NXP Semiconductors
Reference Manual

Document identifier: KW45SRM
Rev. 7, 2025-12-09

KW45 Security Reference Manual

Supports KW45B41Z82AFTA, KW45B41Z83AFTA, KW45B41Z82AFPA, KW45B41Z83AFPA, KW45B41Z52AFTA, KW45B41Z53AFTA, KW45B41Z52AFPA, KW45B41Z53AFPA, KW45Z41082AFTA, KW45Z41083AFTA, KW45Z41082AFPA, KW45Z41083AFPA, KW45Z41052AFTA, KW45Z41053AFTA, KW45Z41052AFPA, KW45Z41053AFPA

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.

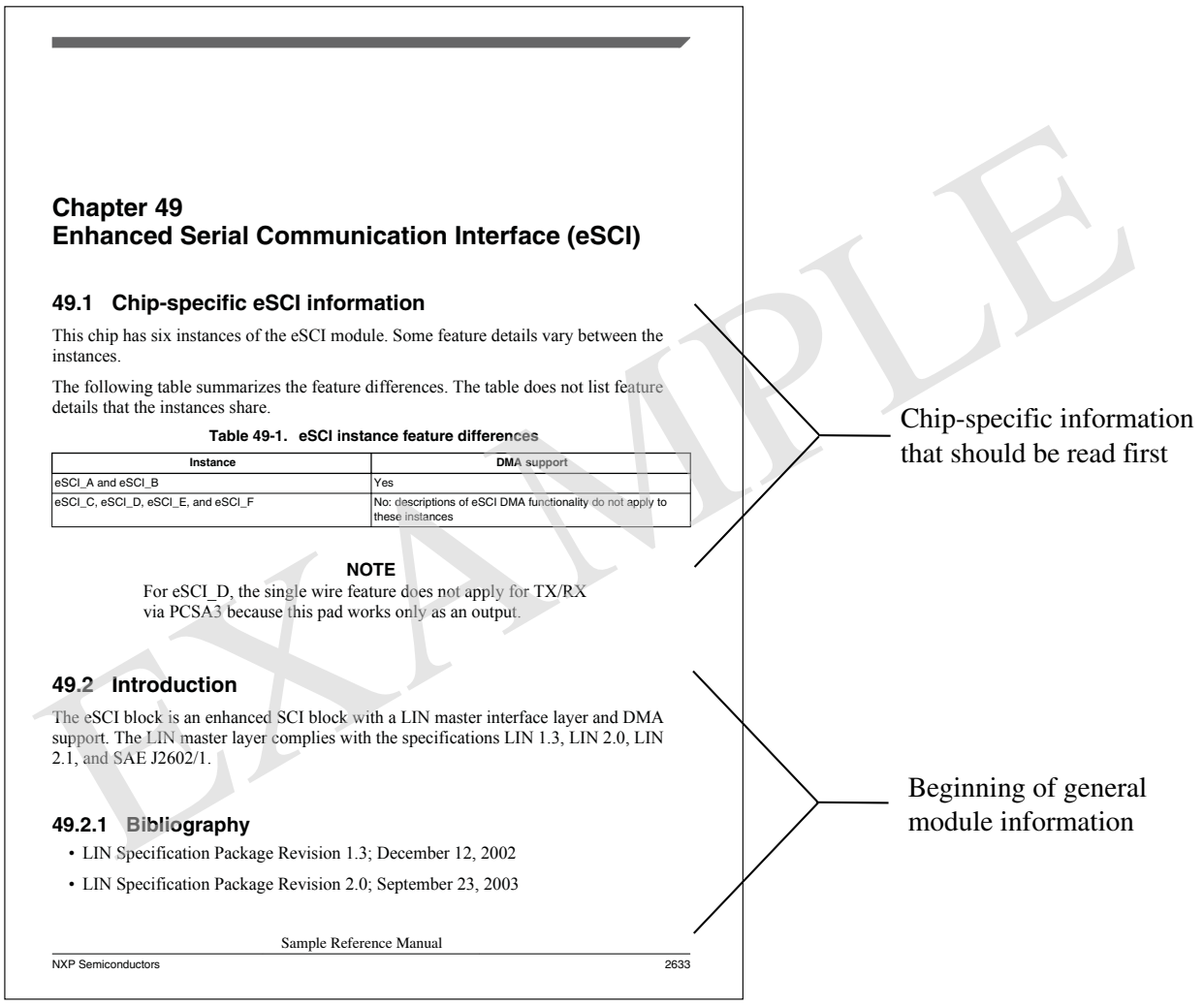
1.3 Module descriptions

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

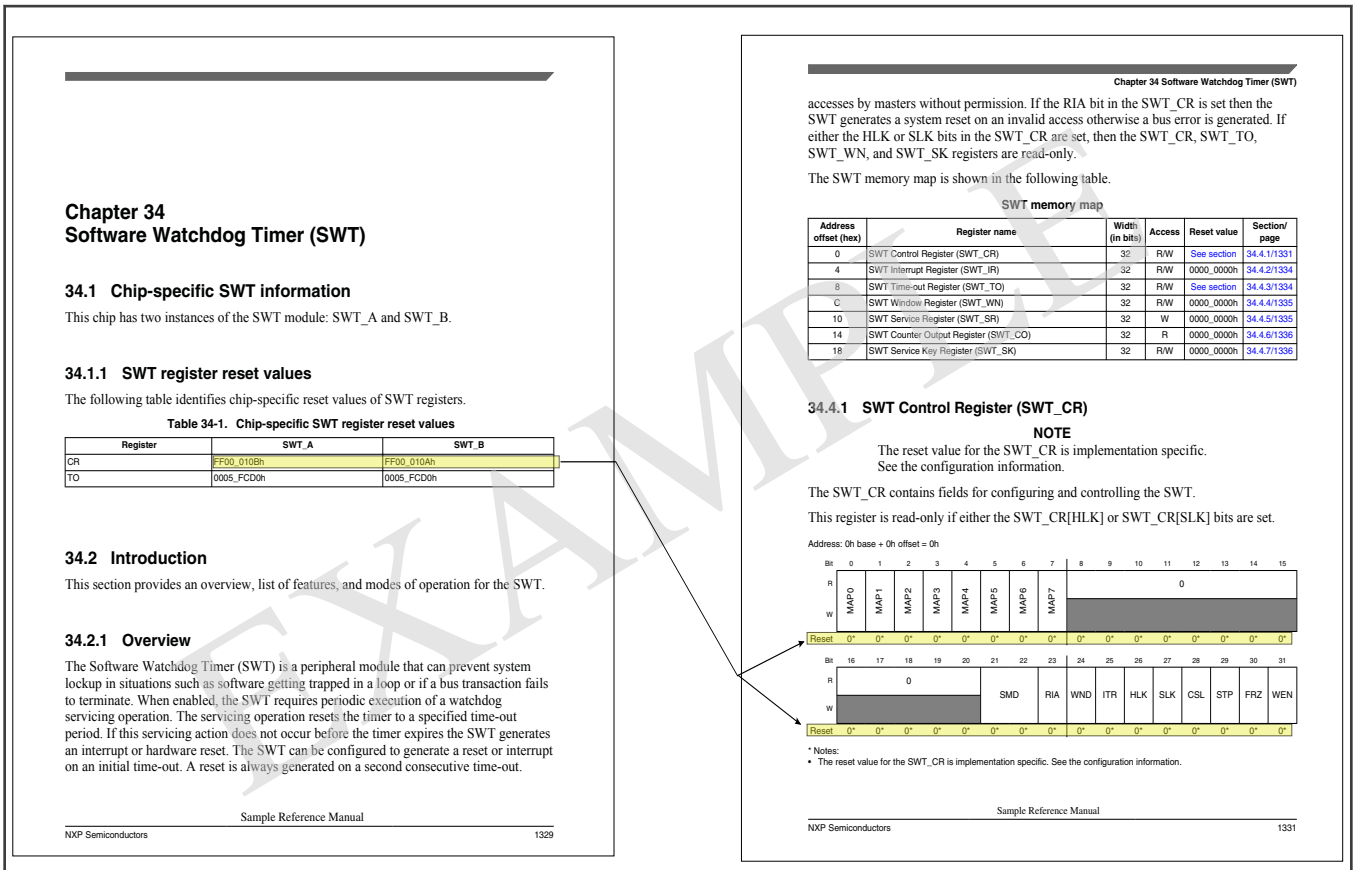
- LIN Specification Package Revision 1.3; December 12, 2002
- LIN Specification Package Revision 2.0; September 23, 2003

Sample Reference Manual
NXP Semiconductors 2633

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.

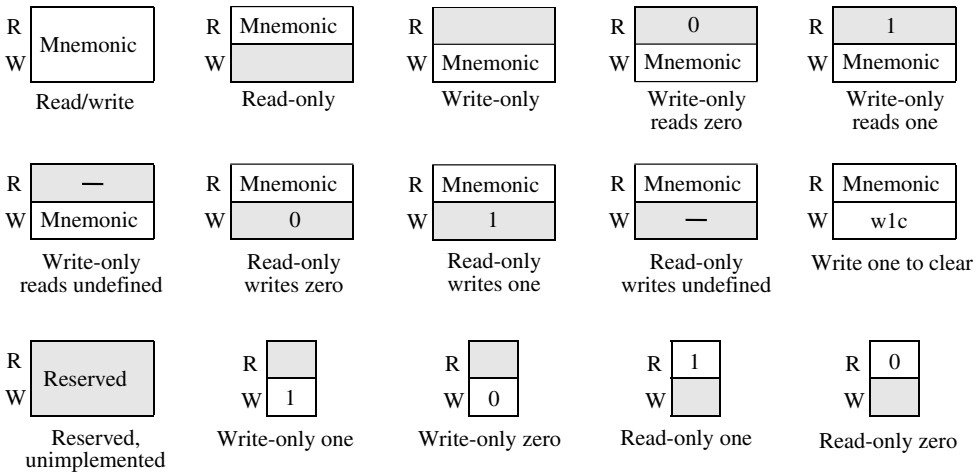


Figure 3. Register figure conventions

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 Overview

This chapter provides an overview of the following chip security components, explaining the purpose and features of each of them.

- 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 controller to store security configuration
- Anomaly Detection and Response sub-system:
  - Low-Voltage (LVD) and High-Voltage (HVD) Glitch Detectors for VDD core
  - Windowed Watch Dog Timers (WDOG)
  - Analog glitch sensor
  - 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 protection
  - Secure Debug: Certificate based debug authentication mechanism using ECDSA P-256 and P-384 keys
  - Secure Provisioning supporting following flows
  - Secure Identity solutions
- Secure Storage:
  - ELE Key store generated from UDF based device unique key.

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
MontDH	255	Curve25519
ECDH(E)	-	NIST P-192, P-224, P-256, P-384, and P-521 curves
ECDSA	192, 224, 256, 384, 521	NIST P-192, P-224, P-256, P-384, and P-521 curves
Ed25519	255	Curve25519

Table continues on the next page...

Table 1. Cryptographic operations (continued)

Algorithm	Key lengths	Modes
SHA2	224, 256, 384, 512	-
HMAC	-	SHA-256
CMAC	128, 256	AES

See KW45 Reference Manual for more details.

2.2 Immutable Root of Trust

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.” It 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, Life-cycle Management, and a number of others. In this device, Immutable RoT is embedded in the Boot ROM (immutable bootloader). It uses the device hardware cryptographic functionality for its function.

2.3 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.

2.4 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.

2.5 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 protection.

2.6 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.

2.7 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, ELE,

NBU) 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:

- TRDC to implement secure isolation, with one single TRDC instance supporting multiple domains and can be configured to set different access permissions for different Domain IDs when accessing different regions of memory (including SRAM, peripherals, and Flash)
- 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

2.8 Secure Attestation

Secure Attestation is a set of mechanisms used to provide evidence to a remote party on the device's genuine identity, its software and firmware versions, as well as its integrity and lifecycle state. Device Identity Composition Engine (DICE), as defined by Trusted Computing Group, uses Immutable RoT during boot time to create a unique Device Identity which considers Unique Device Secret (UDS), hardware state of the device and its firmware. Runtime Fingerprint (RTF) is the NXP-proprietary attestation mechanism, where the ELE ROM or firmware collects the values of the SoC level register consolidating various signals, and generates the RTF using these values.

2.9 Secure Storage

ELE Key store is generated from UDF based device unique key (DUK). The ELE implements a key storage, which can be used to store user keys for crypto operations executed by the ELE. The ELE key storage supports symmetric and asymmetric private keys.

2.10 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.

2.11 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.

2.12 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
EdgeLock Secure Enclave (ELE)

3.1 Introduction

While the emerging application space known as the Internet of Things (IoT) provides an incredible market and business opportunity, it also brings a variety of technical challenges for the microcontrollers located as edge nodes in massively-connected networks. Clearly, one of the most important technical issues remains the definition of an appropriate level of device security for the MCU devices - an application area where simple hardware acceleration of cryptographic functions is no longer an acceptable solution.

Indeed, the IoT market represents a perfect storm where the combined legacy security models for static microcontroller configurations and networked devices are no longer appropriate; instead a new paradigm for MCU security is emerging and NXP is using its industry-leading position as the preeminent supplier of secure solutions to lead the way. An EdgeLock secure enclave, or simply EdgeLock enclave (ELE) has been defined and implemented that significantly raises the level of security provided by a traditional connected MCU device.

The EdgeLock secure enclave is an independent black box processing element providing essential security services to the host system within a single SoC device specifically in two areas: secret key storage and management, and execution of symmetric and public key cryptographic services, including secure hashing, based on requests from the host.

The EdgeLock enclave provides a secure environment within an otherwise non-secure device in which applications can execute secure cryptographic services.

A high-level block diagram for the EdgeLock enclave is shown below:

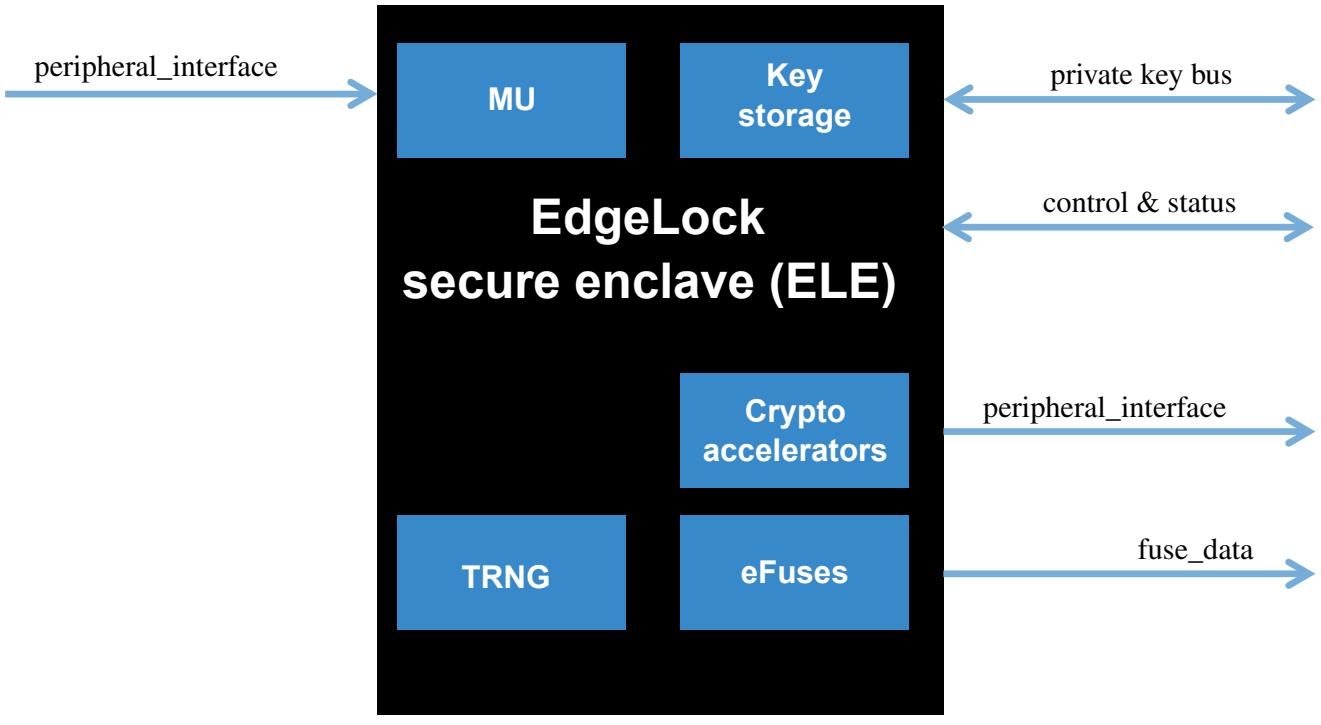


Figure 4. EdgeLock enclave block diagram

From the EdgeLock enclave's perspective, there are two major data interfaces:

```
peripheral_interface         // host-to-peripheral interface (APB) to access the messaging unit
system_interface         // ELE-to-system interface (AHB) to access host memories
```

Additionally, there are three "minor" interfaces:

```
control & status    // miscellaneous i/o connections for control & status
private_key_bus     // secure bus to distribute keys to host IP modules
fuse_data           // output bus to distribute fuse values to host
```

Internally, EdgeLock enclave provides hardware support for a messaging unit (MU) to receive service requests from the host, secret key storage and management, cryptographic accelerators, a True Random Number Generator (TRNG) entropy source and a device configuration fusebox (eFuses). Refer to Software Architecture and API for software interface to configure the EdgeLock enclave features.

