101b do not change the register state. 010b - Lock enabled: cannot write (via IPS) to keys, masks, regions, NPXCR (NPX Control Register), NPXSR (NPX Status Register). Subsequent reads return 010b. 101b - Lock disabled Subsequent reads return 000b.
7 —	This read-only field is reserved and always has the value 0
6-4 GDE	Global Decryption Enable Set by software (W2S); cleared by software (W5C); cleared by POR (Power-On Reset). Attempted writes with values other than 010b or 101b do not change the register state. 010b - Global decryption enabled. NPX on-the-fly decryption is globally enabled. Subsequent reads return 010b. 101b - Global decryption disabled. NPX on-the-fly decryption is globally disabled. Subsequent reads return 000b.
3 —	This read-only field is reserved and always has the value 0
2-0 GEE	Global Encryption Enable Set by software (W2S); cleared by software (W5C); cleared by POR (Power-On Reset). Attempted writes with values other than 010b or 101b do not change the register state. 010b - Global encryption enabled. NPX on-the-fly encryption is enabled if the flash access hits in a valid memory region. Subsequent reads return 010b. 101b - Global encryption disabled. NPX on-the-fly encryption is disabled. Subsequent reads return 000b.

12.6.1.3 NPX Status Register (NPXSR)

Offset

Register	Offset
NPXSR	4h

Function

Contains the Key 0-3 validity flags and the number of supported memory regions (NRGD) .

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0				V3	V2	V1	V0	0				NRGD			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Fields

Field	Function
31-12 —	This read-only field is reserved and always has the value 0
11-8 Vn	Key n Valid V *n* is a read-only copy of the corresponding memory region *n* descriptor's valid bit. 0b - Not valid 1b - Valid
7-4 —	This read-only field is reserved and always has the value 0
3-0 NRGD	Number of implemented memory regions Contains the number of supported memory regions. In this device, there are 4 memory regions. 0000b - No (zero) implemented memory regions 0001b - 1 implemented memory region 0010b - 2 implemented memory regions 0011b - 3 implemented memory regions 0100b - 4 implemented memory regions

12.6.1.4 Memory Region 0, Masked Key Word 0 (MR0MASKEDKEYWORD0)

Offset

Register	Offset
MR0MASKEDKEYWORD0	100h

Function

Contains bits [31:0] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY0WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY0WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY0WORD0	Masked Key Word 0 Bits [31:0] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.5 Memory Region 0, Masked Key Word 1 (MR0MASKEDKEYWORD1)

Offset

Register	Offset
MR0MASKEDKEYWORD1	104h

Function

Contains bits [63:32] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY0WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY0WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY0 WORD1	Masked Key Word 1 Bits [63:32] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.6 Memory Region 0, Masked Key Word 2 (MR0MASKEDKEYWORD2)

Offset

Register	Offset
MR0MASKEDKEYWORD2	108h

Function

Contains bits [95:64] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY0WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY0WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY0WORD2	Masked Key Word 2 Bits [95:64] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.7 Memory Region 0, Masked Key Word 3 (MR0MASKEDKEYWORD3)

Offset

Register	Offset
MR0MASKEDKEYWORD3	10Ch

Function

Contains bits [127:96] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY0WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY0WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY0WORD3	Masked Key Word 3 Bits [127:96] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.8 Memory Region 0, Mask for Key Word 0 (MR0MASKFORKEYWORD0)

Offset

Register	Offset
MR0MASKFORKEYWORD0	110h

Function

Contains bits [31:0] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK0WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK0WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK0WORD0	Mask for Key Word 0 Bits [31:0] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.9 Memory Region 0, Mask for Key Word 1 (MR0MASKFORKEYWORD1)

Offset

Register	Offset
MR0MASKFORKEYWORD1	114h

Function

Contains bits [63:32] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK0WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK0WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK0WORD1	Mask for Key Word 1 Bits [63:32] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.10 Memory Region 0, Mask for Key Word 2 (MR0MASKFORKEYWORD2)

Offset

Register	Offset
MR0MASKFORKEYWORD2	118h

Function

Contains bits [95:64] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK0WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK0WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK0WORD2	Mask for Key Word 2 Bits [95:64] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.11 Memory Region 0, Mask for Key Word 3 (MR0MASKFORKEYWORD3)

Offset

Register	Offset
MR0MASKFORKEYWORD3	11Ch

Function

Contains bits [127:96] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK0WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK0WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK0WORD3	Mask for Key Word 3 Bits [127:96] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.12 Memory Region 0 Start Address (MR0STARTADDR)

Offset

Register	Offset
MR0STARTADDR	120h

Function

Contains the start address for this memory region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	SRTADDR0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R								0								
W	SRTADDR0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-9 SRTADDR0	Start Address for Memory Region 0 Start address for this memory region. Bits [8:0] of the start address are always taken as 0s.
8-0 —	This read-only field is reserved and always has the value 0.

12.6.1.13 Memory Region 0 End Address (MR0ENDADDR)

Offset

Register	Offset
MR0ENDADDR	124h

Function

Contains the end address and the valid indicator for this memory region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	ENDADDR0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	ENDADDR0							Reserved						V		
Reset	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1

Fields

Field	Function
31-9 ENDADDR0	End Address for Memory Region 0 End address for this memory region. Bits [8:0] of the end address are always taken as 1s.
8-3 —	This write-only field is reserved and always has the value 1.
2-0 V	Memory Region 0 is Valid Set by software (W2S, sticky); cleared by POR only (Power-On Reset). Attempted writes with values other than 010b do not change the register state. 010b - Memory Region 0 is valid. Subsequent reads return 111b. 101b - Memory Region 0 is not valid.

12.6.1.14 Memory Region 1, Masked Key Word 0 (MR1MASKEDKEYWORD0)

Offset

Register	Offset
MR1MASKEDKEYWORD0	140h

Function

Contains bits [31:0] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY1WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY1WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY1WORD0	Masked Key Word 1 Bits [31:0] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.15 Memory Region 1, Masked Key Word 1 (MR1MASKEDKEYWORD1)

Offset

Register	Offset
MR1MASKEDKEYWORD1	144h

Function

Contains bits [63:32] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY1WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY1WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY1WORD1	Masked Key Word 1 Bits [63:32] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.16 Memory Region 1, Masked Key Word 2 (MR1MASKEDKEYWORD2)

Offset

Register	Offset
MR1MASKEDKEYWORD2	148h

Function

Contains bits [95:64] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY1WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY1WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY1WORD2	Masked Key Word 2 Bits [95:64] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.17 Memory Region 1, Masked Key Word 3 (MR1MASKEDKEYWORD3)

Offset

Register	Offset
MR1MASKEDKEYWORD3	14Ch

Function

Contains bits [127:96] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY1WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY1WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY1WORD3	Masked Key Word 3 Bits [127:96] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.18 Memory Region 1, Mask for Key Word 0 (MR1MASKFORKEYWORD0)

Offset

Register	Offset
MR1MASKFORKEYWORD0	150h

Function

Contains bits [31:0] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK1WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK1WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK1WORD0	Mask for Key Word 0 Bits [31:0] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.19 Memory Region 1, Mask for Key Word 1 (MR1MASKFORKEYWORD1)

Offset

Register	Offset
MR1MASKFORKEYWORD1	154h

Function

Contains bits [63:32] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK1WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK1WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK1WORD1	Mask for Key Word 1 Bits [63:32] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.20 Memory Region 1, Mask for Key Word 2 (MR1MASKFORKEYWORD2)

Offset

Register	Offset
MR1MASKFORKEYWORD2	158h

Function

Contains bits [95:64] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK1WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK1WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK1WORD2	Mask for Key Word 2 Bits [95:64] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.21 Memory Region 1, Mask for Key Word 3 (MR1MASKFORKEYWORD3)

Offset

Register	Offset
MR1MASKFORKEYWORD3	15Ch

Function

Contains bits [127:96] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK1WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK1WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK1WORD3	Mask for Key Word 3 Bits [127:96] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.22 Memory Region 1 Start Address (MR1STARTADDR)

Offset

Register	Offset
MR1STARTADDR	160h

Function

Contains the start address for this memory region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	SRTADDR1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R								0								
W	SRTADDR1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-9 SRTADDR1	Start Address for Memory Region 1 Start address for this memory region. Bits [8:0] of the start address are always taken as 0s.
8-0 —	This read-only field is reserved and always has the value 0.

12.6.1.23 Memory Region 1 End Address (MR1ENDADDR)

Offset

Register	Offset
MR1ENDADDR	164h

Function

Contains the end address and the valid indicator for this memory region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	ENDADDR1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	ENDADDR1							Reserved							V	
Reset	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1

Fields

Field	Function
31-9 ENDADDR1	End Address for Memory Region 1 End address for this memory region. Bits [8:0] of the end address are always taken as 1s.
8-3 —	This write-only field is reserved and always has the value 1.
2-0 V	Memory Region 1 is Valid Set by software (W2S, sticky); cleared by POR only (Power-On Reset). Attempted writes with values other than 010b do not change the register state. 010b - Memory Region 1 is valid. Subsequent reads return 111b. 101b - Memory Region 1 is not valid.