3.2 Features

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

- Secret key generation, storage and management
  - Hardware key derivation from non-volatile memory within the SoC
  - Implementing bus and memory obfuscation for cryptographic protection
- Secure crypto services during secure boot and normal runtime
  - Simple peripheral interface with the host processor to minimize attack surface
    - Host application makes function calls to EdgeLock enclave via a messaging unit
    - Mailbox messages used to pass crypto services commands, pointers and data
    - EdgeLock enclave implements a “run to completion” execution model for crypto tasks
  - EdgeLock enclave has access to the host's internal and external memories via a system interface
  - Operations and services controlled by CryptoLib software

3.2.1 Symmetric algorithms

“Symmetric-key cryptography refers to encryption methods in which both the sender and receiver share the same key (or, less commonly, in which their keys are different, but related in an easily computable way). This was the only kind of encryption publicly known until June 1976.

Symmetric key ciphers are implemented as either block ciphers or stream ciphers. A block cipher enciphers input in blocks of plaintext as opposed to individual characters, the input form used by a stream cipher.” [1]

[1] Taken from *https://en.wikipedia.org/wiki/Cryptography*

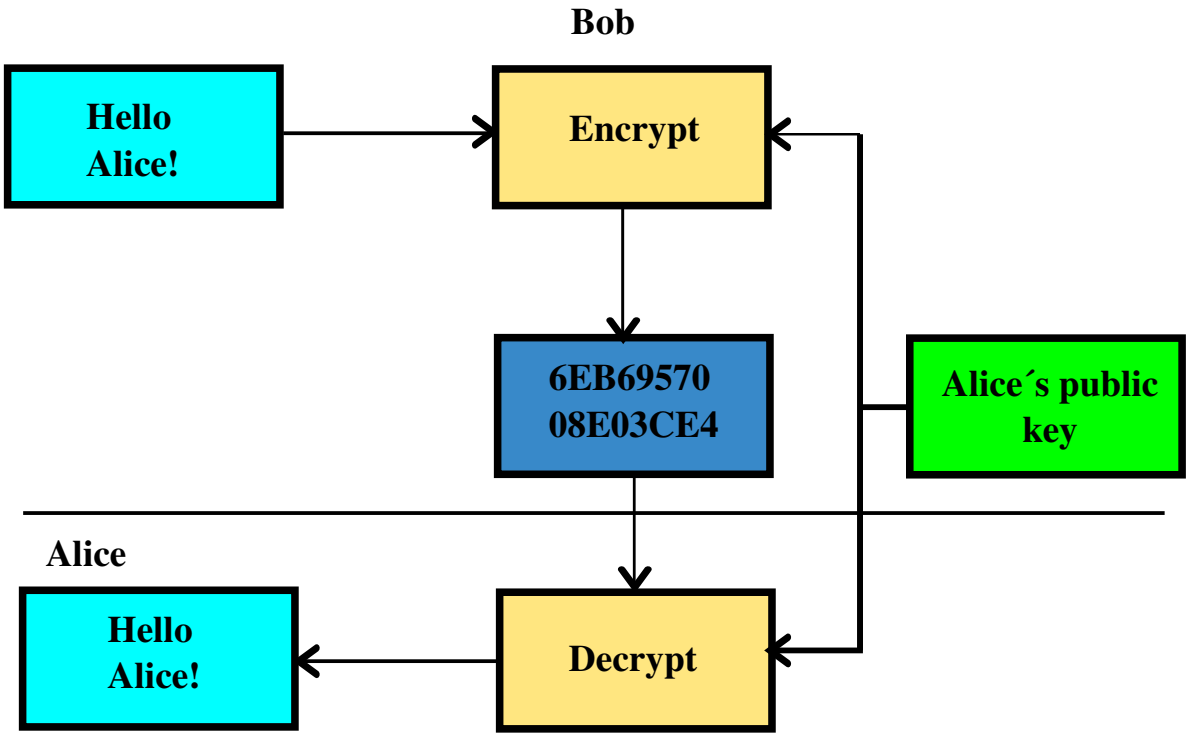


Figure 5. Symmetric-key cryptography, where a single key is used for encryption and decryption [1]

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.

EdgeLock enclave natively supports the following AES algorithms along with a variety of block cipher modes:

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

The following table shows the supported commands for symmetric algorithms.

Table 2. 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

3.2.2 Asymmetric algorithms

“Symmetric-key cryptosystems use the same key for encryption and decryption of a message, although a message or group of messages can have a different key than others. A significant disadvantage of symmetric ciphers is the key management

necessary to use them securely. Each distinct pair of communicating parties must, ideally, share a different key, and perhaps for each ciphertext exchanged as well. The number of keys required increases as the square of the number of network members, which very quickly requires complex key management schemes to keep them all consistent and secret.

In a groundbreaking 1976 paper, Whitfield Diffie and Martin Hellman proposed the notion of public-key (also, more generally, called asymmetric key) cryptography in which two different but mathematically related keys are used—a public key and a private key. A public key system is so constructed that calculation of one key (the 'private key') is computationally infeasible from the other (the 'public key'), even though they are necessarily related. Instead, both keys are generated secretly, as an interrelated pair. The historian David Kahn described public-key cryptography as "the most revolutionary new concept in the field since polyalphabetic substitution emerged in the Renaissance".

In public-key cryptosystems, the public key may be freely distributed, while its paired private key must remain secret. In a public-key encryption system, the public key is used for encryption, while the private or secret key is used for decryption. While Diffie and Hellman could not find such a system, they showed that public-key cryptography was indeed possible by presenting the Diffie–Hellman key exchange protocol, a solution that is now widely used in secure communications to allow two parties to secretly agree on a shared encryption key."[1]

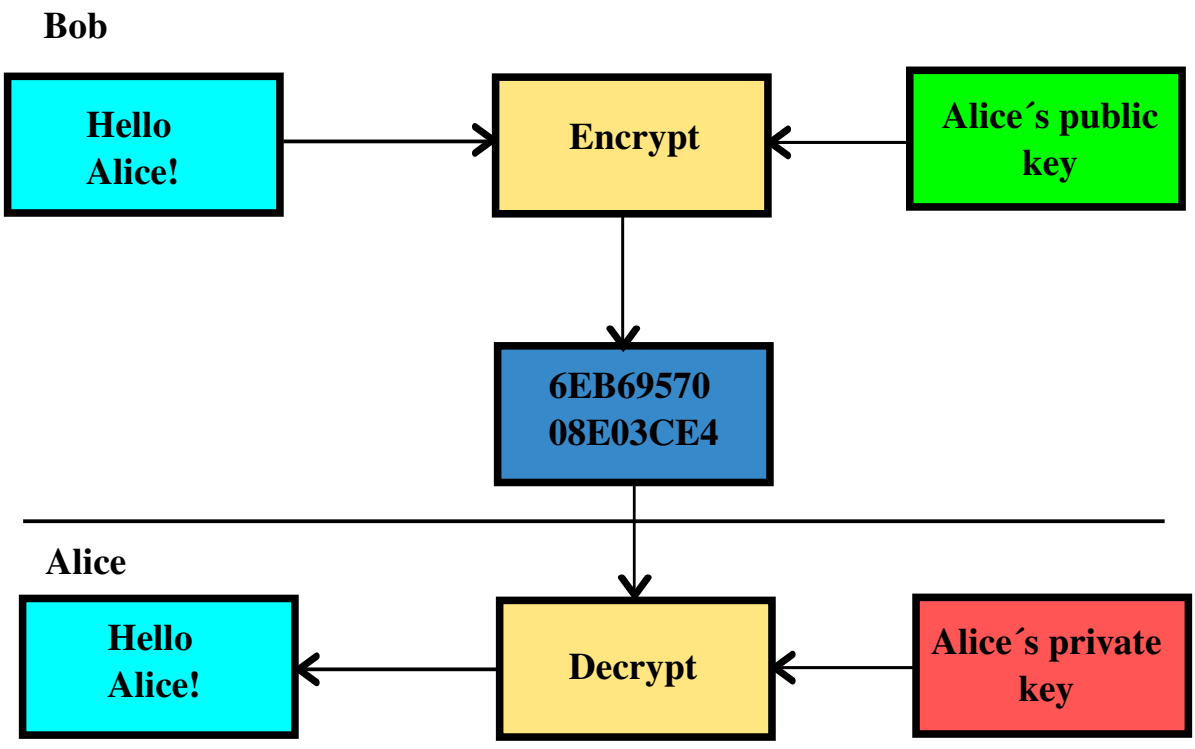


Figure 6. Public-key cryptography, where different keys are used for encryption and decryption[1]

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.

EdgeLock enclave 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)
- Curve25519

The following table shows the supported commands for asymmetric algorithms.

Table 3. 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
0x40	DERIVE_KEY_CONTEXT_INIT	Initialize context for Diffie-Hellman key exchange operation
0x3C	ASYMMETRIC_DH_DERIVE_KEY	Perform Diffie-Hellman key exchange operation

3.2.3 Authenticated encryption with associated data (AEAD) algorithms

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

Table 4. 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

3.2.4 Hash functions

"Cryptographic hash functions are a third type of cryptographic algorithm. They take a message of any length as input, and output a short, fixed length hash, which can be used in (for example) a digital signature. For good hash functions, an attacker cannot find two messages that produce the same hash.

Unlike block and stream ciphers that are invertible, cryptographic hash functions produce a hashed output that cannot be used to retrieve the original input data. Cryptographic hash functions are used to verify the authenticity of data retrieved from an untrusted source or to add a layer of security.[1]

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

Secure Hash Algorithm 2 (SHA-2) - The SHA-2 algorithms 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 SHA-224, SHA-256, SHA-384, and SHA-512. The message digest can be used to verify the integrity of a set of information.

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

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

EdgeLock enclave supports:

- Secure Hash Algorithm 1 (SHA-1)
- Secure Hash Algorithm 224 (SHA-224)
- Secure Hash Algorithm 256 (SHA-256) (hardware support)

- Secure Hash Algorithm 384 (SHA-384)
- Secure Hash Algorithm 512 (SHA-512)

The following table shows the supported commands for HASH functions.

Table 5. 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

3.2.5 Message authentication (MAC) algorithms

Message authentication codes (MACs) are much like cryptographic hash functions, except that a secret key can be used to authenticate the hash value upon receipt; this additional complication blocks an attack scheme against bare digest algorithms, and so has been thought worth the effort."[1]

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

Table 6. Message authentication algorithms commands

Command ID	Commands	Descriptions
0x32	MAC_CONTEX_INIT	Initialize context for message authentication code (MAC) operation
0x35	MAC_ONE_GO	Perform message authentication code (MAC) operation in one step

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

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

Table 7. MBI and SB3 authentication commands

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

3.2.7 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.

EdgeLock enclave 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 8. TRNG commands

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

3.2.8 Key storage services

The EdgeLock enclave implements a key store, which can be used to store user keys for crypto operations executed in EdgeLock enclave. Refer to Key Management for more details.

The following table shows the supported commands for key storage.

Table 9. 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 EdgeLock enclave 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 EdgeLock enclave 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

3.2.9 EdgeLock enclave TrustZone and multicore/multithread support

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

- Host bus attributes masquerading
- Security level protection

- Random identifiers
- Hardware semaphore

3.2.9.1 Host bus attributes masquerading

Some crypto operation commands require access to the host memory resources. If TrustZone environment is configured and enabled, the EdgeLock enclave, as any other bus owner, needs to use proper bus transaction security level during access to the host memory. The EdgeLock enclave bus transaction security level is not controlled directly by a user but it "masquerades" as the same host task. Stated differently, the EdgeLock enclave uses the same security level as the host, which initiated the EdgeLock enclave 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 than host itself. 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 EdgeLock enclave 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 EdgeLock enclave switches back to masquerading mode and inherits the bus transaction security level from the host.

3.2.9.2 EdgeLock enclave security level protection

If TrustZone environment is configured and enabled, the EdgeLock enclave utilizes a protection mechanism to prevent an application thread with a lower security level to interrupt an unfinished operation. For example, if EdgeLock enclave 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 is captured during the KEY_OBJECT_INIT command and key user must have the same or higher security level.

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

3.2.9.3 Random object identifiers

Every EdgeLock enclave 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 EdgeLock enclave commands. The object identifier is generated in runtime as a random number. The random identifier helps to better isolate EdgeLock enclave objects among different cores or application threads and thus avoid misuse of EdgeLock enclave objects.

3.2.9.4 Hardware semaphore

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

3.2.10 Other services

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

Table 10. Commands for other services

Command ID	Commands	Descriptions
0x11	PING	Check whether EdgeLock enclave is alive and ready to receive new command

Table continues on the next page...

Table 10. Commands for other services (continued)

Command ID	Commands	Descriptions
0x13	OPEN_SESSION	Open session between the host and the EdgeLock enclave
0x14	CLOSE_SESSION	Close opened session
0x60	MGMT_ADVANCE_LIFECYCLE	Change lifecycle state into the next state
0x65	MGMT_CONTEXT_INIT	Initialize context for EdgeLock enclave management commands
0x67	MGMT_FUSE_PROGRAM	Perform EdgeLock enclave fuse programming
0x68	MGMT_FUSE_READ	Perform EdgeLock enclave 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 EdgeLock enclave command
0x73	MGMT_GET_RANDOM	Get random number of requested length
0x74	MGMT_CLEAR_ALL_KEYS	Overwrite complete key store with random numbers

3.2.11 Response messages

The following table shows the supported response messages.

Table 11. Response messages

Messages	Descriptions
INITIALIZATION_FINISHED	Asynchronously sent by EdgeLock enclave after RESET
ABORT	EdgeLock enclave sends this message asynchronously when unexpected error detected during command execution.
SECURITY_VIOLATION	EdgeLock enclave sends this message asynchronously when security violation detected during command execution.
UNKNOWN_COMMAND	EdgeLock enclave sends this message when the command ID is not recognized or supported

Chapter 4
Messaging Unit (ELE_MU)

4.1 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.