12.6.1.24 Memory Region 2, Masked Key Word 0 (MR2MASKEDKEYWORD0)

Offset

Register	Offset
MR2MASKEDKEYWORD0	180h

Function

Contains bits [31:0] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY2WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY2WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0	Masked Key Word 2

Table continues on the next page...

Field	Function
MASKEDKEY2 WORD0	Bits [31:0] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.25 Memory Region 2, Masked Key Word 1 (MR2MASKEDKEYWORD1)

Offset

Register	Offset
MR2MASKEDKEYWORD1	184h

Function

Contains bits [63:32] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY2WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY2WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY2 WORD1	Masked Key Word 1 Bits [63:32] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.26 Memory Region 2, Masked Key Word 2 (MR2MASKEDKEYWORD2)

Offset

Register	Offset
MR2MASKEDKEYWORD2	188h

Function

Contains bits [95:64] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY2WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY2WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY2WORD2	Masked Key Word 2 Bits [95:64] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.27 Memory Region 2, Masked Key Word 3 (MR2MASKEDKEYWORD3)

Offset

Register	Offset
MR2MASKEDKEYWORD3	18Ch

Function

Contains bits [127:96] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY2WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY2WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY2WORD3	Masked Key Word 3 Bits [127:96] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.28 Memory Region 2, Mask for Key Word 0 (MR2MASKFORKEYWORD0)

Offset

Register	Offset
MR2MASKFORKEYWORD0	190h

Function

Contains bits [31:0] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK2WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK2WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK2WORD0	Mask for Key Word 0 Bits [31:0] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.29 Memory Region 2, Mask for Key Word 1 (MR2MASKFORKEYWORD1)

Offset

Register	Offset
MR2MASKFORKEYWORD1	194h

Function

Contains bits [63:32] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK2WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK2WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK2WORD1	Mask for Key Word 1 Bits [63:32] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.30 Memory Region 2, Mask for Key Word 2 (MR2MASKFORKEYWORD2)

Offset

Register	Offset
MR2MASKFORKEYWORD2	198h

Function

Contains bits [95:64] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK2WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK2WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK2WORD2	Mask for Key Word 2 Bits [95:64] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.31 Memory Region 2, Mask for Key Word 3 (MR2MASKFORKEYWORD3)

Offset

Register	Offset
MR2MASKFORKEYWORD3	19Ch

Function

Contains bits [127:96] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK2WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK2WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK2WORD3	Mask for Key Word 3 Bits [127:96] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.32 Memory Region 2 Start Address (MR2STARTADDR)

Offset

Register	Offset
MR2STARTADDR	1A0h

Function

Contains the start address for this memory region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	SRTADDR2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R								0								
W	SRTADDR2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-9 SRTADDR2	Start Address for Memory Region 2 Start address for this memory region. Bits [8:0] of the start address are always taken as 0s.
8-0 —	This read-only field is reserved and always has the value 0.

12.6.1.33 Memory Region 2 End Address (MR2ENDADDR)

Offset

Register	Offset
MR2ENDADDR	1A4h

Function

Contains the end address and the valid indicator for this memory region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	ENDADDR2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	ENDADDR2							Reserved						V		
Reset	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1

Fields

Field	Function
31-9 ENDADDR2	End Address for Memory Region 2 End address for this memory region. Bits [8:0] of the end address are always taken as 1s.
8-3 —	This write-only field is reserved and always has the value 1.
2-0 V	Memory Region 2 is Valid

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Set by software (W2S, sticky); cleared by POR only (Power-On Reset). Attempted writes with values other than 010b do not change the register state. 010b - Memory Region 2 is valid. Subsequent reads return 111b. 101b - Memory Region 2 is not valid.

12.6.1.34 Memory Region 3, Masked Key Word 0 (MR3MASKEDKEYWORD0)

Offset

Register	Offset
MR3MASKEDKEYWORD0	1C0h

Function

Contains bits [31:0] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY3WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY3WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY3WORD0	Masked Key Word 3 Bits [31:0] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.35 Memory Region 3, Masked Key Word 1 (MR3MASKEDKEYWORD1)

Offset

Register	Offset
MR3MASKEDKEYWORD1	1C4h

Function

Contains bits [63:32] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY3WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY3WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY3WORD1	Masked Key Word 1 Bits [63:32] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.36 Memory Region 3, Masked Key Word 2 (MR3MASKEDKEYWORD2)

Offset

Register	Offset
MR3MASKEDKEYWORD2	1C8h

Function

Contains bits [95:64] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY3WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY3WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY3 WORD2	Masked Key Word 2 Bits [95:64] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.37 Memory Region 3, Masked Key Word 3 (MR3MASKEDKEYWORD3)

Offset

Register	Offset
MR3MASKEDKEYWORD3	1CCh

Function

Contains bits [127:96] of the 128-bit Masked Key used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASKEDKEY3WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASKEDKEY3WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASKEDKEY3WORD3	Masked Key Word 3 Bits [127:96] of the 128-bit Masked Key used to generate the PRINCE key for this region: • PRINCE key = (**Masked Key**) XOR (Key Mask)

12.6.1.38 Memory Region 3, Mask for Key Word 0 (MR3MASKFORKEYWORD0)

Offset

Register	Offset
MR3MASKFORKEYWORD0	1D0h

Function

Contains bits [31:0] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK3WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK3WORD0															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK3WORD0	Mask for Key Word 0 Bits [31:0] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.39 Memory Region 3, Mask for Key Word 1 (MR3MASKFORKEYWORD1)

Offset

Register	Offset
MR3MASKFORKEYWORD1	1D4h

Function

Contains bits [63:32] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK3WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK3WORD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK3WORD1	Mask for Key Word 1 Bits [63:32] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.40 Memory Region 3, Mask for Key Word 2 (MR3MASKFORKEYWORD2)

Offset

Register	Offset
MR3MASKFORKEYWORD2	1D8h

Function

Contains bits [95:64] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK3WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK3WORD2															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK3WORD2	Mask for Key Word 2 Bits [95:64] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.41 Memory Region 3, Mask for Key Word 3 (MR3MASKFORKEYWORD3)

Offset

Register	Offset
MR3MASKFORKEYWORD3	1DCh

Function

Contains bits [127:96] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	MASK3WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	MASK3WORD3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 MASK3WORD3	Mask for Key Word 3 Bits [127:96] of the 128-bit mask used to generate the PRINCE key for this region: • PRINCE key = (Masked Key) XOR (**Key Mask**)

12.6.1.42 Memory Region 3 Start Address (MR3STARTADDR)

Offset

Register	Offset
MR3STARTADDR	1E0h

Function

Contains the start address for this memory region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	SRTADDR3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R								0								
W	SRTADDR3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-9 SRTADDR3	Start Address for Memory Region 3 Start address for this memory region. Bits [8:0] of the start address are always taken as 0s.
8-0 —	This read-only field is reserved and always has the value 0.

12.6.1.43 Memory Region 3 End Address (MR3ENDADDR)

Offset

Register	Offset
MR3ENDADDR	1E4h

Function

Contains the end address and the valid indicator for this memory region.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	ENDADDR3															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	ENDADDR3							Reserved						V		
Reset	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1

Fields

Field	Function
31-9 ENDADDR3	End Address for Memory Region 3 End address for this memory region. Bits [8:0] of the end address are always taken as 1s.
8-3 —	This write-only field is reserved and always has the value 1.
2-0 V	Memory Region 3 is Valid Set by software (W2S, sticky); cleared by POR only (Power-On Reset). Attempted writes with values other than 010b do not change the register state. 010b - Memory Region 3 is valid. Subsequent reads return 111b. 101b - Memory Region 3 is not valid.

Chapter 13 Secure Miscellaneous System Control Module (SMSCM)

13.1 Chip-specific Secure Miscellaneous System Control Module information

Table 211. Reference links to related information[1]

Topic	Related module	References
Full description	Secure Miscellaneous System Control Module	Secure Miscellaneous System Control Module (SMSCM)
System memory map		See the section "System memory map"
CLocking		See the chapter "Clocking"
Signal multiplexing	Port control	See the chapter "Signal Muxing and Pinout "

1. For all the reference sections and chapters mentioned in this table, refer the Reference Manual.

13.1.1 Module instance

This device has one instance of the SMSCM module.

13.1.2 On-chip memory control

The SMSCM can control the following on-chip memory (OCRAM) with registers.

Table 212. On-chip memory (OCRAM) control

OCRAM	SRAM instance	ECC
OCRAM0	Program Flash	-
OCRAM2	ctcm0	Yes
OCRAM3	stcm0-stcm2	Yes
OCRAM5	stcm8	Yes
Others	Reserved	-

13.2 Overview

The Secure Miscellaneous System Control Module (SMSCM) contains secure CPU configuration registers and secure on-chip memory controller registers.

The SMSCM has two 32-bit registers to allow software to read the values of the LIFECYCLE and LIFECYCLE_B fuses. The SMSCM inputs these fuses, which can change during POR or warm reset. The fuse are stable as reset is negated. After reset is deasserted, if the LIFECYCLE and LIFECYCLE_B values are not complements, a security violation is asserted. The security violation is sent to the security subsystem.

13.3 SMSCM Memory Map/Register Definition

13.3.1 Replicated Control Registers

For certain critical control and configuration functions, a special three D flip-flop (DFF3) register implementation is used.

This design uses a pole, anti-pole implementation where the 101b state represents the negated register state and 010b is the asserted state. In the following register descriptions, the replicated function is shown as a single read data bit plus a 3-bit write data field. The register read data bit is aligned as the "middle bit" within the 3-bit field and the 3-bit write data field is defined as "W2S, W5C". This nomenclature defines a 3-bit write data field of value 2 (010b) is needed to set the asserted state of the three DFFs, while a 3-bit write data field of value 5 (101b) is required to clear the combined DFF3 state. All other write data values do not affect the register state. For reads, the 3-bit field returns {0,register_state,0}b, that is, the register state in the middle bit and zeroes for both adjacent bits.

To change the control state, the user must write BOTH pole and anti-pole registers. The order of the register writes, pole then anti-pole versus anti-pole then pole does not matter. The programming model is updated after the first register is written, however the control state in the hardware doesn't change until the second register is written.

The control state is asserted when the pole register is written with 010b and the anti-pole register is written with 101b. The control state is deasserted when the pole register is written with 101b and the antipole register is written with 010b. If both pole and anti-pole registers are written with the same legal value (010b or 101b), a security violation is asserted. The security violation remains asserted until system reset asserts.

13.3.2 SMSCM register descriptions

13.3.2.1 SMSCM memory map

Writes to the SMSCM programming model must be 32-bits. Non 32-bit writes are terminated with a bus error.

Reads of 8-bit, 16-bit and 32-bit size are allowed.

Writes to read-only registers and reads of write-only registers are terminated with a bus error.

The access permissions for this module are set in the TRDC.

SMSCM base address: 4001_5000h

Offset	Register	Width (In bits)	Access	Reset value
0h	Debug Enable (DBGEN)	32	RW	0000_0000h
4h	Debug Enable Complement (DBGEN_B)	32	RW	0022_2222h
8h	Debug Enable Lock (DBGEN_LOCK)	32	RW	0000_0000h
20h	Debug Authentication Beacon (DBG_AUTH_BEACON)	32	RW	See section
30h	Lifecycle Fuse Word (LIFECYCLE)	32	R	See section
34h	Lifecycle Fuse Word Complement (LIFECYCLE_B)	32	R	See section
40h	ROM Lockout Register (ROM_LOCKOUT)	32	RW	0000_0000h
100h	Security Counter Register (SCTR)	32	RW	0000_0000h
104h	Security Counter Plus 1 Register (SCTRP1)	32	W	0000_0000h
10Ch	Security Counter Minus 1 Register (SCTRM1)	32	W	0000_0000h
114h	Security Counter Plus X Register (SCTRPX)	32	W	0000_0000h
11Ch	Security Counter Minus X Register (SCTRMX)	32	W	0000_0000h
400h	On-Chip Memory Descriptor Register (OCMDR0)	32	RW	0000_0000h
408h	On-Chip Memory Descriptor Register (OCMDR2)	32	RW	0000_0003h

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
40Ch	On-Chip Memory Descriptor Register (OCMDR3)	32	RW	0000_0003h
414h	On-Chip Memory Descriptor Register (OCMDR5)	32	RW	0000_0003h
480h	On-Chip Memory ECC Control Register (OCMECR)	32	RW	0000_0000h
488h	On-Chip Memory ECC Interrupt Register (OCMEIR)	32	RW	0000_0000h
490h	On-Chip Memory Fault Address Register (OCMFAR)	32	R	0000_0000h
494h	On-Chip Memory Fault Attribute Register (OCMFTR)	32	R	0000_0000h
498h	On-Chip Memory ECC Fault Data High Register (OCMFDRH)	32	R	0000_0000h
49Ch	On-Chip Memory ECC Fault Data Low Register (OCMFDRL)	32	R	0000_0000h
C00h	Core Platform Control Register (CPCR)	32	RW	0000_0001h

13.3.2.2 Debug Enable (DBGEN)

Offset

Register	Offset
DBGEN	0h

Function

This register implements debug access control in six 3-bit pole, anti-pole states (DFF3) while the DBGEN_B register implements independent complement versions of the same debug access control states. This register only resets on POR. For warm resets, the value remains in the register and updated when the fuses are valid.

The DBGEN/SPIDEN/NIDEN/SPNIDEN register fields can be updated when DBGEN_LOCK[LOCK] = 000b. The ALTDBGEN register field can be updated when DBGEN_LOCK[ALT_DBGEN_LOCK] = 000b. The ALTEN register field can be updated when DBGEN_LOCK[ALT_EN_LOCK] = 000b. The register is fields are unlocked at reset.

Writes to locked register fields are simply ignored and are not error terminated. Therefore, software must read the DBGEN register after each write to ensure the write completed.

As power-on reset is negated and the fuse initialization is complete, the DBGEN and DBGEN_B are loaded with values determined by the encoded converged lifecycle fuses.

The pole/anti-pole updating sequence works independently for ALTEN, ALTDBGEN, SPNIDEN, NIDEN, SPIDEN, DBGEN bitfields.

For reads, each of the DFF3 3-bit fields return {0,register_state,0}b.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0									ALTEN			0	ALTDBGEN		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	SPNIDEN			0	NIDEN			0	SPIDEN			0	DBGEN		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-23 —	Reserved
22-20 ALTEN	Alternate Enable (DFF3 bitfield) This DFF3 control provides an alternate enable. This field can only be written when DBGEN_LOCK[ALT_EN_LOCK] = 000b NOTE This field behaves differently for register reads and writes. When reading 000b - Alternate Disabled. 010b - Alternate Enabled. When writing 010b - W2S - Enable Alternate. 101b - W5C - Disable Alternate.
19 —	Reserved
18-16 ALTDBGEN	Alternate Invasive Debug Enable (DFF3 bitfield) This DFF3 control provides the ability to enable invasive debug in an Alternate CPU. This field can only be written when DBGEN_LOCK[ALT_DBGEN_LOCK] = 000b NOTE This field behaves differently for register reads and writes. When reading

Table continues on the next page...

Table continued from the previous page...

Field	Function
	000b - Alternate Invasive Debug Disabled. 010b - Alternate Invasive Debug Enabled. When writing 010b - W2S - Enable Alternate Invasive Debug. 101b - W5C - Disable Alternate Invasive Debug.
15 —	Reserved
14-12 SPNIDEN	Secure Non-Invasive Debug Enable (DFF3 bitfield) This DFF3 control provides the ability to enable secure non-invasive debug. This field can only be written when DBGEN_LOCK[LOCK] = 000b NOTE This field behaves differently for register reads and writes. When reading 000b - Secure Non-Invasive Debug Disabled. 010b - Secure Non-Invasive Debug Enabled. When writing 010b - W2S - Enable Secure Non-Invasive Debug. 101b - W5C - Disable Secure Non-Invasive Debug.
11 —	Reserved
10-8 NIDEN	Non-Invasive Debug Enable (DFF3 bitfield) This DFF3 control provides the ability to enable non-invasive debug. This field can only be written when DBGEN_LOCK[LOCK] = 000b NOTE This field behaves differently for register reads and writes. When reading 000b - Non-Invasive Debug Disabled. 010b - Non-Invasive Debug Enabled. When writing 010b - W2S - Enable Non-Invasive Debug. 101b - W5C - Disable Non-Invasive Debug.
7	Reserved

Table continues on the next page...

Table continued from the previous page...

Field	Function
—	
6-4 SPIDEN	Secure Invasive Debug Enable (DFF3 bitfield) This DFF3 control provides the ability to enable secure invasive debug. This field can only be written when DBGEN_LOCK[LOCK] = 000b NOTE This field behaves differently for register reads and writes. When reading 000b - Secure Invasive Debug Disabled. 010b - Secure Invasive Debug Enabled. When writing 010b - W2S - Enable Secure Invasive Debug. 101b - W5C - Disable Secure Invasive Debug.
3 —	Reserved
2-0 DBGEN	Invasive Debug Enable (DFF3 bitfield) This DFF3 control provides the ability to enable invasive debug. This field can only be written when DBGEN_LOCK[LOCK] = 000b NOTE This field behaves differently for register reads and writes. When reading 000b - Invasive Debug Disabled. 010b - Invasive Debug Enabled. When writing 010b - W2S - Enable Invasive Debug. 101b - W5C - Disable Invasive Debug.