4.1.1 Block Diagram

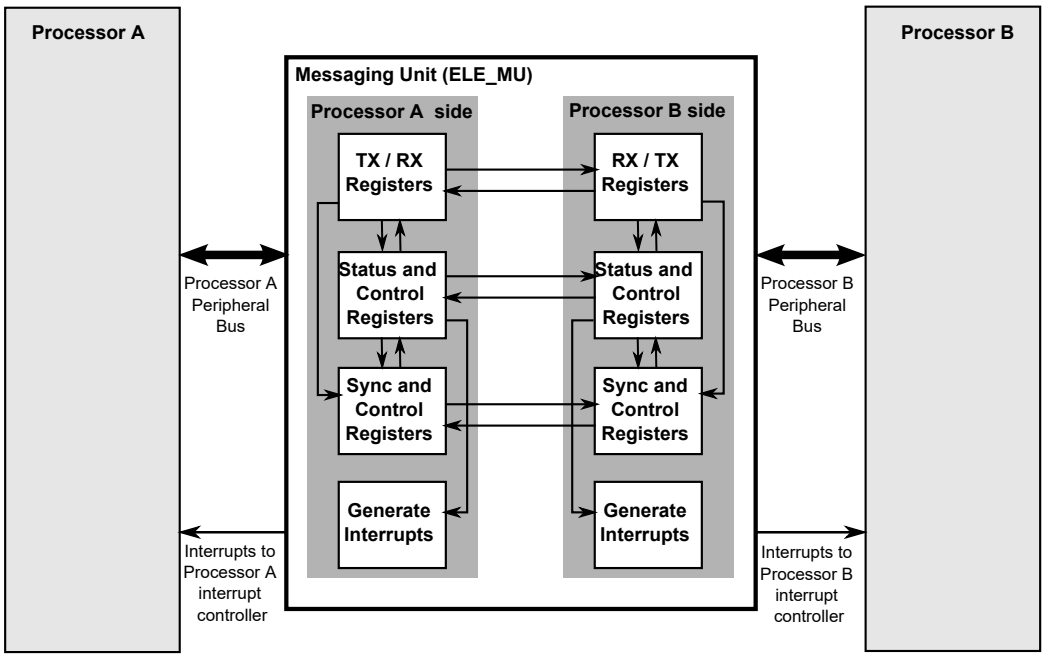


Figure 7. ELE_MU Block Diagram

4.1.2 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.

4.2 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.

4.2.1 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.

4.3 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.

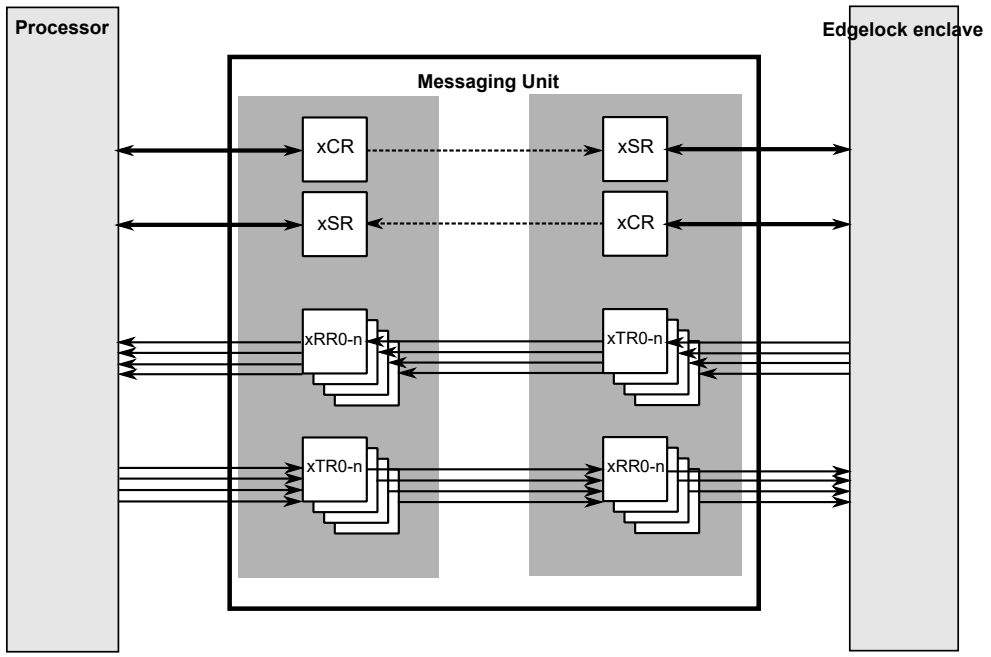


Figure 8. ELE_MU Registers

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.

4.3.1 ELE_MUA register descriptions

This section contains the detailed register descriptions for the ELE_MUA registers.

4.3.1.1 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

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
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

4.3.1.2 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.

4.3.1.3 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)

Table continues on the next page...

Table continued from the previous page...

Field	Function
7-0 TR_NUM	Number of Transmit (TRn) registers (8'd16)

4.3.1.4 Unused Register 0 (UNUSED0)

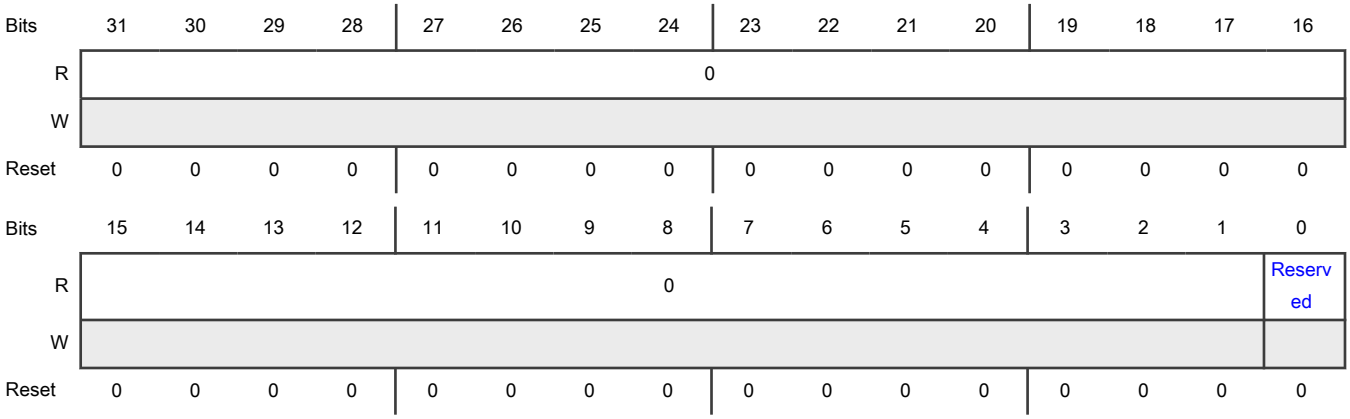
Offset

Register	Offset
UNUSED0	8h

Function

DO NOT WRITE

Diagram



Fields

Field	Function
31-1 —	Reserved
0 —	DO NOT WRITE TO THIS BIT FIELD.

4.3.1.5 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

4.3.1.6 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	TEIEn															
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 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.

4.3.1.7 Transmit Status Register (TSR)

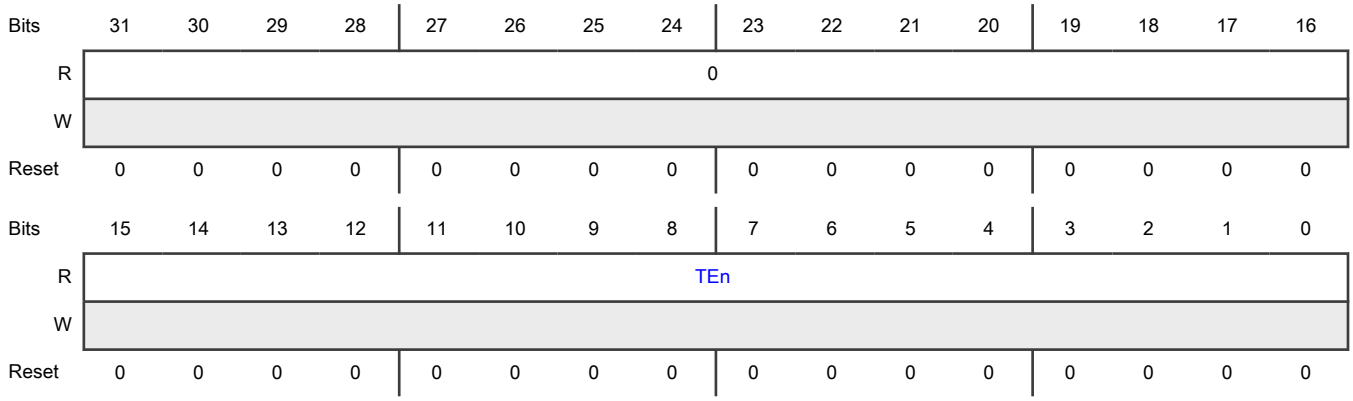
Offset

Register	Offset
TSR	124h

Function

The TSR register shows which TR registers are ready to be written.

Diagram



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. • 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.

4.3.1.8 Receive Status Register (RSR)

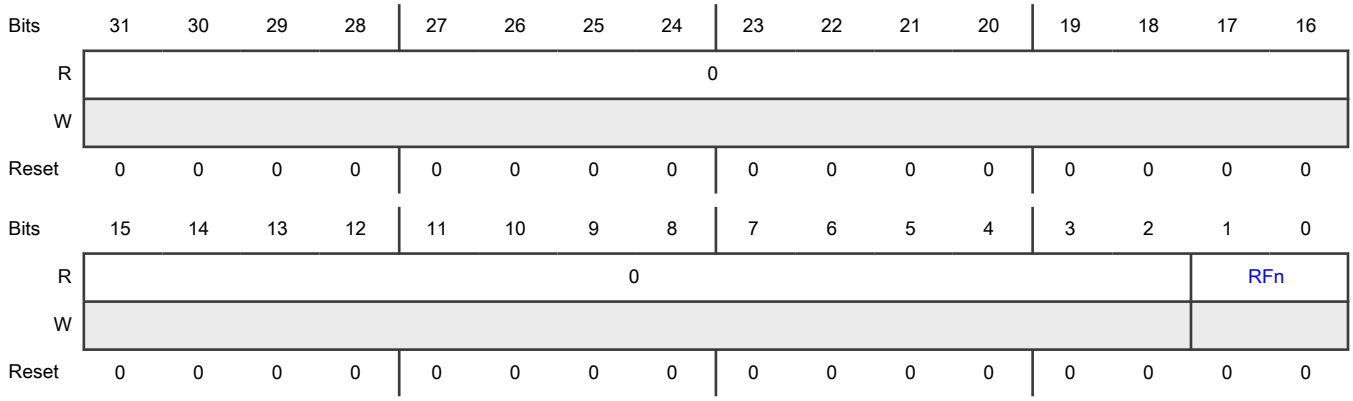
Offset

Register	Offset
RSR	12Ch

Function

The RSR register shows which RR registers are ready to be read.

Diagram



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. • 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.

4.3.1.9 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	DATA16															
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 DATA16	Unused 16-bit Register

4.3.1.10 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.

4.3.1.11 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).

Table continues on the next page...

Field	Function
	• 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".

4.3.1.12 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

Table continues on the next page...

Table continued from the previous page...

Field	Function
	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

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`

4.3.1.13 Semaphore Ownership Register (SEMA4_OWNR)

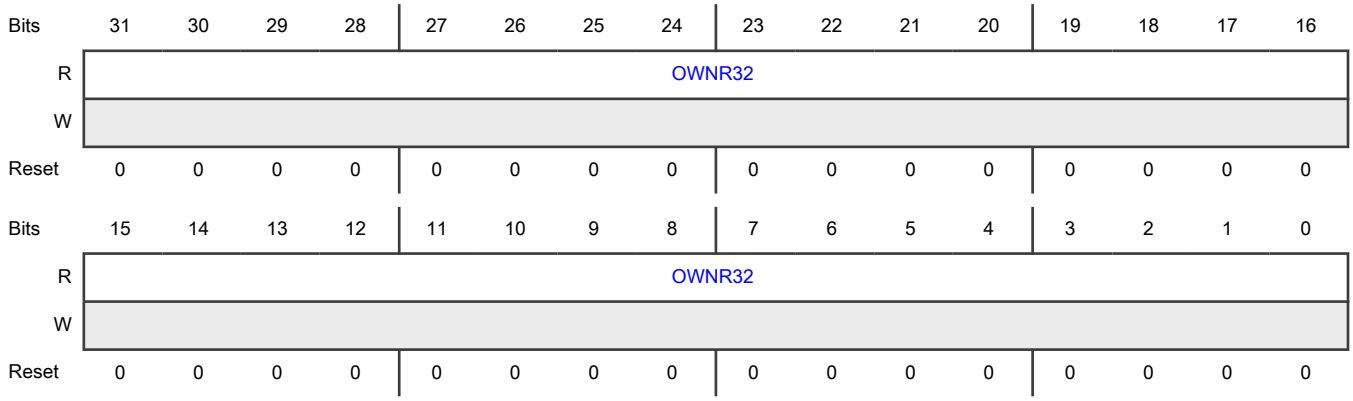
Offset

Register	Offset
SEMA4_OWNR	474h

Function

Reads to this virtual register return the semaphore ownership state.

Diagram



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 {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`

4.3.1.14 Semaphore Acquire Register (SEMA4_ACQ)

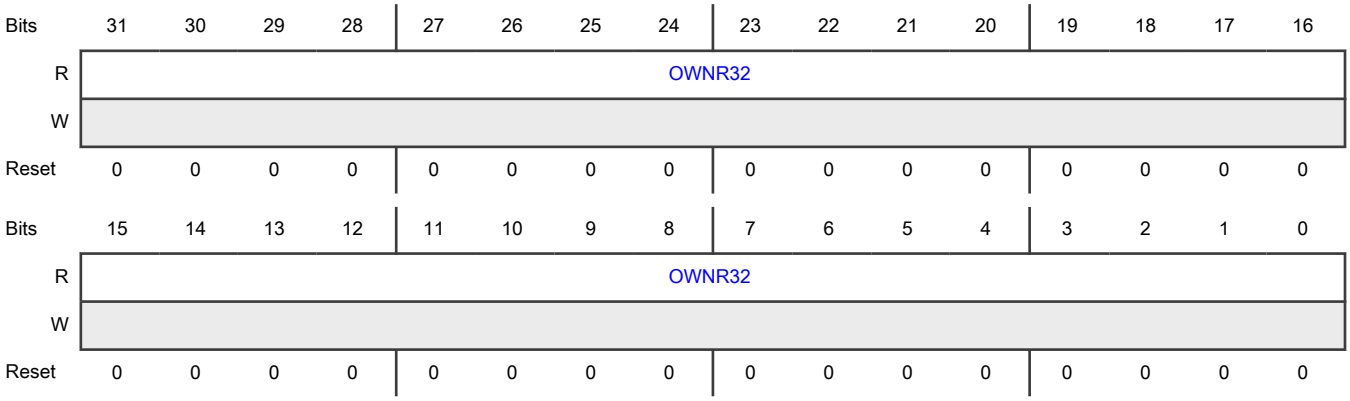
Offset

Register	Offset
SEMA4_ACQ	998h

Function

Reads to this virtual register attempt to acquire (lock) the semaphore for the requesting task.