13.3.2.3 Debug Enable Complement (DBGEN_B)

Offset

Register	Offset
DBGEN_B	4h

Function

This register implements debug access control in six 3-bit pole, anti-pole states (DFF3) while the DBGEN_B register implements independent complement versions of the same debug access control states. This register only resets on POR. For warm resets, the value remains in the register and updated when the fuses are valid.

The DBGEN_B/SPIDEN_B/NIDEN_B/SPNIDEN_B register fields can be updated when DBGEN_LOCK[LOCK] = 000b. The ALTDBGEN_B register field can be updated when DBGEN_LOCK[ALT_DBGEN_LOCK] = 000b. The ALTEN_B register field can be updated when DBGEN_LOCK[ALT_EN_LOCK] = 000b. The register is fields are unlocked at reset.

Writes to locked register fields are simply ignored and are not error terminated. Therefore, software must read the DBGEN_B register after each write to ensure the write completed.

As power-on reset is negated and the fuse initialization is complete, the DBGEN and DBGEN_B are loaded with values determined by the encoded converged lifecycle fuses.

The pole/anti-pole updating sequence works independently for ALTDBGEN, SPNIDEN, NIDEN, SPIDEN, DBGEN bitfields.

For reads, each of the DFF3 3-bit fields return {0,register_state,0}b.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0									ALTEN_B			0	ALTDBGEN_B		
W																
Reset	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	SPNIDEN_B			0	NIDEN_B			0	SPIDEN_B			0	DBGEN_B		
W																
Reset	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0

Fields

Field	Function
31-23 —	Reserved
22-20 ALTEN_B	Alternate Enable Complement (DFF3 bitfield) This DFF3 control provides the complement version of ALT_EN. This field can only be written when DBGEN_LOCK[ALT_EN_LOCK] = 000b NOTE This field behaves differently for register reads and writes. When reading 000b - Alternrate Enabled. 010b - Alternate Disabled. When writing

Table continues on the next page...

Table continued from the previous page...

Field	Function
	010b - W2S - Disable Alternate. 101b - W5C - Enable Alternate.
19 —	Reserved
18-16 ALTDBGEN_B	Alternate Invasive Debug Enable Complement (DFF3 bitfield) This DFF3 control provides the complement version of ALT_DBGEN. This field can only be written when DBGEN_LOCK[ALT_DBGEN_LOCK] = 000b NOTE This field behaves differently for register reads and writes. When reading 000b - Alternate Invasive Debug Enabled. 010b - Alternate Invasive Debug Disabled. When writing 010b - W2S - Alternate Disable Invasive Debug. 101b - W5C - Alternate Enable Invasive Debug.
15 —	Reserved
14-12 SPNIDEN_B	Secure Non-Invasive Debug Enable Complement (DFF3 bitfield) This DFF3 control provides the complement version of SPNIDEN. NOTE This field behaves differently for register reads and writes. When reading 000b - Secure Non-Invasive Debug Enabled. 010b - Secure Non-Invasive Debug Disabled. When writing 010b - W2S - Disable Secure Non-Invasive Debug. 101b - W5C - Enable Secure Non-Invasive Debug.
11 —	Reserved
10-8 NIDEN_B	Non-Invasive Debug Enable Complement (DFF3 bitfield) This DFF3 control provides the complement version of NIDEN.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	This field can only be written when DBGEN_LOCK[LOCK] = 000b NOTE This field behaves differently for register reads and writes. When reading 000b - Non-Invasive Debug Enabled. 010b - Non-Invasive Debug Disabled. When writing 010b - W2S - Disable Non-Invasive Debug. 101b - W5C - Enable Non-Invasive Debug.
7 —	Reserved
6-4 SPIDEN_B	Secure Invasive Debug Enable - Complement (DFF3 bitfield) This DFF3 control provides the complement version of SPIDEN. This field can only be written when DBGEN_LOCK[LOCK] = 000b NOTE This field behaves differently for register reads and writes. When reading 000b - Secure Invasive Debug Enabled. 010b - Secure Invasive Debug Disabled. When writing 010b - W2S - Disable Secure Invasive Debug. 101b - W5C - Enable Secure Invasive Debug.
3 —	Reserved
2-0 DBGEN_B	Invasive Debug Enable Complement (DFF3 bitfield) This DFF3 control provides the complement version of DBGEN. This field can only be written when DBGEN_LOCK[LOCK] = 000b NOTE This field behaves differently for register reads and writes. When reading 000b - Invasive Debug Enabled.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	010b - Invasive Debug Disabled. When writing 010b - W2S - Disable Invasive Debug. 101b - W5C - Enable Invasive Debug.

13.3.2.4 Debug Enable Lock (DBGEN_LOCK)

Offset

Register	Offset
DBGEN_LOCK	8h

Function

DBG_EN_LOCK fuse is used by ROM to control whether this lock is set prior to executing customer code.

Writes to locked register fields are simply ignored and are not error terminated. Therefore, software must read the DBGEN_LOCK register after each write to ensure the write completed.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0									ALT_EN_LOCK			0	ALT_DBGEN_LOCK		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0													LOCK		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-23 —	Reserved
22-20 ALT_EN_LOCK	Alternate Lock (DFF3 bitfield) This DFF3 control provides the ability to prevent write accesses to the DBGEN[ALT_EN], DBGEN[ALT_EN_B] and DBGEN_LOCK[ALT_EN_LOCK] registers.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	NOTE This field behaves differently for register reads and writes. When reading 000b - ALTEN, ALTEN_B, ALT_EN_LOCK unlocked. 010b - ALTEN, ALTEN_B, ALT_EN_LOCK locked. ALTEN, ALTEN_B, ALT_EN_LOCK locked and cannot be written until the next system reset. They remain readable. When writing 010b,000b,001b,011b,100b,111b - Lock DBGEN[ALTEN], DBGEN_B[ALTEN_B, and DBGEN_LOCK[ALT_EN_LOCK]. Writing any value other than 101b locks the DBGEN[ALTEN], DBGEN_B[ALTEN_B], and DBGEN_LOCK[ALT_EN_LOCK] fields. 101b - f When ALT_EN_LOCK is locked, ALT_EN_LOCK cannot be unlocked with a write of 101b to this field. When ALT_EN_LOCK is unlocked, a write of 101b to this field, ALT_EN_LOCK remains unlocked and ALTEN/ALTEN_B remains writeable.
19 —	Reserved
18-16 ALT_DBGEN_LOCK	Alternate Lock (DFF3 bitfield) This DFF3 control provides the ability to prevent write accesses to the DBGEN[ALT_DBGEN], DBGEN[ALT_DBGEN_B] and DBGEN_LOCK[ALT_DBGEN_LOCK] registers. NOTE This field behaves differently for register reads and writes. When reading 000b - ALT_DBGEN, ALT_DBGEN_B, ALT_DBGEN_LOCK unlocked. 010b - ALT_DBGEN, ALT_DBGEN_B, ALT_DBGEN_LOCK locked. ALT_DBGEN, ALT_DBGEN_B, ALT_DBGEN_LOCK locked and cannot be written until the next system reset. They remain readable. When writing 010b,000b,001b,011b,100b,111b - Lock DBGEN[ALTDBGEN], DBGEN_B[ALTDBGEN_B, and DBGEN_LOCK[ALT_DBGEN_LOCK]. Writing any value other than 101b locks the DBGEN[ALTDBGEN], DBGEN_B[ALTDBGEN_B], and DBGEN_LOCK[ALT_DBGEN_LOCK] fields. 101b - When ALT_DBGEN_LOCK is locked, ALT_DBGEN_LOCK cannot be unlocked with a write of 101b to this field. When ALT_DBGEN_LOCK is unlocked, a write of 101b to this field, ALT_DBGEN_LOCK remains unlocked and DBGEN/DBGEN_B remains writeable.
15-3 —	Reserved
2-0	Lock (DFF3 bitfield)

Table continues on the next page...

Table continued from the previous page...