Diagram



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.
	`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`

4.3.1.15 Semaphore Release Register (SEMA4_REL)

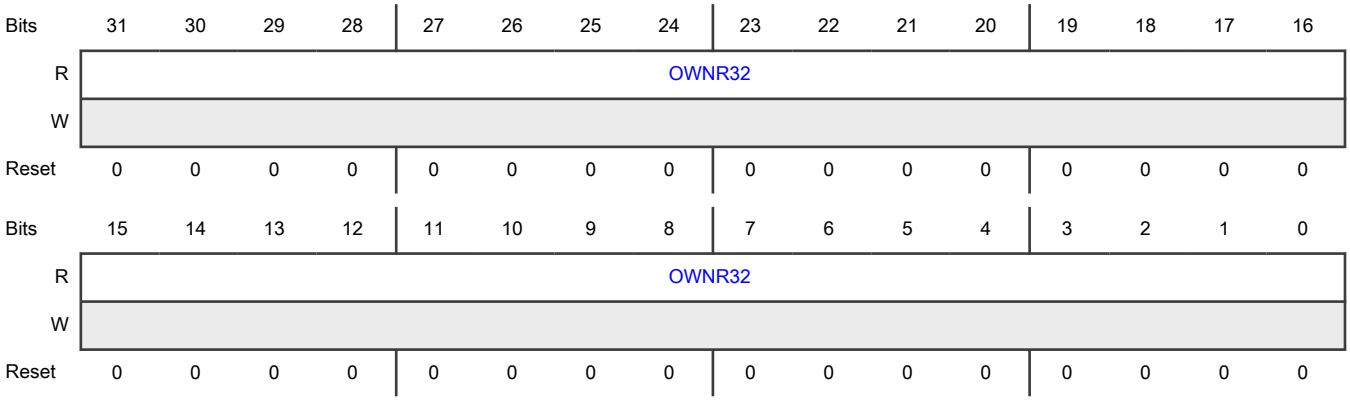
Offset

Register	Offset
SEMA4_REL	ACCh

Function

Reads to this virtual register attempt to release (unlock) the semaphore for the requesting task.

Diagram



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`

4.3.1.16 Semaphore Forced Release Register (SEMA4_FREL)

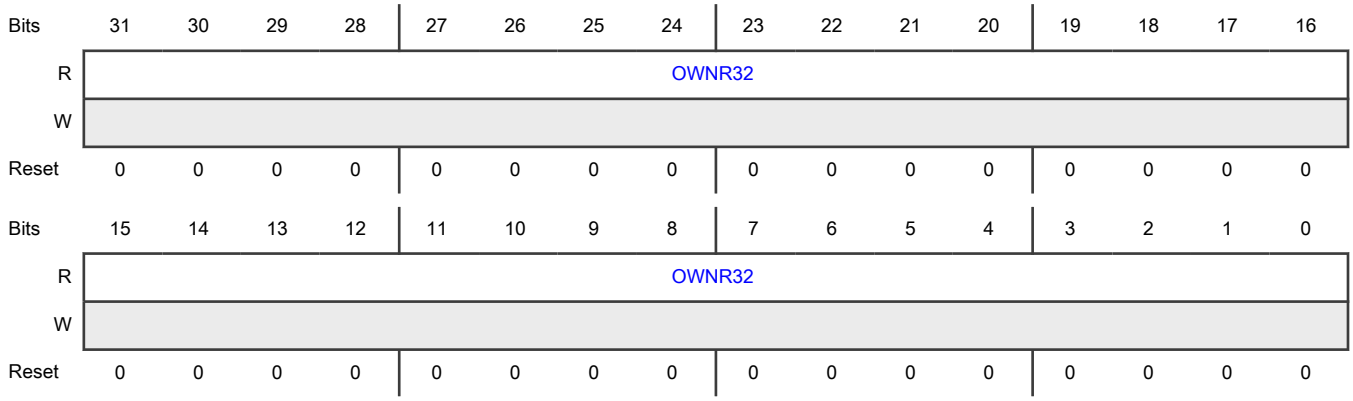
Offset

Register	Offset
SEMA4_FREL	BA4h

Function

Reads to this virtual register attempt to unconditionally release (unlock) the semaphore.

Diagram



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 5
Lifecycle

5.1 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.

5.2 Lifecycle states and transitions

Figure 9 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.

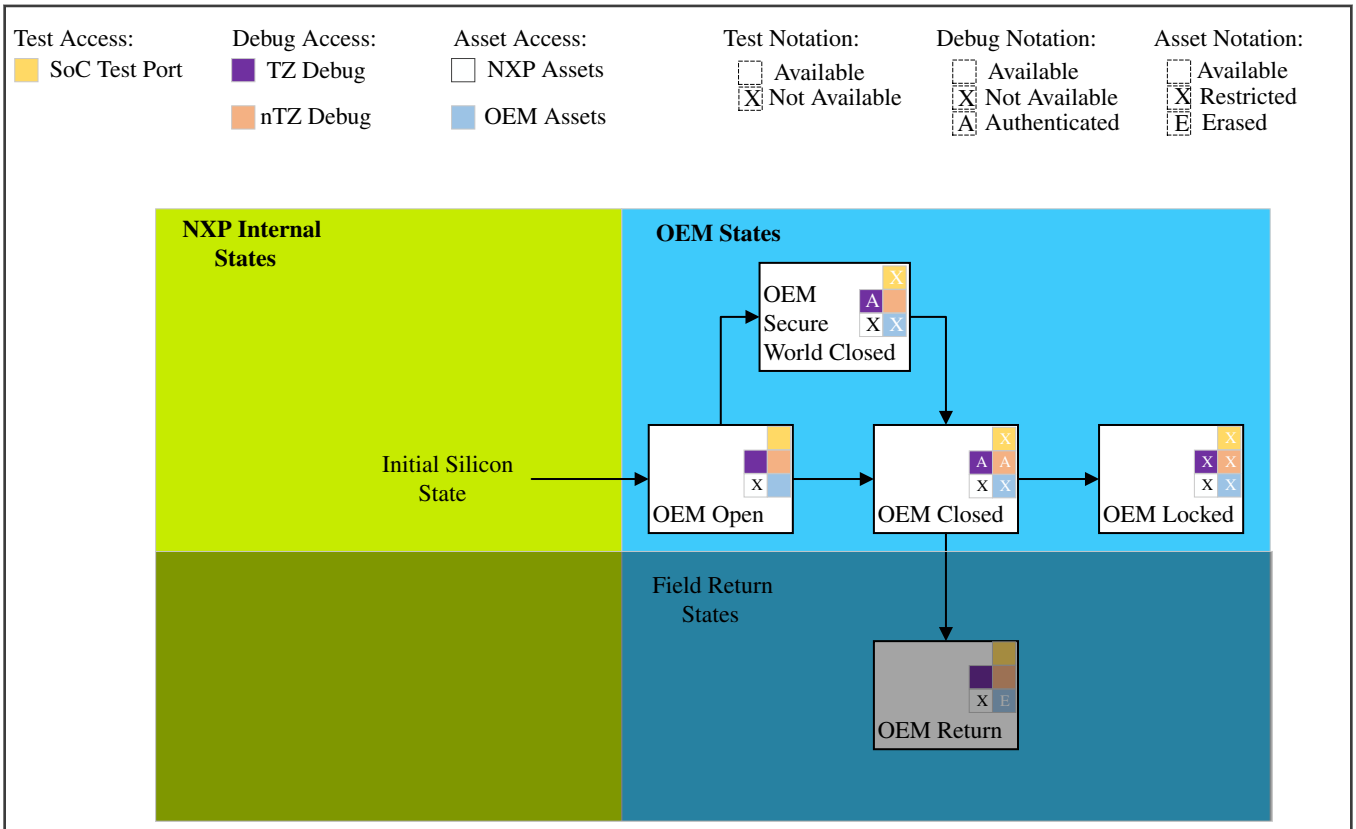


Figure 9. Security lifecycle states

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.

5.3 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. See the attached fuse fields for more information.

Table 12. Lifecycle states

Lifecycle	Lifecycle state	CM33 authentication required?	Radio core authentication required?	CM33 debug authentication required?	TZ debug port authentication required?	nTZ debug port authentication required?	ISP cmds	NXP assets	OEM assets
OEM OPEN	0000_0111	No (boot even if authentication fails)	No	No	No	No	All	Restricted	Available
OEM SECURE WORLD CLOSED	0000_1111	Yes	Yes	Yes	Yes	No	Limited	Restricted	Restricted
OEM Closed	0001_1111	Yes	Yes	Yes	Yes	Yes	Limited	Restricted	Restricted
OEM Locked	1001_1111	Yes	Yes	NA	NA	NA	Limited	Restricted	Restricted
OEM Return	0011_1111	NA	NA	No	No	No	None	Restricted	Erased

5.4 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.

5.4.1 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.

5.4.1.1 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

5.4.1.2 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

5.4.2 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.

5.4.2.1 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.

5.4.3 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.

5.4.3.1 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.

5.4.3.2 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

5.4.4 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 life cycle, the device cannot be transitioned to any other state.

5.5 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.

5.5.1 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 6
Key Management

6.1 Introduction

The EdgeLock enclave implements a key store, which can be used to store user keys for crypto operations executed by the EdgeLock enclave. The EdgeLock enclave key store supports the following key types:

- Symmetric keys
- Asymmetric private keys, public keys or key pairs for elliptic-curve cryptography (ECC) with length up to 521 bits (NIST P-192, P-224, P-256, P-384, P-521 and Curve25519)

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: 4608 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 EdgeLock enclave 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 example below:

```
//KEY = 1f8e4973953f3fb0bd6b16662e9a3c17
uint8_t symKeyData[16] = {0x1f, 0x8e, 0x49, 0x73, 0x95, 0x3f, 0x3f, 0xb0, 0xbd, 0x6b, 0x16, 0x66,
0x2e, 0x9a, 0x3c, 0x17};
```

6.2 Key object properties

The key object properties can restrict key usage for specific crypto operations or specific mode within crypto operation.
The key object properties are stored as a 32-bit value that is sent or received by the ELEMUA_TRn and ELEMUA_RRn registers. The key properties are set during the KEY_OBJECT_ALLOCATE_HANDLE command or any time later using the KEY_OBJECT_SET_PROPERTIES command. KEY_OBJECT_GET_PROPERTIES command can be used to read back the properties for a key object. Both SEC and TRUSTED_KEY are read only properties. The SEC property is set during the KEY_OBJECT_INIT command. The TRUSTED_KEY property is set during KEY_STORE_GENERATE_KEY command.

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.

Table continues on the next page...

Table continued from the previous page...

Field	Description
	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
28 TRUSTED_KEY	This field indicates that the key was randomly generated by the EdgeLock enclave and never left the EdgeLock enclave. 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 (when generated inside ELE, has plaintext reads/writes disabled, no_import/export property is set and the LOCK property is set)
27-17	Reserved
16 NO_IMPORT/EXPORT	This field disables key import/export using all EdgeLock enclave key blob types except EdgeLock enclave die unique blob. EdgeLock enclave 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

Table continues on the next page...

Table continued from the previous page...

Field	Description
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 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

Table continues on the next page...

Table continued from the previous page...

Field	Description
	bit1: MAC bit2: AEAD bit3: Asymmetric sign/verify bit4: KDF bit5: Reserved bit6: Reserved bit7: Reserved bit8: Reserved bit9: Reserved

Table 13 shows the key properties used in each command. The following identifiers are used in this table:

- X: The command evaluates this key property.
- W1: The command writes into this key property. The written value is determined by command.
- W2: The command writes user value into this key property.
- R: The command reads this key property.

Table 13. Command key properties usage

Command	Bit field														
	31	30-29	28	16	15	14	13	12	11	10	4	3	2	1	0
CIPHER_ONE_GO		X							X	X					X
AEAD_ONE_GO		X							X	X			X		
MAC_ONE_GO		X												X	
ASYMMETRIC_SIGN		X						X				X			
ASYMMETRIC_VERIFY		X					X					X			
DERIVE_KEY	X	X									X				
ASYMMETRIC_DH_DERIVE_KEY	X	X									X				
KEY_STORE_SET_KEY	X	X				X									
KEY_STORE_GET_KEY		X			X										
KEY_STORE_EXPORT_KEY		X		X											
KEY_STORE_IMPORT_KEY	X/W1	X		X/W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1	W1
KEY_STORE_GENERATE_KEY	X	X	W1		X	X									

Table continues on the next page...

Table 13. Command key properties usage (continued)

Command	Bit field														
	31	30-29	28	16	15	14	13	12	11	10	4	3	2	1	0
KEY_STORE_OPEN_KEY															
KEY_STORE_ERASE_KEY	X	X													
KEY_OBJECT_INIT		W1													
KEY_OBJECT_ALLOCATE_HANDLE	W2	X		W2	W2	W2	W2	W2	W2	W2	W2	W2	W2	W2	W2
KEY_OBJECT_GET_HANDLE		X													
KEY_OBJECT_GET_PROPERTIES	R	X/R	R	R	R	R	R	R	R	R	R	R	R	R	R
KEY_OBJECT_SET_PROPERTIES	X/W2	X		W2	W2	W2	W2	W2	W2	W2	W2	W2	W2	W2	W2
KEY_OBJECT_FREE	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.

In the above table:

- X: The command evaluates the key property.
- R: The command reads this key property.
- W1: The command writes to this key property. The written value is determined by the command.
- W2: The command writes user value to this key property.

6.3 Import/export keys

The EdgeLock enclave supports the following key blob types for import/export operation:

- Import/export using die unique key
- Bluetooth LE key export
- EdgeLock enclave to EdgeLock enclave import/export

6.3.1 Import/export using die unique key

The EdgeLock enclave 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 exported key blobs can be stored in host non-volatile memory without exposing plaintext key data outside of the EdgeLock enclave. After the system is powered up again, all stored keys can be imported back into EdgeLock enclave. The blob contains all key information including key type, key properties and key data. Because the blob is encrypted by the die unique key, the exported key blobs cannot

be reused on a different device. The import/export using die unique key is implicitly enabled and cannot be disabled by the NO_IMPORT/EXPORT key property.

6.3.2 Bluetooth LE key export

The NBU and ELE are connected together via Private Key Bus (PKB). The ELE can generate random NBU_DKEY_SK and NBU_DKEY_IRK keys, which are sent via PKB into the NBU. These keys are generated and sent into NBU by KEY_STORE_OPEN_KEY command. When these keys are created, the ELE can create key blobs ESK, EIRK encrypted by NBU_DKEY_SK and NBU_DKEY_IRK keys. The SK and IRK keys can be stored in any key object in the key store. The host software can send ESK, EIRK blobs into NBU when needed. This mechanism provides secure key transfer from ELE into NBU.

Any KEY_STORE_OPEN_KEY command call generates new random NBU_DKEY_x key. So all blobs created prior to this command call become invalid, and SK and IRK keys have to be exported again.

6.3.3 Bluetooth LE 5.x secure connections key generation function f5

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-CMACT (Counter = 1 || keyID || N1 || N2|| A1|| A2 || Length = 256)

The DERIVE_KEY command implements this function as two-step operation:

Step1, Calculate the MacKey as MacKey = AES-CMACT (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.

Step2, Calculate the LTK key similarly.

The LTK key is calculated as LTK = AES-CMACT (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 calculations are fully independent and can be executed in any order. See Bluetooth LE 5.x specification for more information about DHKey and salt data calculation .

6.3.4 EdgeLock enclave to EdgeLock enclave import/export

The EdgeLock enclave to EdgeLock enclave 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 EdgeLock enclave to EdgeLock enclave blob key derivation. The exported key blob can be sent into a different device and imported into its EdgeLock enclave. For more details, refer to EdgeLock enclave to EdgeLock enclave key exchange.

6.4 EdgeLock enclave to EdgeLock enclave key exchange

Some applications require exchanging keys among different nodes/devices. The EdgeLock enclave supports a secure mechanism to exchange keys between two EdgeLock enclave subsystems without exposing plaintext data out of any EdgeLock enclave subsystem. This mechanism is supported by two EdgeLock enclave commands:

- ASYMMETRIC_DH_DERIVE_KEY command using EdgeLock enclave to EdgeLock enclave blob key derivation algorithm
- KEY_STORE_IMPORT_KEY/KEY_STORE_EXPORT_KEY with the key blob type parameter configured as EdgeLock enclave to EdgeLock enclave blob.

The EdgeLock enclave to EdgeLock enclave 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 EdgeLock enclave to EdgeLock enclave key blob. The complete key exchange between two EdgeLock enclave subsystems is shown in the following table:

EdgeLock enclave A	EdgeLock enclave 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 EdgeLock enclave B	Send public part of the key to EdgeLock enclave A
Execute EdgeLock enclave to EdgeLock enclave key derivation algorithm	Execute EdgeLock enclave to EdgeLock enclave key derivation algorithm
Export selected key using EdgeLock enclave to EdgeLock enclave export command and send it to EdgeLock enclave B	Import received blob into EdgeLock B via EdgeLock enclave to EdgeLock enclave import command
Now both EdgeLock enclave subsystems share the same key without exposing plaintext key data out of any EdgeLock enclave	

Chapter 7
ELE Software Architecture and API

7.1 Software Architecture and API Examples

The software abstractions of EdgeLock enclave services are defined and used in the form of a standard set of application-callable API functions. In the context of the software application programming interfaces (APIs), the EdgeLock enclave Security SubSystem is simply referenced as the SSS. Accordingly, the software interface detailed here is called the SSS API.

As the SSS is not customer programmable, to use the SSS-provided services, the user programmable host core must initialize, and access said services by interacting with the SSS through a specified interface. This interface is exposed as a low-level Application Programming Interface that operates with the SSS Messaging Unit (ELE_MU). Serialization of the SSS API is also defined as the Security Subsystem Communication Protocol (SSCP).

Selected requirements for the SSS API include:

- Stateful function context is kept on the host side and is transferred to/from subsystem for stateful operation
- Session context to establish a virtual connection between an application/user context and specific security subsystem and functions
- Key store (secure storage of persistent and transient keys and other attributes)
- Key management (key revocations, key usage constraints)
- Crypto operations (symmetric cipher, authenticated encryption with additional data, message digest, message authentication code, asymmetric crypto, key derivation)
- Integrated with mbedTLS in MCUXpresso SDK

The SSS API is intended to easily integrate with generic software crypto libraries. As an out of box example, the MCUXpresso SDK delivers a version of Arm mbedTLS library integrated with the SSS API, thus enabling the EdgeLock enclave in applications with TLS/DTLS protocol.

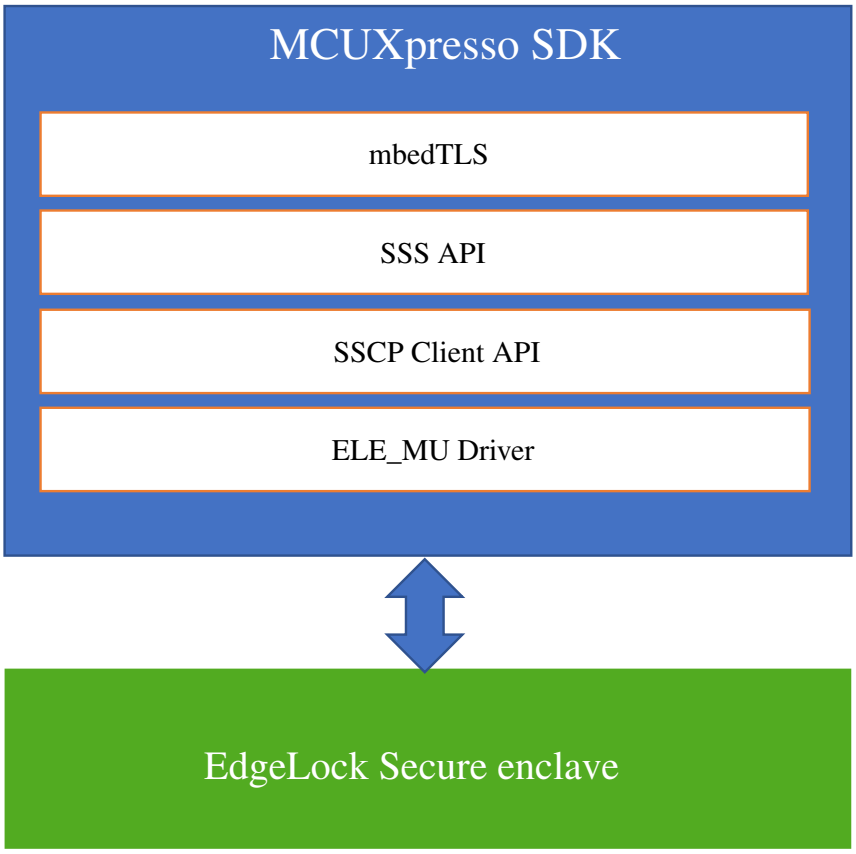


Figure 10. Host SW Security Stack

7.1.1 AES128-ECB encryption examples

In this example, consider the operation of performing a 128-bit AES encryption on a single 16-byte block of data using ECB (Electronic Code Book) mode, the simplest symmetric key mode of operation.

At https://tls.mbed.org/api/aes_8h.html, the Arm mbed site, the following API is defined:

```
int mbedtls_aes_crypt_ecb
(
 mbedtls_aes_context  *ctx,
 int                  mode,
 const unsigned char  input[16],
 unsigned char        output[16]
)
```

This function performs an AES single-block encryption or decryption operation.

It performs the operation defined in the mode parameter (encrypt or decrypt) on the input data buffer defined in the input [*] pointer.

mbedtls_aes_init(), and either mbedtls_aes_setkey_enc() or mbedtls_aes_setkey_dec() must be called before the first call to this API with the same context.

Table 14. mbedtls_aes_crypt_ecb parameters

Parameters	Description
ctx	AES context to use for encryption and decryption It must be initialized and bound to a key
mode	AES operation: MBEDTLS_AES_ENCRYPT (1), MBEDTLS_AES_DECRYPT (0)
input	Buffer holding the input data It must be readable and at least 16 bytes long
output	Buffer where the output data will be written Must be writeable and at least 16 bytes long
ReturnValues	
0	Operation completed successfully

```
sss_status_t sss_cipher_one_go
            (
              sss_symmetric_t *context,
              uint8_t         *iv,
              size_t          ivLen,
              const uint8_t   *srcData,
              uint8_t         *destData,
              size_t          dataLen
             )
```

This function performs an assortment of symmetric key calculations including an AES single-block encryption or decryption operation. Since this function supports a superset of the required AES_ECB computation, some of the input parameters are unused in this instance.

Table 15. status_t sss_cipher_one_go parameters

Parameters	Description
context	Pointer to the symmetric crypto context Must be initialized and bound to a key
iv	Pointer to the symmetric operation Initialization Vector Unused for AES_ECB
ivLen	Length of the Initialization Vector [bytes] Unused for AES_ECB
srcData	Pointer to buffer containing the input data Must be readable and at least 16 bytes long
destData	Pointer to buffer containing the output data Must be writeable and at least 16 bytes long

Table continues on the next page...

Table 15. status_t sss_cipher_one_go parameters (continued)

Parameters	Description
dataLen	Size of the input and output data buffer [bytes]
ReturnValues	
kStatus_SSS_Success	Operation completed successfully (0)
kStatus_SSS_Fail	Operation failed (!0)

The ELE_MU driver running on the host processor executes a series of register writes to the ELE_MUA_TRn registers to transmit the service request's parameters to the EdgeLock enclave. When the EdgeLock enclave execution is complete, the ELE_MUB_TRn registers are written with status information which is retrieved by the host processor by reading the ELE_MUA_RRn registers.

The mbedTLS arguments and the SSS API are designed to very efficiently connect with the EdgeLock enclave's ELE_MU hardware interface to maximize performance of the crypto service request's initiation and termination response.

7.1.2 AES128-CBC encryption/decryption example

This is an example of a more complex block cipher mode, consider the mbedTLS and SSS APIs for an AES-CBC (Cipher Block Chaining) service request:

```
int mbedtls_aes_crypt_cbc
                    (
                     mbedtls_aes_context  *ctx,
                     int                  mode,
                     size_t               length,
                     unsigned char        iv[16],
                     const unsigned char  input[16],
                     unsigned char        output[16]
                    )
```

This function performs an AES-CBC encryption or decryption operation on full blocks.

It performs the operation defined in the mode parameter (encrypt or decrypt) on the input data buffer defined in the input parameter.

It can be called as many times as needed, until all the input data is processed. mbedtls_aes_init(), and either mbedtls_aes_setkey_enc() or mbedtls_aes_setkey_dec() must be called before the first call to this API with the same context.

NOTE

This function operates on full blocks, that is, the input size must be a multiple of the AES block size of 16 bytes.

Upon exit, the content of the IV is updated so that you can call the same function again on the next block(s) of data and get the same result as if it was encrypted in one call. This allows a "streaming" usage, that is, the cipher block chaining supports the hardware's ability to continue to add new data blocks into the crypto calculation as the new data becomes available.. If you need to retain the contents of the IV, you should either save it manually or use the cipher module instead.

Table 16. mbedtls_aes_crypt_cbc parameters

Parameters	Description
ctx	AES context to use for encryption and decryption It must be initialized and bound to a key

Table continues on the next page...

Table 16. mbedtls_aes_crypt_cbc parameters (continued)

mode	AES operation: MBEDTLS_AES_ENCRYPT (1) or MBEDTLS_AES_DECRYPT (0)
length	The length of the input data in bytes. This must be a multiple of the block size (16 bytes).
iv	Initialization vector (updated after use). It must be a readable and writeable buffer of 16 bytes.
input	Buffer holding the input data It must be readable and of size of length bytes.
output	Buffer where the output data will be written It must be writeable and of size length bytes.
Return Values	
0	Operation completed successfully
!0	MBEDTLS_ERR_AES_INVALID_INPUT_LENGTH on failure

The SSS API is the same one previously used in the AES_ECB function call.

```
sss_status_t sss_cipher_one_go
                    (
                      sss_symmetric_t *context,
                      uint8_t         *iv,
                      size_t          ivLen,
                      const uint8_t   *srcData,
                      uint8_t         *destData,
                      size_t          dataLen
                     )
```

For this function, all the input parameters are used (unlike the simpler AES_ECB call).

To summarize, the combination of the ELE_MU hardware module plus the security software stack defining the interface from the mbedTLS APIs through the SSS API and eventually the EdgeLock enclave software provides an efficient, secure mechanism to process host crypto service requests.

7.2 EdgeLock enclave service format

EdgeLock enclave services are available via command messages communicated using a messaging unit ELEMUA. Every EdgeLock enclave service works on request -> response basis. It means that every service includes a command message to write to the ELEMUA_TRn registers to request the EdgeLock enclave service. After the service is completed, the EdgeLock enclave replies every command request with a response message, which can be read via ELEMUA_RRn registers, see the following figure.

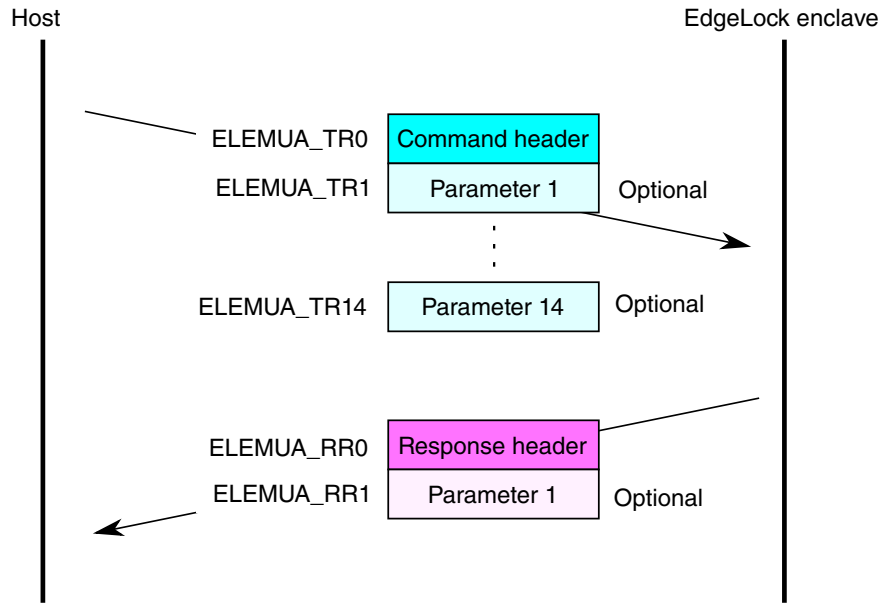


Figure 11. Host to EdgeLock enclave communication

7.3 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 12. 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 13. Command Header format

The fields in header above are defined as:

Field Name	Description
Command ID	8-bit command field defining the EdgeLock enclave 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 EdgeLock enclave 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 EdgeLock enclave is ready to receive a new command again.

7.4 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 14. 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 15. Response Header format

The fields in the header above are defined as:

Field Name	Description
Command ID	8-bit command field defining the EdgeLock enclave service that was performed. The command field is "returned" to the host in the response header. This allows the host software to precisely associate EdgeLock enclave 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)

7.5 Asynchronous messages

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

7.5.1 INITIALIZATION_FINISHED response message

This message is asynchronously sent by EdgeLock enclave after RESET and it consists of two 32-bit words. The first 32-bit word is written into ELEMUA_RR0 at the beginning of EdgeLock enclave initialization sequence. The second 32-bit word is written into ELEMUA_RR1 after EdgeLock enclave initialization is completed. After these two words are received by a host, the EdgeLock enclave 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 EdgeLock enclave is already ready for operation. However, if the EdgeLock enclave is reset by the user application, the EdgeLock enclave sends this message again during initialization sequence and user needs to process it appropriately.

Table 17. 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

7.5.2 ABORT response message

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

Table 18. ABORT response message format

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

Table 19. 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

7.5.3 SECURITY_VIOLATION response message

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

Table 20. SECURITY_VIOLATION response message format

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

7.6 EdgeLock enclave commands

The EdgeLock enclave provides the following commands.

7.6.1 PING command

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

Table 21. 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.	

7.6.2 OPEN_SESSION command

This command opens a session between the host and the EdgeLock enclave. 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 22. OPEN_SESSION command format

Command ID	0x13	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	2/1
Security level	Any		
Command message format			
Parameter	**ELEMUA_TRn number**	**Description**	
Message header	0	0x13020017	
Subsystem type	1	0x00000002: EdgeLock enclave	
Session ID	2	Session ID	

Table continues on the next page...

Table 22. OPEN_SESSION command format (continued)

		Because the current EdgeLock enclave implementation supports only one opened session at a time, this parameter can be kept as zero.
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.

7.6.3 CLOSE_SESSION command

This command closes an opened session.

Table 23. 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	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x1400xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	

7.6.4 CONTEXT_FREE command

This command closes an opened operation context. This command can be used to close any operation type context.

Table 24. 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			

Table continues on the next page...

Table 24. CONTEXT_FREE command format (continued)

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

7.6.5 SYMMETRIC_CONTEXT_INIT command

This command initializes context for a symmetric operation. Only one SYMMETRIC context can be opened at a time.

Table 25. 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	
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	

7.6.6 CIPHER_ONE_GO command

This command performs symmetric encryption or decryption in one step.

Table 26. 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. Maximum data length is 65535 bytes. **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.	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x2300xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	

7.6.7 AEAD_CONTEXT_INIT command

This command initializes context for AEAD operation. Only one AEAD context can be opened at a time.