Field	Function
LOCK	This DFF3 control provides the ability to prevent write accesses to the DBGEN[DBGEN, SPIDEN, NIDEN, SPNIDEN], DBGEN_B[DBGEN_B, SPIDEN_B, NIDEN_B, SPNIDEN_B] and DBGEN_LOCK[LOCK] register fields. NOTE This field behaves differently for register reads and writes. When reading 000b - DBGEN[SPNIDEN,NIDEN,SPIDEN,DBGEN], DBGEN_B[SPNIDEN_B,NIDEN_B,SPIDEN_B,DBGEN_B], and DBGEN_LOCK[LOCK] unlocked. 010b - DBGEN[SPNIDEN,NIDEN,SPIDEN,DBGEN], DBGEN_B[SPNIDEN_B,NIDEN_B,SPIDEN_B,DBGEN_B], and DBGEN_LOCK[LOCK] locked. DBGEN[SPNIDEN,NIDEN,SPIDEN,DBGEN], DBGEN_B[SPNIDEN_B,NIDEN_B,SPIDEN_B,DBGEN_B], and DBGEN_LOCK[LOCK] are locked and cannot be written until the next system reset. They remain readable. When writing 010b,000b,001b,011b,100b,111b - Lock DBGEN[SPNIDEN,NIDEN,SPIDEN,DBGEN], DBGEN_B[SPNIDEN_B,NIDEN_B,SPIDEN_B,DBGEN_B], and DBGEN_LOCK[LOCK]. Writing any value other than 101b locks DBGEN[SPNIDEN,NIDEN,SPIDEN,DBGEN], DBGEN_B[SPNIDEN_B,NIDEN_B,SPIDEN_B,DBGEN_B], and DBGEN_LOCK[LOCK]. 101b - When DBGEN_LOCK[LOCK] is locked, DBGEN_LOCK[LOCK] cannot be unlocked with a write of 101b to this field. When DBGEN_LOCK[LOCK] is unlocked, a write of 101b to this field, DBGEN_LOCK[LOCK] remains unlocked and the DBGEN[DBGEN, SPIDEN, NIDEN, SPNIDEN],DBGEN_B[DBGEN_B, SPIDEN_B, NIDEN_B, SPNIDEN_B] fields remain writeable.

13.3.2.5 Debug Authentication Beacon (DBG_AUTH_BEACON)

Offset

Register	Offset
DBG_AUTH_BEACON	20h

Function

This register only resets on POR or the domain's cold reset. It is not locked by DBGEN_LOCK.

It does not control any logic, written by debug authentication and used only by customer code.

NOTE

The reset value of this register is user defined.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	CREDENTIAL_BEACON															
Reset	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	AUTH_BEACON															
Reset	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u

Fields

Field	Function
31-16 CREDENTIAL_BEACON	Credential Beacon
15-0 AUTH_BEACON	Authentication Beacon

13.3.2.6 Lifecycle Fuse Word (LIFECYCLE)

Offset

Register	Offset
LIFECYCLE	30h

Function

The LIFECYCLE register is cleared with POR and loaded directly from the fuses when the fuses are ready.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	SWD_ID				OEM_SEC_BOOT_EN				0				CTRK_REVOKE			
W																
Reset	u	u	u	u	0	0	0	0	0	0	0	0	u	u	u	u

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	WAKEUP_...	SERIAL_...	0	NBU_SEC...	0	TZM_EN	DBG_AUT...	DBG_EN_...	CLC							
W																
Reset	u	u	0	0	0	u	u	u	0	0	0	0	0	0	0	0

Fields

Field	Function
31-28 SWD_ID	Serial Wire Debug Instance ID
27-24 OEM_SEC_BOOT_EN	Main Core Secure Boot Enable 0000b - Disabled, Boot image doesn't need to be authenticated to boot (default). 1010b - Enabled, Boot image must be authenticated to boot. All other values - Reserved
23-20 —	Reserved
19-16 CTRK_REVOKE	Revocation indicator from OEM Firmware Authentication Public Key
15 WAKEUP_DIS	Wakeup Disabled 0b - Boot-ROM LP wakup is enabled. 1b - Boot-ROM LP wakup is disabled.
14 SERIAL_DIS	Serial Download Disabled 0b - Serial download path is enabled. 1b - Serial download path is disabled.
13 —	Reserved
12	NBU Secure Boot Enable 0b - Disabled, NBU image doesn't need to be authenticated (default).

Table continues on the next page...

Table continued from the previous page...

Field	Function
NBU_SEC_BOOT_EN	1b - Enabled, NBU image must be authenticated to boot.
11 —	Reserved
10 TZM_EN	Trust Zone Mode Enable 0b - TZ-M is disabled by default, can be enabled by software. 1b - TZ-M is enabled.
9 DBG_AUTH_DIS	Debug Authentication Disabled 0b - Debug Authentication enabled. 1b - Debug Authentication disabled.
8 DBG_EN_LOCK	Debug Enable Lock 0b - The debug access control registers remain open when jumping to customer code. 1b - The debug access control registers are write-locked before jumping to customer code.
7-0 CLC	Converged Lifecycle 0000_0000b - BLANK 0000_0001b - NXP Fab 0000_0011b - NXP Provisioned 0000_0111b - OEM Open 0000_1111b - OEM Secure World Closed 0001_1111b - OEM Closed 0011_1111b - OEM Return 0111_1111b - NXP Return 1001_1111b - OEM Locked 11xx_xxxxb - BRICK

13.3.2.7 Lifecycle Fuse Word Complement (LIFECYCLE_B)

Offset

Register	Offset
LIFECYCLE_B	34h

Function

The LIFECYCLE_B register is cleared with POR and loaded directly from the fuses when the fuses are ready.

This register is the complement of the LIFECYCLE register.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	SWD_ID_B				OEM_SEC_BOOT_EN_B				0				CTRK_REVOKE_B			
W																
Reset	u	u	u	u	0	0	0	0	0	0	0	0	u	u	u	u

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	WAKEUP_...	SERIAL_...	0	NBU_SEC...	0	TZM_EN_B	DBG_AUT...	DBG_EN_...	CLC_B							
W																
Reset	u	u	0	0	0	u	u	u	0	0	0	0	0	0	0	0

Fields

Field	Function
31-28 SWD_ID_B	Serial Wire Debug Instance ID Complement
27-24 OEM_SEC_BOOT_EN_B	Main Core Secure Boot Enable Complement 0101b - Enabled, Boot image must be authenticated to boot. 1111b - Disabled, Boot image doesn't need to be authenticated to boot (default). All other values - Reserved
23-20 —	Reserved
19-16 CTRK_REVOKE_B	Revocation indicator from OEM Firmware Authentication Public Key Complement
15 WAKEUP_DIS_B	Wakeup Disabled Complement 0b - Boot-ROM LP wakup is disabled. 1b - Boot-ROM LP wakup is enabled.
14 SERIAL_DIS_B	Serial Download Disabled Complement 0b - Serial download path is disabled. 1b - Serial download path is enabled.
13 —	Reserved
12	NBU Secure Boot Enable Complement 0b - Enabled, NBU image must be authenticated to boot.

Table continues on the next page...

Table continued from the previous page...

Field	Function
NBU_SEC_BOOT_EN_B	1b - Disabled, NBU image doesn't need to be authenticated (default).
11 —	Reserved
10 TZM_EN_B	Trust Zone Mode Enable Complement 0b - TZ-M is enabled. 1b - TZ-M is disabled by default, can be enabled by software.
9 DBG_AUTH_DIS_B	Debug Authentication Disabled Complement 0b - Debug Authentication disabled. 1b - Debug Authentication enabled.
8 DBG_EN_LOCK_B	Debug Enable Lock Complement 0b - The debug access control registers are write-locked before jumping to customer code. 1b - The debug access control registers remain open when jumping to customer code.
7-0 CLC_B	Converged Lifecycle Complement 00xx_xxxxb - BRICK 0110_0000b - OEM Locked 1000_0000b - NXP Return 1100_0000b - OEM Return 1110_0000b - OEM Closed 1111_0000b - OEM Secure World Closed 1111_1000b - OEM Open 1111_1100b - NXP Provisioned 1111_1110b - NXP Fab 1111_1111b - BLANK

13.3.2.8 ROM Lockout Register (ROM_LOCKOUT)

Offset

Register	Offset
ROM_LOCKOUT	40h

Function

This register provides the ability to set a watermark address in the ROM. Once enabled, accesses below the watermark address are not allowed and result in a bus error.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	REGLOCK			0							ROMWA					
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	ROMWA												0			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-29 REGLOCK	ROM_LOCKOUT Register Lock (DFF3 bitfield) This DFF3 control provides the ability to prevent write accesses to the ROM_LOCKOUT register and enable the ROM watermark feature. **NOTE** This field behaves differently for register reads and writes. When reading 000b - ROM_LOCKOUT unlocked. ROM_LOCKOUT is writable, and ROM array is fully accessible. 010b - ROM_LOCKOUT locked. ROM_LOCKOUT register cannot be written until next reset. Accesses to addresses below the ROM_LOCKOUT[ROMWA] result in an error response. When writing 010b,000b,001b,011b,100b,111b - Lock ROM_LOCKOUT register. Writing any value other than 101b locks the ROM_LOCKOUT register. 101b - Writing this value has no effect.
28-22 —	Reserved
21-4 ROMWA	ROM Watermark Address The first accessible address (0-mod-16) after the lockout region. When ROM_LOCKOUT[REGLOCK] = 010b, reads to addresses below this watermark value result in an error response.
3-0 —	Reserved

13.3.2.9 Security Counter Register (SCTR)

Offset

Register	Offset
SCTR	100h

Function

This register provides a security counter function, which is intended to validate the control flow integrity of CPU code execution. This register is the actual counter register; there are four additional virtual registers that support a set of arithmetic operations including {+1, -1, +x, -x} where x is any 32-bit operand.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	DATA32															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	DATA32															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 DATA32	Data, 32 bits This is the 32-bit value contained as the security counter.