Table 27. 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	

7.6.8 AEAD_ONE_GO command

This command performs an authenticated symmetric encryption or decryption in one step. This operation is non-interruptible.

Table 28. 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	

Table continues on the next page...

Table 28. AEAD_ONE_GO command format (continued)

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

7.6.9 DIGEST_CONTEXT_INIT command

This command initializes context for message digest operation. Four digest contexts can be opened at the same time.

Table 29. 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**	

Table continues on the next page...

Table 29. DIGEST_CONTEXT_INIT command Format (continued)

Message header	0	0x2C030017
Session ID	1	Session ID
Digest algorithm	2	Hash algorithm: 0x00000014: SHA-1 0x0000001C: SHA2-224 0x00000020: SHA2-256 0x00000030: SHA2-384 0x00000040: SHA2-512
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

7.6.10 DIGEST_ONE_GO command

This command performs message digest operation in one step. This operation is non-interruptible.

Table 30. 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		
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**	

Table continues on the next page...

Table 30. DIGEST_ONE_GO command format (continued)

Message header	0	0x2F01xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Digest length	1	Number of bytes written into digest buffer

7.6.11 DIGEST_INIT command

This command initializes a block (init/update/finish) message digest operation. This operation can be interrupted by any other digest operation.

Table 31. 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	

7.6.12 DIGEST_UPDATE command

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 32. 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	

Table continues on the next page...

Table 32. DIGEST_UPDATE command format (continued)

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

7.6.13 DIGEST_FINISH command

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 33. 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			
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	

7.6.14 MAC_CONTEX_INIT command

This command initializes context for a message authentication code (MAC) operation. Only one MAC context can be opened at a time.

Table 34. 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 0x0000000E: HMAC with SHA256	
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	

7.6.15 MAC_ONE_GO command

This command performs a message authentication code (MAC) operation in one step. This operation is non-interruptible.

Table 35. 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 Maximum data length is 65535 bytes in case of CMAC with AES	

Table continues on the next page...

Table 35. MAC_ONE_GO command format (continued)

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

7.6.16 ASYMMETRIC_CONTEXT_INIT command

This command initializes context for an asymmetric (sign/verify) operation. Only one ASYMMETRIC context can be opened at a time.

Table 36. ASYMMETRIC_CONTEXT_INIT command format

Command ID	0x37	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	4/1
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	0x37040017	
Session ID	1	Session ID	
Key object ID	2	Key object ID used for sign/verify digest operation	
Algorithm	3	Algorithm used for sign/verify digest operation: 0x0000001F: ECDSA with P-224 0x00000020: ECDSA with P-256 0x00000021: ECDSA with P-384 0x00000022: ECDSA with P-521 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:	

Table continues on the next page...

Table 36. ASYMMETRIC_CONTEXT_INIT command format (continued)

		0x3C3C: Command success 0xC3C3: Command fail
Operation context ID	1	Sign/verify digest operation context ID

7.6.17 ASYMMETRIC_SIGN command

This command performs an asymmetric sign operation.

Table 37. 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	
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	

7.6.18 ASYMMETRIC_VERIFY command

This command performs an asymmetric verify operation.

Table 38. 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			

Table continues on the next page...

Table 38. ASYMMETRIC_VERIFY command format (continued)

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

7.6.19 DERIVE_KEY_CONTEXT_INIT command

This command initializes context for a Diffie-Hellman key exchange operation. Only one DERIVE_KEY context can be open at a time.

Table 39. DERIVE_KEY_CONTEXT_INIT command format

Command ID	0x40	Command data size [ELEMUA_TRn/ ELEMUA_RRn]	4/1
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	Algorithm used for key derivation: 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). 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.	

Table continues on the next page...

Table 39. DERIVE_KEY_CONTEXT_INIT command format (continued)

		0x00000030: X25519 (Diffie-Hellman key exchange using Curve25519) 0x00000040: EdgeLock enclave to EdgeLock enclave blob key derivation 0x00000050: Bluetooth LE 5.x secure Connections key generation function f5
Operation mode	4	0x00000004: 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

7.6.20 DERIVE_KEY command

This command performs key derivation function. See Bluetooth LE 5.x secure connections key generation function f5 for more details.

Table 40. DERIVE_KEY command format

Command ID	0x3F	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	4/1
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	
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	0x3C01xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	
Reserved	1	Reserved. Always returns 0	

7.6.21 ASYMMETRIC_DH_DERIVE_KEY command

This command performs a Diffie-Hellman key exchange operation.

Table 41. 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 EdgeLock enclave to EdgeLock enclave 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 0xC3C3: Command fail	
Reserved	1	Reserved. Always returns 0	

7.6.22 TUNNEL_CONTEXT_INIT command

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)
- EdgeLock enclave firmware upload (authenticate, decrypt and run EdgeLock enclave firmware)

Table 42. TUNNEL_CONTEXT_INIT command format

Command ID	0x3D	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	2/1

Table continues on the next page...

Table 42. TUNNEL_CONTEXT_INIT command format (continued)

Security level	Any	
Command message format		
Parameter	**ELEMUA_TRn number**	**Description**
Message header	0	0x3D020017
Session ID	1	Session ID
Tunnel type	2	Tunnel type: 0x00000020: Main boot image authentication 0x00000021: SB3 file authentication 0x00000022: EdgeLock enclave firmware upload
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

7.6.23 TUNNEL_REQUEST command

This command performs a tunnel operation.

Table 43. TUNNEL_REQUEST command format

Command ID	0x3E	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	3/1
Security level	The same or higher security level used with 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 in bytes	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x3E01xxxx, where xxxx: 0x3C3C: Command success	

Table continues on the next page...

Table 43. TUNNEL_REQUEST command format (continued)

		0xC3C3: Command fail
Duplicate command result	1	0x0A0B0C0D: Tunnel is executed successfully Any other values: Tunnel is failed to be executed

7.6.23.1 Main boot image authentication (Tunnel type = 0x20)

For tunnel type 0x20 the tunnel data (ELEMUA_TR2) contains the start address of main 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 main boot image authentication command is a one go command.

7.6.23.2 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 EdgeLock enclave 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 EdgeLock enclave firmware is decrypted and uploaded into internal EdgeLock enclave RAM. After all data blocks are successfully decrypted, EdgeLock enclave 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.

7.6.23.3 EdgeLock enclave 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 EdgeLock enclave 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 EdgeLock enclave in one step. This command can be used if the whole firmware is available before the upload is started.

The command returns success if EdgeLock enclave firmware was successfully loaded and executed, or fail in case of failed authentication, decryption, or another error.

7.6.24 KEY_STORE_ INIT command

This command initializes the user key store. Only one key store context can be opened at a time.

Table 44. 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	

Table continues on the next page...

Table 44. KEY_STORE_CONTEXT_INIT command format (continued)

Session ID	1	Session ID
User key store ID	2	User key store ID: 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

7.6.25 KEY_STORE_FREE command

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 45. 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			
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	

7.6.26 KEY_STORE_SET_KEY command

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:

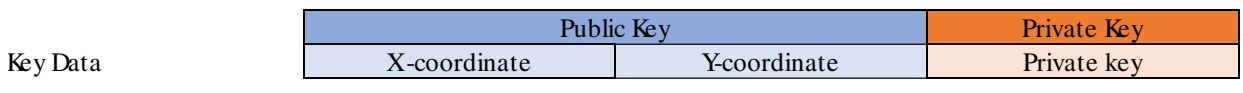


Figure 16. Asymmetric key format

Table 46. 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	
Response message format			
Parameter	**ELEMUA_RRn number**	**Description**	
Message header	0	0x4c00xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: command fail	

7.6.27 KEY_STORE_GET_KEY command

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 47. 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	

Table continues on the next page...

Table 47. KEY_STORE_GET_KEY command format (continued)

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		
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

7.6.28 KEY_STORE_EXPORT_KEY command

This command exports a key/key pair as key blob if allowed by key properties (see chapter Key Object Properties).

Table 48. 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	

Table continues on the next page...

Table 48. KEY_STORE_EXPORT_KEY command format (continued)

Blob data	3	Start address of buffer for blob data
Blob data length	4	Blob data buffer size in bytes
Key blob type	5	Key blob type: 0x00000001: EdgeLock enclave die unique blob 0x00000002: EdgeLock enclave to EdgeLock enclave blob 0x00000003: NBU ESK blob 0x00000004: NBU EIRK blob **NOTE** Refer to Bluetooth LE key export for details of NBU blob.
Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
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

7.6.29 KEY_STORE_IMPORT_KEY command

This command imports a key/key pair blob into the EdgeLocK enclave internal key store if allowed by key properties (see chapter Key Object Properties).

Table 49. 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	
Key blob type	5	Key blob type: 0x00000001: EdgeLock enclave die unique blob 0x00000002: EdgeLock enclave to EdgeLock enclave blob	

Table continues on the next page...

Table 49. KEY_STORE_IMPORT_KEY command format (continued)

Response message format		
Parameter	**ELEMUA_RRn number**	**Description**
Message header	0	0x7800xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

7.6.30 KEY_STORE_GENERATE_KEY command

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 50. 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	

7.6.31 KEY_STORE_OPEN_KEY command

This command sends an internal EdgeLock enclave key to another peripheral via private secure key bus. This command currently supports NPX keys only.

Table 51. KEY_STORE_OPEN_KEY command format

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: 0x80000007: NPX keys	
Response message format			
Parameter	ELEMUA_RRn number	Description	
Message header	0	0x4F00xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail	

7.6.32 KEY_STORE_ERASE_KEY command

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 52. 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	

7.6.33 KEY_STORE_GET_PROPERTY command

This command returns the configured and actual key store property (key store data size, maximum key objects number).

Table 53. 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	
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	

7.6.34 KEY_OBJECT_INIT command

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 54. 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			

Table continues on the next page...

Table 54. KEY_OBJECT_INIT command format (continued)

Parameter	ELEMUA_RRn number	Description
Message header	0	0x4101xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail
Key object ID	1	Key object ID

7.6.35 KEY_OBJECT_ALLOCATE_HANDLE command

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 55. 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	

Table continues on the next page...

Table 55. KEY_OBJECT_ALLOCATE_HANDLE command format (continued)

		0x00000040: Asymmetric ECC NIST-P 0x00000050: Asymmetric ECC MONTGOMERY 0x00000051: Asymmetric ECC TWISTED EDWARDS
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

7.6.36 KEY_OBJECT_GET_HANDLE command

This command returns the key object ID if a user key ID is found in the key store.

Table 56. 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	

Table continues on the next page...

Table 56. KEY_OBJECT_GET_HANDLE command format (continued)

User key ID	1	User key ID: 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 command 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

7.6.37 KEY_OBJECT_GET_PROPERTIES command

This command returns key object properties.

Table 57. 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	

7.6.38 KEY_OBJECT_SET_PROPERTIES command

This command sets key object properties.

Table 58. 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 58. 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

7.6.39 KEY_OBJECT_FREE command

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 59. 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 59. 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

7.6.40 MGMT_CONTEXT_INIT command

This command initializes context for EdgeLock enclave management commands. Only one management context can be opened at a time.

Table 60. 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	

7.6.41 MGMT_ADVANCE_LIFECYCLE command

This command changes life cycle state into the next state. The next allowed state depends on lifecycle state machine.

Table 61. MGMT_ADVANCE_LIFECYCLE command format

Command ID	0x60	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	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.

7.6.42 MGMT_FUSE_PROGRAM command

This command performs EdgeLock enclave fuse programming. The fuse programming requires the CPU_CLK clock frequency in MHz, which is provided as option data.

Table 62. 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		

Table continues on the next page...

Table 62. MGMT_FUSE_PROGRAM command format (continued)

Command message format		
Parameter	**ELEMUA_TRn number**	**Description**
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 63. 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	OEM_ENABLEMENT_TOKEN_LOCK	3	R/W
8	DCFG_CC_SOCU_L1_LOCK	3	R/W
9	DCFG_CC_SOCU_L2_LOCK	3	R/W
10	LIFECYCLE	8	RO

Table continues on the next page...

Table 63. Fuses ID list (continued)

Fuse ID	Fuse name	Fuse size (bits)	Access
11	DBG_EN_LOCK	1	R/W
12	DBG_AUTH_DIS	1	R/W
13	TZM_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	OEM_ENABLEMENT_TOKEN	256	R/W
34	DCFG_CC_SOCU_L1	32	R/W
35	DCFG_CC_SOCU_L2	32	R/W
36	SOC_VER_CNT	512	R/W
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

Table continues on the next page...

Table 63. Fuses ID list (continued)

Fuse ID	Fuse name	Fuse size (bits)	Access
44	RADIO_VER_CNT_VIRTUAL	n/a	R/W
45	SNT_VER_CNT_VIRTUAL	n/a	R/W
46	CM33_BOOTLOADER_VER_CNT_VIRTUAL	n/a	R/W

7.6.43 MGMT_FUSE_READ command

This command performs EdgeLock enclave fuse read. The fuse reading requires the CPU_CLK clock frequency in MHz, which is provided as option data.

Table 64. 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 63)	
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.	
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	

7.6.44 MGMT_GET_LIFECYCLE command

This command returns actual device lifecycle.

Table 65. 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.	

7.6.45 MGMT_GET_PROPERTY command

This command returns security device properties, which are managed by EdgeLock enclave. For successful command operation, the correct property data length must be provided.

Table 66. 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	

Table continues on the next page...

Table 66. MGMT_GET_PROPERTY command format (continued)

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: EdgeLock enclave 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) 0x00000070: Public portion of ID authentication key (data length = 64 bytes) 0x00000080: Radio image owner (data length = 4 bytes), the expected returns are: • 0x00: NXP is owner • Non-zero value: OEM is owner 0x00000090: UUID (data length = 16 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

7.6.46 MGMT_SET_PROPERTY command

This command sets security device properties, which are managed by EdgeLock enclave. For successful command operation, the correct property data length must be provided.

Table 67. MGMT_SET_PROPERTY command format

Command ID	0x71	**Command data size [ELEMUA_TRn/ ELEMUA_RRn]**	4/0

Table continues on the next page...

Table 67. MGMT_SET_PROPERTY 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	0x71040017
Management context ID	1	Management context ID
Property ID	2	Property ID: 0x11: EdgeLock enclave RAM Power Control (data length = 4 bytes) Source data: **Bit0:** 0: PKC RAM disabled 1: PKC RAM enabled **Bit1:** 0: EdgeLock enclave firmware RAM disabled 1: EdgeLock enclave firmware RAM enabled **NOTE** The PKC RAM contains part of the key store. If key store is used, it must be deleted using KEY_STORE_ALLOCATE FREE command before PKC RAM is disabled. If EdgeLock enclave firmware RAM executes firmware, the FW_BACK_TO_ROM command must be called before the firmware RAM is disabled. The FW_BACK_TO_ROM command is part of the downloaded firmware. Disabling any of RAM will delete its content.
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

7.6.47 MGMT_SET_HOST_ACCESS_PERMISSION command

This command controls secure/non-secure and privilege/user bus attributes, which host uses for all bus transactions performed during next EdgeLock enclave command execution. It can be executed any time the EdgeLock enclave 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 EdgeLock enclave uses masquerading technique to inherit bus transaction security level for all data accesses to the SoC memory. It means that EdgeLock enclave uses the same bus security attribute as the host, which writes command into ELEMUA_TRn register. Said another way, if EdgeLock enclave command was initiated by secure privilege host, then the EdgeLock enclave behaves as secure privilege bus owner for all bus transactions performed within the current command execution. If the EdgeLock enclave command was initiated by non-secure privilege host, the EdgeLock enclave 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 EdgeLock enclave 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 EdgeLock enclave 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 68. 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**	

Table continues on the next page...