13.3.2.10 Security Counter Plus 1 Register (SCTRP1)

Offset

Register	Offset
SCTRP1	104h

Function

This virtual write-only register provides a security counter arithmetic function - register writes to this address increment the security counter by 1 (+1). The security counter provides a mechanism to validate the control flow integrity of CPU code execution. This is one of four virtual registers that support a set of arithmetic operations including {+1, -1, +x, -x} where x is any 32-bit operand.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	DONTCARE32															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	DONTCARE32															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 DONTCARE32	Don't Care Data, 32 bits The entire contents of the write data word are ignored as register writes to this location increment the security counter by 1, that, is, next-state SCTR = current-state SCTR + 1.

13.3.2.11 Security Counter Minus 1 Register (SCTRM1)

Offset

Register	Offset
SCTRM1	10Ch

Function

This virtual write-only register provides a security counter arithmetic function - register writes to this address decrement the security counter by 1 (-1). The security counter provides a mechanism to validate the control flow integrity of CPU code execution. This is one of four virtual registers that support a set of arithmetic operations including {+1, -1, +x, -x} where x is any 32-bit operand.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	DONTCARE32															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	DONTCARE32															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 DONTCARE32	Don't Care Data, 32 bits The entire contents of the write data word are ignored as register writes to this location decrement the security counter by 1, that, is, next-state SCTR = current-state SCTR - 1.

13.3.2.12 Security Counter Plus X Register (SCTRPX)

Offset

Register	Offset
SCTRPX	114h

Function

This virtual write-only register provides a security counter arithmetic function - register writes to this address increment the security counter by "x" (+x). The security counter provides a mechanism to validate the control flow integrity of CPU code execution. This is one of four virtual registers that support a set of arithmetic operations including {+1, -1, +x, -x} where x is any 32-bit operand.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	DATA32															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	DATA32															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 DATA32	Data, 32 bits The entire contents of the write data word are added to the security counter, that, is, next-state SCTR = current-state SCTR + DATA32.

13.3.2.13 Security Counter Minus X Register (SCTRMX)

Offset

Register	Offset
SCTRMX	11Ch

Function

This virtual write-only register provides a security counter arithmetic function - register writes to this address decrement the security counter by "x" (-x). The security counter provides a mechanism to validate the control flow integrity of CPU code execution. This is one of four virtual registers that support a set of arithmetic operations including {+1, -1, +x, -x} where x is any 32-bit operand.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	DATA32															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R																
W	DATA32															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 DATA32	Data, 32 bits The entire contents of the write data word are subtracted to the security counter, that, is, next-state SCTR = current-state SCTR - DATA32.

13.3.2.14 On-Chip Memory Descriptor Register (OCMDR0)

Offset

Register	Offset
OCMDR0	400h

Function

This section of the programming model is an array of 32-bit generic on-chip memory descriptor registers that provides configurable controls (where appropriate).

- Any access with a size other than 32 bits are terminated with an error.

An on-chip memory has a corresponding SMSCM OCMDRn register when one of the following is true:

- The memory type is data flash
- The memory type is program flash
- The memory type is system RAM and ECC = true

The following table describes each on-chip memory type:

On-chip memory n	On-Chip Memory Type
On-chip memory 0	Program Flash
On-chip memory 2	System RAM
On-chip memory 3	System RAM
On-chip memory 5	System RAM

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															RO
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0				OCMCF2				OCMCF1				OCMCF0			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-17 —	Reserved
16 RO	Read-Only This register bit provides a mechanism to “lock” the configuration state defined by OCMDRn[11:0]. Once asserted, attempted writes to the OCMDRn[11:0] register are ignored until the next reset clears the flag. 0b - Writes to the OCMDRn[11:0] are allowed 1b - Writes to the OCMDRn[11:0] are ignored
15-12 —	Reserved
11-8	OCMEM Control Field 2

Table continues on the next page...

Table continued from the previous page...

Field	Function
OCMCF2	The flash controller needs to be idle when writing to an OCMDRn register associated with Flash memory. Changing controller configuration while active can cause undesired results. The flash cache on this device is a small, read-only cache to help accelerate flash accesses. The 4 bits in CF2 configure and maintain this cache. CF2 = Control Field 2 - for flash cache • CF2[3] - disable flash cache • CF2[2] - disable instruction caching • CF2[1] - disable flash data caching • CF2[0] - clear flash cache CF2[3:1] • 000b - the flash cache is enabled and will cache both flash instruction and flash data fetches • 001b - the flash cache is enabled and will cache flash instruction fetches but not flash data fetches • 010b - the flash cache is enabled and will cache flash data fetches but not flash instruction fetches • 011b - the flash cache is enabled but will not be used (basically disabled) • 1xxb - the flash cache is diabled CF2[0] • 0b - do not clear the flash cache • 1b - clear the flash cache (negate all valid bits in the cache)
7-4 OCMCF1	OCMEM Control Field 1 The flash controller needs to be idle when writing to an OCMDRn register associated with Flash memory. This means no read/erase/execute/etc operations from the Flash memory should be made while writing to the OCMDRn register associated with that memory. Changing controller configuration while active can cause undesired results. CF1 = Control Field 1 - for flash controller CF1[3] LKIFR1 - Lock IFR1 • 0b = IFR1 space can be accessed. • 1b = IFR1 space cannot be accessed. This bit is sticky. Once set, it is cleared with reset. CF1[2] Reserved CF1[1] DFS - Disable Flash Speculate • 0b = Enable Flash Speculate • 1b = Disable Flash Speculate CF1[0] DDP - Disable Data Prefetch • 0b = Enable Data Prefetch • 1b = Disable Data Prefetch

Table continues on the next page...

Table continued from the previous page...

<table>
<tr><th>Field</th><th>Function</th></tr>
<tr><td></td><td>
<p>CF1[1:0] bits controls whether prefetches (or speculative accesses) are initiated in response to instruction fetches or data references.</p>
<p>On new flash accesses, the flash controller reads 128-bit aligned data and stores it in a normal read buffer while bypassing the desired 32-bit to the input AHB bus. Subsequent accesses to the same 128-bits of flash data can be serviced from this buffer without re-reading the flash arrays and has buffers to hold a flash access. The flash controller also has a 128-bit speculation buffer. During idle cycles or cycles where the flash controller accesses are being serviced from the flash cache or the normal buffer, the flash controller will read the next sequential 128-bits of flash data and store it in the speculation buffer. Then, subsequent sequential accesses can hit this speculation buffer without accessing the flash. The following CF1[1:0] bits configure and maintain this speculation buffer.</p>
<table>
<tr><th>CF1[1]
Flash Speculate Disable</th><th>CF1[0]
Data Prefetch Disable</th><th>Result</th></tr>
<tr><td>Disabled (1b)</td><td>Disabled (1b)</td><td rowspan="2">All speculation disabled and speculation buffer is cleared</td></tr>
<tr><td>Disabled (1b)</td><td>Enabled (0b)</td></tr>
<tr><td>Enabled (0b)</td><td>Disabled (1b)</td><td>Speculation for Instruction enabled and Speculation for Data disabled</td></tr>
<tr><td>Enabled (0b)</td><td>Enabled (0b)</td><td>Speculation for both Instruction and Data enabled</td></tr>
</table>
<p>The reset state for CF1[1:0] is 00b - the speculation buffer is enabled and speculation will occur for both flash instruction accesses and flash data accesses.</p>
<p>Note, whenever the speculation buffer is disabled (CF1[1:0] = 1xb, the speculation buffer is cleared (its valid bit is negated). This provides a way to clear the speculation buffer. The speculation buffer is also cleared when the flash controller is reset.</p>
</td></tr>
<tr><td>3-0
OCMCF0</td><td>
<p>OCMEM Control Field 0</p>
<p><b>OCMCF0 is used to Program or Data Flash and System RAM ECC.</b></p>
<p>The flash controller needs to be idle when writing to an OCMDRn register associated with Flash memory. This means no read/erase/execute/etc operations from the Flash memory should be made while writing to the OCMDRn register associated with that memory. Changing controller configuration while active can cause undesired results.</p>
<p>CF0 = Control Field 0 - for ECC control functions.</p>
<p>CF0[3] DNCBED - Disable non-correctable bus errors on flash data fetches</p>
<ul>
<li>0b = Bus error for non-correctable error on data fetch from flash</li>
<li>1b = Disable bus error response for non-correctable error on data fetch from flash</li>
</ul>
<p>CF0[2] DNCBEI - Disable non-correctable bus errors on flash instruction fetches</p>
<ul>
<li>0b = Bus error for non-correctable error on instruction fetch from flash</li>
</ul>
</td></tr>
</table>

Table continues on the next page...

Table continued from the previous page...

Field	Function
	• 1b = Disable bus error response for non-correctable error on instruction fetch from flash CF0[1:0] - Reserved

13.3.2.15 On-Chip Memory Descriptor Register (OCMDR2 - OCMDR5)

Offset

Register	Offset
OCMDR2	408h
OCMDR3	40Ch
OCMDR5	414h

Function

This section of the programming model is an array of 32-bit generic on-chip memory descriptor registers that provides configurable controls (where appropriate).

- Any access with a size other than 32 bits are terminated with an error.

An on-chip memory has a corresponding SMSCM OCMDRn register when one of the following is true:

- The memory type is data flash
- The memory type is program flash
- The memory type is system RAM and ECC = true

The following table describes each on-chip memory type:

On-chip memory n	On-Chip Memory Type
On-chip memory 0	Program Flash
On-chip memory 2	System RAM
On-chip memory 3	System RAM
On-chip memory 5	System RAM

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															RO
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0				0				0				OCMCF0			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Fields

Field	Function
31-17 —	Reserved
16 RO	Read-Only This register bit provides a mechanism to “lock” the configuration state defined by OCMDRn[11:0]. Once asserted, attempted writes to the OCMDRn[11:0] register are ignored until the next reset clears the flag. 0b - Writes to the OCMDRn[11:0] are allowed 1b - Writes to the OCMDRn[11:0] are ignored
15-12 —	Reserved
11-8 —	Reserved
7-4 —	Reserved
3-0 OCMCF0	OCMEM Control Field 0 **OCMCF0 is used Program or Data Flash and System RAM ECC.** The flash controller needs to be idle when writing to an OCMDRn register associated with Flash memory. This means no read/erase/execute/etc operations from the Flash memory should be made while writing to the OCMDRn register associated with that memory. Changing controller configuration while active can cause undesired results. CF0 = Control Field 0 - for ECC control functions. CF0[3:2] - Reserved CF0[1] EERC - Enable ECC read check

Table continues on the next page...

Table continued from the previous page...

Field	Function
	• 0b = Disable ECC on reads • 1b = Enable ECC on reads CF0[0] EERW - Enable ECC write generation • 0b = Disable ECC on writes • 1b = Enable ECC on writes

13.3.2.16 On-Chip Memory ECC Control Register (OCMECR)

Offset

Register	Offset
OCMECR	480h

Function

The On-Chip Memory ECC Control Register is an 32-bit control register for specifying which types of memory errors are reported. In systems with ECC, the occurrence of a non-correctable error causes the current access to be terminated with an error condition. In many cases, this error termination is reported directly by the initiating bus master. However, there are certain situations where the occurrence of this type of non-correctable error is *not* reported by the master. Examples include speculative instruction fetches which are discarded due to a change-of-flow operation, and buffered operand writes. The ECC reporting logic in MSCM provides an optional error interrupt mechanism to signal all non-correctable memory errors. In addition to the interrupt generation, the MSCM captures specific information (memory address, attributes and data, bus master number, etc.) which may be useful for subsequent failure analysis.

Single bit memory corrections are performed on-the-fly, returning the corrected read data to the requesting bus master. For these single bit corrections, another configuration bit allows reporting of these events.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0							E1BR	0							ENCR
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-9 —	Reserved
8 E1BR	Enable RAM ECC 1 Bit Reporting 0b - 1-bit reporting disabled 1b - 1-bit reporting enabled
7-1 —	Reserved
0 ENCR	Enable RAM ECC Non-correctable Reporting 0b - Non-correctable reporting disabled 1b - Non-correctable reporting enabled

13.3.2.17 On-Chip Memory ECC Interrupt Register (OCMEIR)

Offset

Register	Offset
OCMEIR	488h

Function

The On-Chip Memory ECC Interrupt Register is a 32-bit control register for signaling which types of properly-enabled ECC events have occurred. The OCMEIR includes two bit maps to record the occurrence of individual ECC events, both 1-bit correctable and multi-bit non-correctable events.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	VALID	0			EELOC				0							
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	E1BERRN								ENCERRN							
W	W1C								W1C							
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31 VALID	Valid ECC Error Location field The VALID bit is set whenever a bit in [15:0] is set. When all of the errors are cleared, the VALID bit is cleared. 0b - ECC Error Location field is not valid 1b - ECC Error Location field is valid
30-28 —	Reserved
27-24 EELOC	ECC Error Location 0000b - non-correctable on OCRAM0 0001b - non-correctable on OCRAM1 0010b - non-correctable on OCRAM2 0011b - non-correctable on OCRAM3 0100b - non-correctable on OCRAM4 0101b - non-correctable on OCRAM5 0110b - non-correctable on OCRAM6 0111b - non-correctable on OCRAM7 1000b - 1-bit correctable on OCRAM0 1001b - 1-bit correctable on OCRAM1 1010b - 1-bit correctable on OCRAM2 1011b - 1-bit correctable on OCRAM3 1100b - 1-bit correctable on OCRAM4 1101b - 1-bit correctable on OCRAM5 1110b - 1-bit correctable on OCRAM6 1111b - 1-bit correctable on OCRAM7
23-16 —	Reserved
15-8 E1BERRN	ECC 1-bit Error OCRAMn This field is the bit map of 1-bit correctable errors. NOTE An E1BERRN[n] bit is set when a non-correctable error has occured, the corresponding OCMDRn[1:0] = 11b, and the corresponding OCRAMn has the ECC feature. • E1BERRN[0] - OCRAM0 has a 1-bit correctable error

Table continues on the next page...

Table continued from the previous page...

Field	Function
	- E1BERRN[1] - OCRAM1 has a 1-bit correctable error - E1BERRN[2] - OCRAM2 has a 1-bit correctable error - E1BERRN[3] - OCRAM3 has a 1-bit correctable error - E1BERRN[4] - OCRAM4 has a 1-bit correctable error - E1BERRN[5] - OCRAM5 has a 1-bit correctable error - E1BERRN[6] - OCRAM6 has a 1-bit correctable error - E1BERRN[7] - OCRAM7 has a 1-bit correctable error W1C - Write-1-Clear individually clear each 1-bit correctable error.
7-0 ENCERRN	ECC Non-correctable Error OCRAMn This field is the bit map of non-correctable errors. NOTE An ENCERRN[n] bit is set when a non-correctable error has occured, the corresponding OCMDRn[1:0] = 11b, and the corresponding OCRAMn has the ECC feature. - ENCERRN[0] - OCRAM0 has a non-correctable error - ENCERRN[1] - OCRAM1 has a non-correctable error - ENCERRN[2] - OCRAM2 has a non-correctable error - ENCERRN[3] - OCRAM3 has a non-correctable error - ENCERRN[4] - OCRAM4 has a non-correctable error - ENCERRN[5] - OCRAM5 has a non-correctable error - ENCERRN[6] - OCRAM6 has a non-correctable error - ENCERRN[7] - OCRAM7 has a non-correctable error W1C - Write-1-Clear individually clear each non-correctable error.

13.3.2.18 On-Chip Memory Fault Address Register (OCMFAR)

Offset

Register	Offset
OCMFAR	490h

Function

The OCMFAR is a 32-bit read-only register for capturing the address of the last, properly-enabled ECC event in the on-chip memory. Depending on the state of the On-Chip Memory ECC Control Register, an ECC event in the on-chip memory causes the address, attributes and read data associated with the access to be loaded into the OCMFAR, OCMFTR and OCMFDR registers, and the appropriate flag in the On-Chip Memory ECC Interrupt Register to be asserted. Attempted writes are ignored.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	EFADD															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	EFADD															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 EFADD	ECC Fault Address