Table 68. MGMT_SET_HOST_ACCESS_PERMISSION command format (continued)

Message header	0	0x7000xxxx, where xxxx: 0x3C3C: Command success 0xC3C3: Command fail

7.6.48 MGMT_SET_RETURN_FA_MODE command

This command sets OEM/NXP field return analysis modes.

Table 69. 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	

7.6.49 MGMT_GET_RANDOM command

This command returns a random number of the requested length. This command can be executed any time the EdgeLock enclave is ready to receive new command and without opening EdgeLock enclave 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 70. 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.	

7.6.50 MGMT_CLEAR_ALL_KEYS command

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 71. 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 71. 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

7.6.51 UKNOWN_COMMAND response message

The EdgeLock enclave sends this message in case the command ID is not recognized or not supported.

Table 72. 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.

7.7 EdgeLock enclave code example

This code example demonstrates EdgeLock enclave usage for AES CBC operation. The AES CBC operation is performed in following steps:

1. Open EdgeLock enclave session
2. Create key store
3. Create and allocate key object
4. Set the key
5. Initialize AES CBC operation context
6. Perform AES CBC operation
7. Close 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 8
Flash Memory Controller (FMC) with NVM PRINCE Encryption and Decryption (NPX)

8.1 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
- An NVM Prince Counter Mode (CM) module called NPX to optionally 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 next table shows the supported read/write operations.

Flash memory type	Read	Write
Program, IFR, IFR1 flash memory	8-bit, 16-bit, and 32-bit reads	16-byte and 128-byte writes

The FMC provides quick access to flash memory using 3 separate acceleration mechanisms:

- A flash-phrase-sized buffer holds the most recently accessed flash phrase
- An optional flash-phrase-sized speculation buffer can prefetch the next flash phrase
- An optional 4-way, set-associative cache can store previously accessed flash phrases

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.

8.1.1 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 FMC's 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, used once for aligned 16-byte flash write phrases or used 8 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-word write of an aligned phrase in order.
- Acceleration of data transfer from flash memory to the device:

- A flash-phrase-sized buffer that holds the current decrypted flash phrase fetched due to a FMC read request. Subsequent FMC read requests *that hit in the current 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 invalidate the current and speculation buffers
- The flash cache has input controls:
  - to disable instruction caching
  - to disable operand caching
  - to disable all caching
  - to clear the cache
- The size of the flash cache in bytes is calculated as follows:

  flash cache size = [number of ways] x [number of sets] x [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) x (1 set) x (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.

8.2 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.

8.2.1 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, the FMC is disabled.

8.2.2 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 current and speculation buffers are cleared by reset.
- Prefetch support for data and instructions is enabled.
- The cache is cleared by reset.
- The cache is configured for data or instruction replacement.

8.2.3 Configuration options

The default configuration provides a high degree of flash acceleration, however, 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 cache and buffering controls allow the tuning of resources to suit specific application requirements. The cache and buffer are each controlled individually. The controls enable buffering 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 cache or the single-entry buffer. Likewise, speculation can be enabled or disabled for either type of access.

For best performance, the FMC cache and speculation buffering should be enabled for instruction and/or data fetching. The following is recommended for best performance:

- If the speculation buffer is enabled for both instruction and data speculation, also enable the flash cache for both instruction and data caching.
- If the speculation buffer is enabled for instruction speculation only, also enable the flash cache for at least instruction caching.

8.2.4 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.

All wait states and synchronization delays are handled automatically by the Flash Memory Controller. No direct user configuration is required (or even allowed) to set up the flash wait states.

8.2.5 Speculative reads

The FMC has a single 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/or data accesses. Because many code accesses are sequential, using the speculative prefetch buffer improves performance in most cases.

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.

8.2.6 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.

8.2.6.1 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)

8.2.6.2 NPX block diagram

The NPX is directly connected to the Flash Memory Controller (FMC).

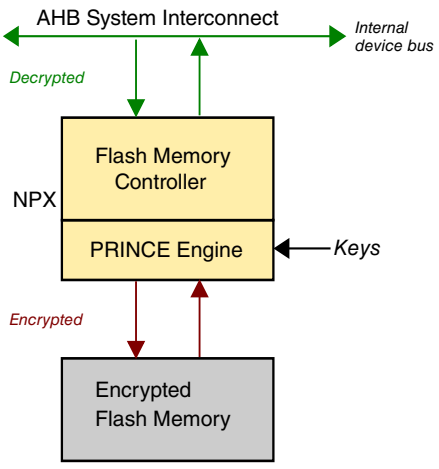


Figure 17. NPX block diagram

8.2.6.3 NPX modes of operation

The NPX operating modes are controlled and reported in the NPX Control Register (NPXCR):

- Global Decryption Enable (GDE) field

- If GDE=101b, then all decryption is disabled.
- If GDE=010b, then for reads that hit a valid context (memory region), NPX performs on-the-fly data decryptions.

- Global Encryption Enable (GEE) field
  - If GEE=101b, then all encryption is disabled.
  - If GEE=010b, then for writes that hit a valid context (memory region), NPX performs on-the-fly data encryptions.

8.2.6.4 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 (CACMSK) before being stored in the cache. Cache data is unmasked when read. To modify CACMSK:

1. Disable the flash cache.
2. Modify CACMSK.
3. Clear the cache.
4. Re-enable the cache.

8.2.6.5 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 REMAP, remapping is enabled when LIM[19:15] == LIM_DP[19:15] and LIM[19:15] is non-zero. LIM[19:15] and LIM_DP[19:15] define the *remap_address*[19:15] for the remapping, providing an address range granularity of 32 KB. For a remapped FMC access to address *access_address*:

Table 73. 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*[19:15]	bank1_base + access_address	bank 1
flash bank 1	bank1_base + *remap_address*[19:15]	*access_address* – bank1_base	bank 0

Note that 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 below:

Table 74. 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	

8.2.6.6 NPX initialization

The operating configuration of the NPX is controlled by programmable bits in NPXCR.

8.2.6.7 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. The NPX does not support memory region overlap because each region has a unique key.

8.2.6.8 Interrupts

This module has no interrupts.

8.3 External signals

The FMC has no external signals.

8.4 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. As a result, if an application is executing flash memory commands, the FMC's current buffer, speculation buffer, and cache might need to be disabled and/or flushed to prevent the possibility of returning stale data.

8.5 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

3-bit pole/anti-pole control fields:

For certain critical control and configuration functions, a special 3-bit pole/anti-pole implementation provides additional fault protection against attempted attacks. In this robust 3-bit implementation, the middle bit is taken as the operative value. When reading a pole/anti-pole field:

- 101b is the negated state
- 010b is the asserted state

When writing a 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.

8.5.1 FMC register descriptions

8.5.1.1 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
8h	NPX Status Register (NPXSR)	32	R	0000_0004h
10h	Flash Cache Obfuscation Mask (CACMSK)	32	W	0000_0000h
20h	Data Remap (REMAP)	32	RW	0000_0000h
40h	Bitmap of Valid Control for Memory Context 0 (VMAPCTX0_WD0)	32	RW	0000_0000h
44h	Bitmap of Valid Control for Memory Context 0 (VMAPCTX0_WD1)	32	RW	0000_0000h
48h	Block Initial Vector for Memory Context 0 (BIVCTX0_WD0)	32	W	0000_0000h
4Ch	Block Initial Vector for Memory Context 0 (BIVCTX0_WD1)	32	W	0000_0000h
50h	Bitmap of Valid Control for Memory Context 1 (VMAPCTX1_WD0)	32	RW	0000_0000h
54h	Bitmap of Valid Control for Memory Context 1 (VMAPCTX1_WD1)	32	RW	0000_0000h
58h	Block Initial Vector for Memory Context 1 (BIVCTX1_WD0)	32	W	0000_0000h
5Ch	Block Initial Vector for Memory Context 1 (BIVCTX1_WD1)	32	W	0000_0000h
60h	Bitmap of Valid Control for Memory Context 2 (VMAPCTX2_WD0)	32	RW	0000_0000h
64h	Bitmap of Valid Control for Memory Context 2 (VMAPCTX2_WD1)	32	RW	0000_0000h
68h	Block Initial Vector for Memory Context 2 (BIVCTX2_WD0)	32	W	0000_0000h
6Ch	Block Initial Vector for Memory Context 2 (BIVCTX2_WD1)	32	W	0000_0000h
70h	Bitmap of Valid Control for Memory Context 3 (VMAPCTX3_WD0)	32	RW	0000_0000h
74h	Bitmap of Valid Control for Memory Context 3 (VMAPCTX3_WD1)	32	RW	0000_0000h
78h	Block Initial Vector for Memory Context 3 (BIVCTX3_WD0)	32	W	0000_0000h
7Ch	Block Initial Vector for Memory Context 3 (BIVCTX3_WD1)	32	W	0000_0000h
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

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
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
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

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
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
1E0h	Memory Region 3 Start Address (MR3STARTADDR)	32	W	0000_0000h
1E4h	Memory Region 3 End Address (MR3ENDADDR)	32	W	0000_01FDh

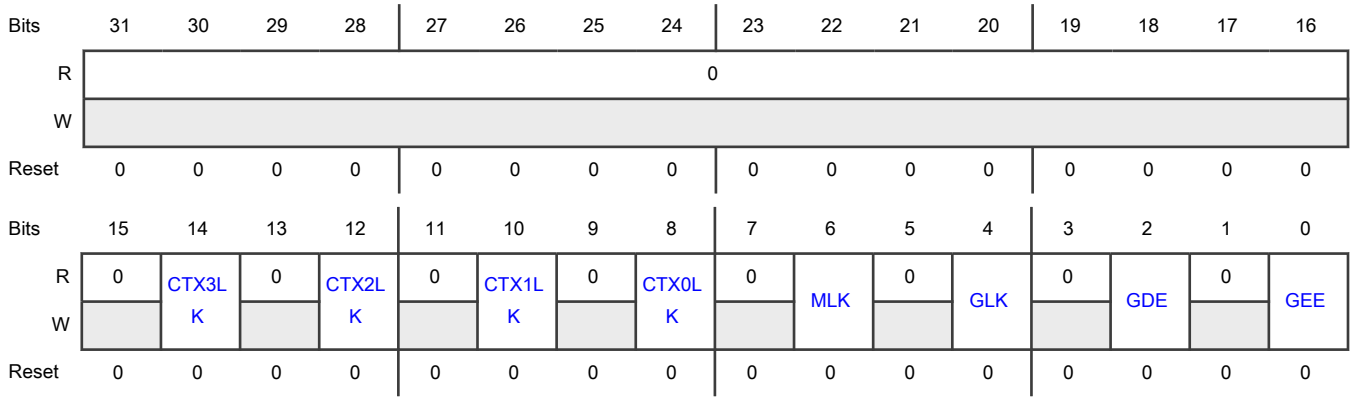
8.5.1.2 NPX Control Register (NPXCR)

Offset

Register	Offset
NPXCR	0h

Function
Contains enables for Global Lock, Context Locks, Mask Lock, Global Decryption and Global Encryption.

Diagram



Fields

Field	Function
31-15 —	This read-only field is reserved and always has the value 0
14 CTX3LK	Lock Enable for Context 3 Affects the register access properties of VMAPCTX3. BIVCTX3 is not affected and remains write-only. Set by software (sticky); cleared only by a reset. 0b - Lock disabled: VMAPCTX3 remains read-write 1b - Lock enabled: cannot write to VMAPCTX3 (becomes read-only)
13 —	This read-only field is reserved and always has the value 0
12 CTX2LK	Lock Enable for Context 2 Affects the register access properties of VMAPCTX2. BIVCTX2 is not affected and remains write-only. Set by software (sticky); cleared only by a reset. 0b - Lock disabled: VMAPCTX2 remains read-write 1b - Lock enabled: cannot write to VMAPCTX2 (becomes read-only)
11 —	This read-only field is reserved and always has the value 0
10 CTX1LK	Lock Enable for Context 1 Affects the register access properties of VMAPCTX1. BIVCTX1 is not affected and remains write-only. Set by software (sticky); cleared only by a reset. 0b - Lock disabled: VMAPCTX1 remains read-write 1b - Lock enabled: cannot write to VMAPCTX1 (becomes read-only)

Table continues on the next page...

Table continued from the previous page...

Field	Function
9 —	This read-only field is reserved and always has the value 0
8 CTX0LK	Lock Enable for Context 0 Affects the register access properties of VMAPCTX0. BIVCTX0 is not affected and remains write-only. Set by software (sticky); cleared only by a reset. 0b - Lock disabled: VMAPCTX0 remains read-write 1b - Lock enabled: cannot write to VMAPCTX0 (becomes read-only)
7 —	This read-only field is reserved and always has the value 0
6 MLK	Mask Lock Enable Set by software (sticky); cleared only by a reset. 0b - Lock disabled. Subsequent reads return 0. 1b - Lock enabled: cannot write to mask. Subsequent reads return 1.
5 —	This read-only field is reserved and always has the value 0
4 GLK	Global Lock Enable Affects the register access properties of VMAPCTX*n*, NPXCR (NPX Control Register), and CACMSK (Flash Cache Obfuscation Mask). BIVCTX*n* is not affected and remains write-only. Set by software (sticky); cleared only by a reset. 0b - Lock disabled. Subsequent reads return 0. 1b - Lock enabled: cannot write to VMAPCTXn, NPXCR, or CACMSK. Subsequent reads return 1.
3 —	This read-only field is reserved and always has the value 0
2 GDE	Global Decryption Enable Set by software; cleared by software; cleared by POR (Power-On Reset). 0b - Global decryption disabled. NPX on-the-fly decryption is globally disabled. Subsequent reads return 0. 1b - Global decryption enabled. NPX on-the-fly decryption is globally enabled. Subsequent reads return 1.
1 —	This read-only field is reserved and always has the value 0

Table continues on the next page...

Table continued from the previous page...

Field	Function
0 GEE	Global Encryption Enable Set by software; cleared by software; cleared by POR (Power-On Reset). 0b - Global encryption disabled. NPX on-the-fly encryption is disabled. Subsequent reads return 0. 1b - Global encryption enabled. NPX on-the-fly encryption is enabled if the flash access hits in a valid memory context. Subsequent reads return 1.

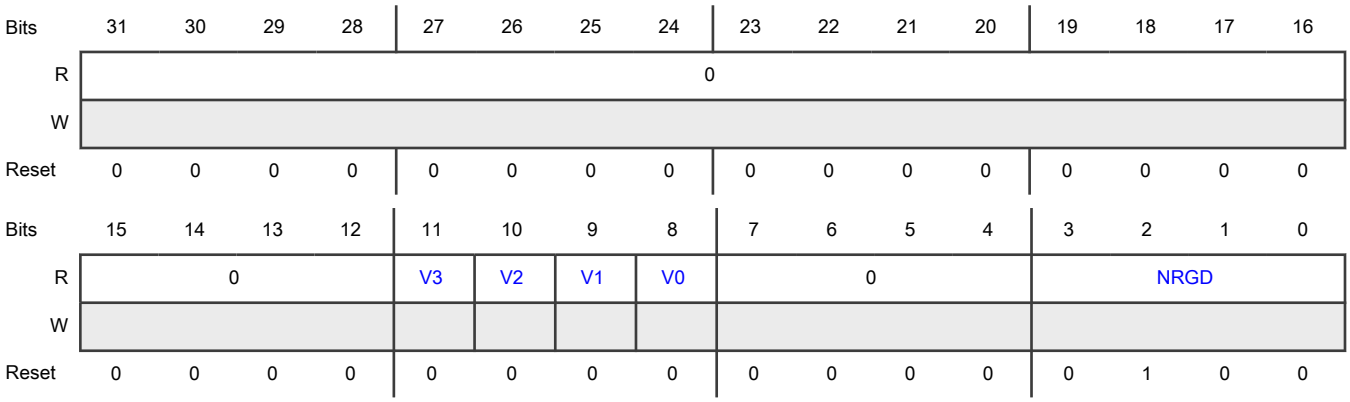
8.5.1.3 NPX Status Register (NPXSR)

Offset

Register	Offset
NPXSR	8h

Function
Contains the Key 0-3 validity flags and the number of supported memory regions (NRGD).