13.3.2.19 On-Chip Memory Fault Attribute Register (OCMFTR)

Offset

Register	Offset
OCMFTR	494h

Function

The OCMFTR is a 32-bit read-only register for capturing the attributes of the last, properly-enabled ECC event in the on-chip memory. Depending on the state of the On-Chip Memory ECC Control Register, an ECC event in the on-chip memory causes the address, attributes and read data associated with the access to be loaded into the OCMFAR, OCMFTR and OCMFDR registers, and the appropriate flag in the On-Chip Memory ECC Interrupt Register to be asserted. Attempted writes are ignored.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0								EFSYN							
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	EFMST								EFW	EFMS			EFPRT			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-24 —	Reserved
23-16 EFSYN	On-Chip Memory ECC Fault Syndrome This read-only field specifies the checkbit syndrome from the last captured ECC event. For 32-bit memories, the syndrome is 7-bits, and EFSYN[7] == 0b. For 64-bit memories, the syndrome is 8-bits. The following describes the syndrome for a single-bit correctable error for a 32-bit RAM. For example, if there is a single-bit error in ram_rdata[21], then EFSYN[6:0] = 16h. If the EFSYN value is not found in the list below, a multi-bit, non-correctable error has been detected. • (EFSYN[6:0] == 00h) - no errors ram_rdata[31:0] • (EFSYN[6:0] == 61h) - 1-bit error on ram_rdata[0] • (EFSYN[6:0] == 51h) - 1-bit error on ram_rdata[1] • (EFSYN[6:0] == 19h) - 1-bit error on ram_rdata[2] • (EFSYN[6:0] == 45h) - 1-bit error on ram_rdata[3] • (EFSYN[6:0] == 43h) - 1-bit error on ram_rdata[4] • (EFSYN[6:0] == 31h) - 1-bit error on ram_rdata[5] • (EFSYN[6:0] == 29h) - 1-bit error on ram_rdata[6] • (EFSYN[6:0] == 13h) - 1-bit error on ram_rdata[7] • (EFSYN[6:0] == 62h) - 1-bit error on ram_rdata[8] • (EFSYN[6:0] == 52h) - 1-bit error on ram_rdata[9] • (EFSYN[6:0] == 4ah) - 1-bit error on ram_rdata[10] • (EFSYN[6:0] == 46h) - 1-bit error on ram_rdata[11] • (EFSYN[6:0] == 32h) - 1-bit error on ram_rdata[12] • (EFSYN[6:0] == 2ah) - 1-bit error on ram_rdata[13] • (EFSYN[6:0] == 23h) - 1-bit error on ram_rdata[14] • (EFSYN[6:0] == 1ah) - 1-bit error on ram_rdata[15] • (EFSYN[6:0] == 2ch) - 1-bit error on ram_rdata[16] • (EFSYN[6:0] == 64h) - 1-bit error on ram_rdata[17] • (EFSYN[6:0] == 26h) - 1-bit error on ram_rdata[18] • (EFSYN[6:0] == 25h) - 1-bit error on ram_rdata[19] • (EFSYN[6:0] == 34h) - 1-bit error on ram_rdata[20] • (EFSYN[6:0] == 16h) - 1-bit error on ram_rdata[21] • (EFSYN[6:0] == 15h) - 1-bit error on ram_rdata[22] • (EFSYN[6:0] == 54h) - 1-bit error on ram_rdata[23] • (EFSYN[6:0] == 0bh) - 1-bit error on ram_rdata[24]

Table continues on the next page...

Table continued from the previous page...

Field	Function
	• (EFSYN[6:0] == 58h) - 1-bit error on ram_rdata[25] • (EFSYN[6:0] == 1ch) - 1-bit error on ram_rdata[26] • (EFSYN[6:0] == 4ch) - 1-bit error on ram_rdata[27] • (EFSYN[6:0] == 38h) - 1-bit error on ram_rdata[28] • (EFSYN[6:0] == 0eh) - 1-bit error on ram_rdata[29] • (EFSYN[6:0] == 0dh) - 1-bit error on ram_rdata[30] • (EFSYN[6:0] == 49h) - 1-bit error on ram_rdata[31]
15-8 EFMST	On-Chip Memory ECC Fault Master Number This read-only field specifies the bus master responsible for initiating the last captured ECC event.
7 EFW	On-Chip Memory ECC Fault Write This read-only field specifies if the last captured ECC event was a write bus cycle. Since the ECC check only occurs on read operations, if the EFW bit is asserted, the captured ECC event was associated with the read-modify-write needed to generate the required check bits. This type of read-modify-write sequence only occurs on writes of less than 64 bits; for 64-bit writes, the destination check bits are calculated directly from the write data without the need for any read-modify-write sequence. 0b - Last captured ECC event was not a write bus cycle 1b - Last captured ECC event was a write bus cycle
6-4 EFMS	On-Chip Memory ECC Fault Master Size This read-only field specifies the access size of the last captured ECC event. The size encodings are: 000b - 8-bit size 001b - 16-bit size 010b - 32-bit size 011b - 64-bit size 100b - Reserved 101b - Reserved 110b - Reserved 111b - Reserved
3-0 EFPRT	On-Chip Memory ECC Fault Protection This read-only field specifies the protection field of the last captured ECC event. The protection encodings are: EFPRT[3] : Cacheable • 0 = non-cacheable • 1 = cacheable

Table continues on the next page...

Table continued from the previous page...

Field	Function
	EFPRT[2] : Bufferable • 0 = non-bufferable • 1 = bufferable EFPRT[1] : Mode 0 • 0 = user mode • 1 = supervisor mode EFPRT[0] : Type 0 • 0 = I-Fetch • 1 = Data

13.3.2.20 On-Chip Memory ECC Fault Data High Register (OCMFDRH)

Offset

Register	Offset
OCMFDRH	498h

Function

The OCMFDR is a 64-bit read-only register for capturing the read data of the last, properly-enabled ECC event in the on-chip memory. OCMFDRH is the upper 32 bits. Depending on the state of the On-Chip Memory ECC Control Register (OCMECR), an ECC event in the on-chip memory causes the address, attributes and read data associated with the access to be loaded into the OCMFAR, OCMFTR and OCMFDR registers, and the appropriate flag in the On-Chip Memory ECC Interrupt Register to be asserted. The read data is captured after it has been processed by the ECC logic, that is, in the case of a correctable single-bit ECC event, the read data is the corrected value. The contents of OCMFTR[EFSYN] can be used to locate the original faulted data bit. Attempted writes are ignored.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	EFDH															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	EFDH															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 EFDH	On-Chip Memory ECC Fault Data High This read-only field specifies the upper 32-bit read data word (data[63:32]) from the last captured ECC event. For ECC events that occur in 32-bit RAMs, this 32-bit field will return 32'h0.

13.3.2.21 On-Chip Memory ECC Fault Data Low Register (OCMFDRL)

Offset

Register	Offset
OCMFDRL	49Ch

Function

The OCMFDR is a 64-bit read-only register for capturing the read data of the last, properly-enabled ECC event in the on-chip memory. OCMFDRL is the lower 32 bits. Depending on the state of the On-Chip Memory ECC Control Register (OCMECR), an ECC event in the on-chip memory causes the address, attributes and read data associated with the access to be loaded into the OCMFAR, OCMFTR and OCMFDR registers, and the appropriate flag in the On-Chip Memory ECC Interrupt Register to be asserted. The read data is captured after it has been processed by the ECC logic, that is, in the case of a correctable single-bit ECC event, the read data is the corrected value. The contents of OCMFTR[EFSYN] can be used to locate the original faulted data bit. Attempted writes are ignored.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	EFDL															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	EFDL															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 EFDL	On-Chip Memory ECC Fault Data Low This read-only field specifies the lower 32-bit read data word (data[31:0]) from the last captured ECC event.

13.3.2.22 Core Platform Control Register (CPCR)

Offset

Register	Offset
CPCR	C00h

Function

This register provides the ability to control the round robin arbitration mode in the various bus crossbars in the system.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0															AXBS0_R...
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Fields

Field	Function
31-1 —	Reserved
0 AXBS0_RREN	AXBS0 Round Robin Enable When set, this forces the AXBS0 to round robin arbitration mode out of reset. To change the arbitration mode after reset is negated, the AXBS0 programming model must be used. Enabled at reset. 0b - AXBS0 in fixed priority arbitration mode at reset. 1b - AXBS0 in round robin arbitration mode at reset.

Appendix A
Release notes

A.1 About this manual changes

No substantial content changes.

A.2 Security Overview changes

No substantial content changes.

A.3 Lifecycle changes

No substantial content changes.

A.4 ROM Bootloader changes

- Updated Low power wakeup path.

A.5 ROM API changes

- Added the following sections
  - nboot_force_one_shot_secure_level.
  - nboot_sb3_consistency_verify.

A.6 ISP Path changes

No substantial content changes.

A.7 ELE changes

No substantial content changes.

A.8 ELE Software Architecture and API changes

No substantial content changes.

A.9 ELE Messaging Unit (ELEMU)

A.9.1 ELE Messaging Unit bridge topic changes

No substantial content changes.

A.9.2 S3MU changes

No substantial content changes.

A.10 Key Management changes

No substantial content changes.

A.11 Debug Subsystem (DBGMB)

A.11.1 Debug Mailbox bridge topic changes

No substantial content changes.

A.11.2 DBGMB module changes

No substantial content changes.

A.12 Flash Memory Controller (FMC) with NVM PRINCE Encryption and Decryption (NPX)

A.12.1 Flash Memory Controller (FMC-NPX) bridge topic changes

No substantial content changes.

A.12.2 Flash Memory Controller (FMC-NPX) module changes

No substantial content changes.

A.13 Secure Miscellaneous System Control Module (SMSCM)

A.13.1 Secure Miscellaneous System Control bridge topic changes

No substantial content changes.

A.13.2 SMSCM changes

No substantial content changes.

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors’ warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors’ specifications such use shall be solely at customer’s own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors’ standard warranty and NXP Semiconductors’ product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer’s applications and products. Customer’s responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer’s applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Please be aware that important notices concerning this document and the product(s) described herein, have been included in section 'Legal information'.

For more information, please visit: https://www.nxp.com

Date of release: 2 December 2025

Document identifier: KW47SRM