Diagram



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.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	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

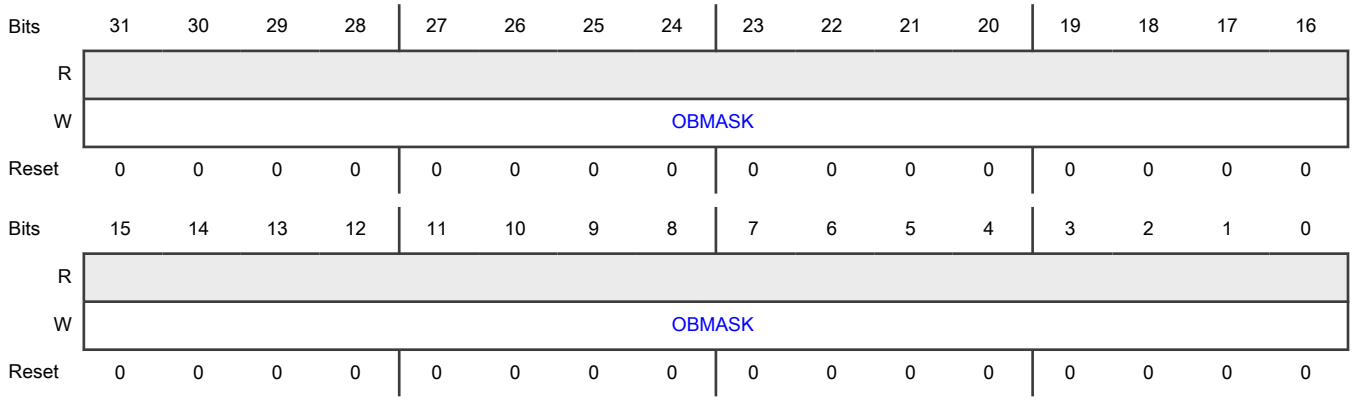
8.5.1.4 Flash Cache Obfuscation Mask (CACMSK)

Offset

Register	Offset
CACMSK	10h

Function
Contains the obfuscation mask for the flash cache.

Diagram



Fields

Field	Function
31-0 OBMASK	Obfuscation Mask

8.5.1.5 Data Remap (REMAP)

Offset

Register	Offset
REMAP	20h

Function

Provides a data remapping mechanism.

To write to REMAP, the remap lock must be disabled (REMAPLK=0). To control the remapping, follow these options:

- To write to LIMDP, write 0x*NNNN*A5A5 (where *NNNN* is data) to REMAP. In this case, bits 28 to 24 are written to LIMPD. All other fields of REMAP remain unchanged.
- To write to LIM, write 0x*NNNN*5A5A (where *NNNN* is data) to REMAP. In this case, bits 20 to 16 are written to LIM.
- Any other write to REMAP is ignored.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	Reserved				LIMDP				Reserved				LIM			
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															REMAPLK
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-29 —	Always reads as 0. When writing, see the register description above.
28-24 LIMDP	LIM_DP data Contains the LIM_DP[19:15] data for remapping.

Table continues on the next page...

Table continued from the previous page...

Field	Function
23-21 —	Always reads as 0. When writing, see the register description above.
20-16 LIM	LIM data Contains the LIM[19:15] data for remapping.
15-1 —	Always reads as 0. When writing, see the register description above.
0 REMAPLK	Remap Lock Enable To set the remap lock, write 0xXXXXC3C3 (X = don't care) to REMAP. All other fields of REMAP remain unchanged. Set by software (sticky); cleared only by a reset. 0b - Lock disabled: can write to REMAP 1b - Lock enabled: cannot write to REMAP

8.5.1.6 Bitmap of Valid Control for Memory Context 0 (VMAPCTX0_WD0)

Offset

Register	Offset
VMAPCTX0_WD0	40h

Function

Each VMAPCTX*n*_WD*m* register contains a 32-bit bitmap that controls whether the corresponding 32-KByte block of flash (out of a maximum size of 2 MBytes) is encrypted/decrypted within Memory Context *n* using KEY*n*.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	VAL31	VAL30	VAL29	VAL28	VAL27	VAL26	VAL25	VAL24	VAL23	VAL22	VAL21	VAL20	VAL19	VAL18	VAL17	VAL16
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	VAL15	VAL14	VAL13	VAL12	VAL11	VAL10	VAL9	VAL8	VAL7	VAL6	VAL5	VAL4	VAL3	VAL2	VAL1	VAL0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 VALi	Block valid enable for encryption/decryption Set bit *i* to enable encryption/decryption for the corresponding 32-KByte block *i*. Set by software; cleared by software; cleared by POR (Power-On Reset). 0b - Disable 1b - Enable

8.5.1.7 Bitmap of Valid Control for Memory Context 0 (VMAPCTX0_WD1)

Offset

Register	Offset
VMAPCTX0_WD1	44h

Function

Each VMAPCTX*n*_WD*m* register contains a 32-bit bitmap that controls whether the corresponding 32-KByte block of flash (out of a maximum size of 2 MBytes) is encrypted/decrypted within Memory Context *n* using KEY*n*.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	VAL63	VAL62	VAL61	VAL60	VAL59	VAL58	VAL57	VAL56	VAL55	VAL54	VAL53	VAL52	VAL51	VAL50	VAL49	VAL48
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	VAL47	VAL46	VAL45	VAL44	VAL43	VAL42	VAL41	VAL40	VAL39	VAL38	VAL37	VAL36	VAL35	VAL34	VAL33	VAL32
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 VALi	Block valid enable for encryption/decryption Set bit *i* to enable encryption/decryption for the corresponding 32-KByte block *i*. Set by software; cleared by software; cleared by POR (Power-On Reset). 0b - Disable 1b - Enable

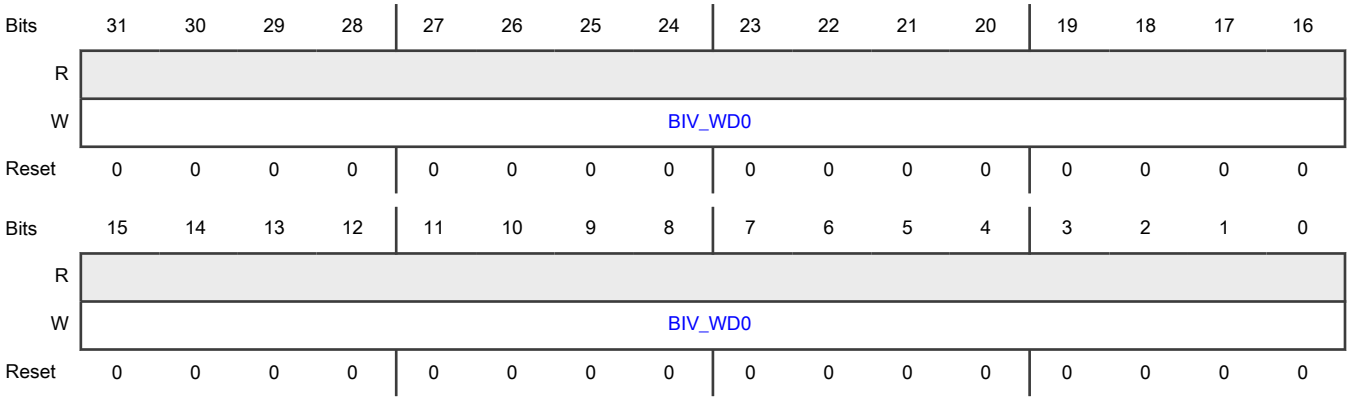
8.5.1.8 Block Initial Vector for Memory Context 0 (BIVCTX0_WD0)

Offset

Register	Offset
BIVCTX0_WD0	48h

Function
For a given CTX*n*, the two combined BIVCTX*n*_WD*m* registers contain the 64-bit initial vector used as the nonce to encrypt/decrypt this block.

Diagram



Fields

Field	Function
31-0 BIV_WD0	Block Initial Vector Word0 Contains bits 31 to 0 of the block initial vector. Set by software; cleared by software; cleared by POR (Power-On Reset).

8.5.1.9 Block Initial Vector for Memory Context 0 (BIVCTX0_WD1)

Offset

Register	Offset
BIVCTX0_WD1	4Ch

Function
For a given CTX*n*, the two combined BIVCTX*n*_WD*m* registers contain the 64-bit initial vector used as the nonce to encrypt/decrypt this block.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	BIV_WD1															
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	BIV_WD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 BIV_WD1	Block Initial Vector Word1 Contains bits 63 to 32 of the block initial vector. Set by software; cleared by software; cleared by POR (Power-On Reset).

8.5.1.10 Bitmap of Valid Control for Memory Context 1 (VMAPCTX1_WD0)

Offset

Register	Offset
VMAPCTX1_WD0	50h

Function

Each VMAPCTX*n*_WD*m* register contains a 32-bit bitmap that controls whether the corresponding 32-KByte block of flash (out of a maximum size of 2 MBytes) is encrypted/decrypted within Memory Context *n* using KEY*n*.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	VAL31	VAL30	VAL29	VAL28	VAL27	VAL26	VAL25	VAL24	VAL23	VAL22	VAL21	VAL20	VAL19	VAL18	VAL17	VAL16
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	VAL15	VAL14	VAL13	VAL12	VAL11	VAL10	VAL9	VAL8	VAL7	VAL6	VAL5	VAL4	VAL3	VAL2	VAL1	VAL0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 VALi	Block valid enable for encryption/decryption Set bit *i* to enable encryption/decryption for the corresponding 32-KByte block *i*. Set by software; cleared by software; cleared by POR (Power-On Reset). 0b - Disable 1b - Enable

8.5.1.11 Bitmap of Valid Control for Memory Context 1 (VMAPCTX1_WD1)

Offset

Register	Offset
VMAPCTX1_WD1	54h

Function

Each VMAPCTX*n*_WD*m* register contains a 32-bit bitmap that controls whether the corresponding 32-KByte block of flash (out of a maximum size of 2 MBytes) is encrypted/decrypted within Memory Context *n* using KEY*n*.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	VAL63	VAL62	VAL61	VAL60	VAL59	VAL58	VAL57	VAL56	VAL55	VAL54	VAL53	VAL52	VAL51	VAL50	VAL49	VAL48
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	VAL47	VAL46	VAL45	VAL44	VAL43	VAL42	VAL41	VAL40	VAL39	VAL38	VAL37	VAL36	VAL35	VAL34	VAL33	VAL32
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 VALi	Block valid enable for encryption/decryption Set bit *i* to enable encryption/decryption for the corresponding 32-KByte block *i*. Set by software; cleared by software; cleared by POR (Power-On Reset). 0b - Disable 1b - Enable

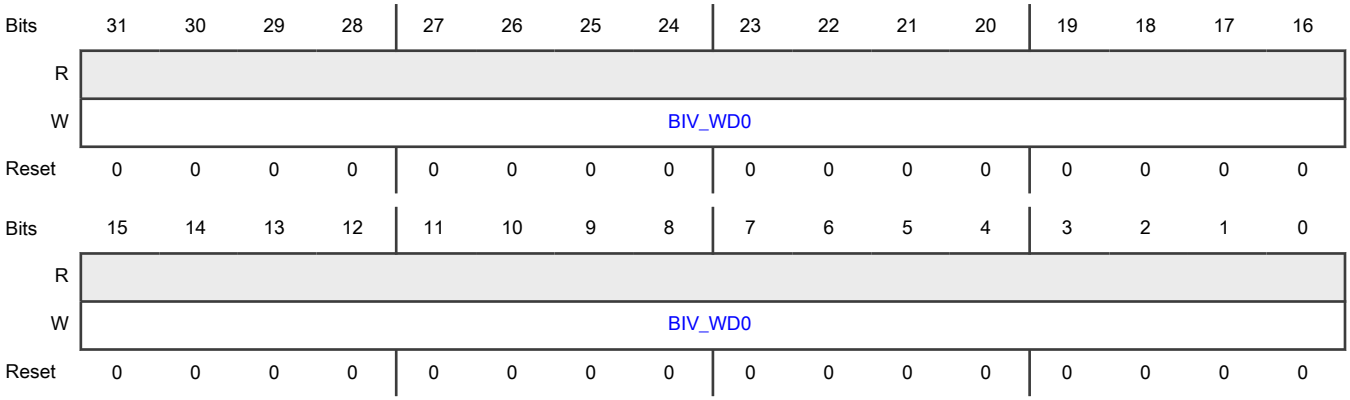
8.5.1.12 Block Initial Vector for Memory Context 1 (BIVCTX1_WD0)

Offset

Register	Offset
BIVCTX1_WD0	58h

Function
For a given CTX*n*, the two combined BIVCTX*n*_WD*m* registers contain the 64-bit initial vector used as the nonce to encrypt/decrypt this block.

Diagram



Fields

Field	Function
31-0 BIV_WD0	Block Initial Vector Word0 Contains bits 31 to 0 of the block initial vector. Set by software; cleared by software; cleared by POR (Power-On Reset).

8.5.1.13 Block Initial Vector for Memory Context 1 (BIVCTX1_WD1)

Offset

Register	Offset
BIVCTX1_WD1	5Ch

Function
For a given CTX*n*, the two combined BIVCTX*n*_WD*m* registers contain the 64-bit initial vector used as the nonce to encrypt/decrypt this block.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	BIV_WD1															
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	BIV_WD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 BIV_WD1	Block Initial Vector Word1 Contains bits 63 to 32 of the block initial vector. Set by software; cleared by software; cleared by POR (Power-On Reset).

8.5.1.14 Bitmap of Valid Control for Memory Context 2 (VMAPCTX2_WD0)

Offset

Register	Offset
VMAPCTX2_WD0	60h

Function

Each VMAPCTX*n*_WD*m* register contains a 32-bit bitmap that controls whether the corresponding 32-KByte block of flash (out of a maximum size of 2 MBytes) is encrypted/decrypted within Memory Context *n* using KEY*n*.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	VAL31	VAL30	VAL29	VAL28	VAL27	VAL26	VAL25	VAL24	VAL23	VAL22	VAL21	VAL20	VAL19	VAL18	VAL17	VAL16
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	VAL15	VAL14	VAL13	VAL12	VAL11	VAL10	VAL9	VAL8	VAL7	VAL6	VAL5	VAL4	VAL3	VAL2	VAL1	VAL0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 VALi	Block valid enable for encryption/decryption Set bit *i* to enable encryption/decryption for the corresponding 32-KByte block *i*. Set by software; cleared by software; cleared by POR (Power-On Reset). 0b - Disable 1b - Enable

8.5.1.15 Bitmap of Valid Control for Memory Context 2 (VMAPCTX2_WD1)

Offset

Register	Offset
VMAPCTX2_WD1	64h

Function

Each VMAPCTX*n*_WD*m* register contains a 32-bit bitmap that controls whether the corresponding 32-KByte block of flash (out of a maximum size of 2 MBytes) is encrypted/decrypted within Memory Context *n* using KEY*n*.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	VAL63	VAL62	VAL61	VAL60	VAL59	VAL58	VAL57	VAL56	VAL55	VAL54	VAL53	VAL52	VAL51	VAL50	VAL49	VAL48
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	VAL47	VAL46	VAL45	VAL44	VAL43	VAL42	VAL41	VAL40	VAL39	VAL38	VAL37	VAL36	VAL35	VAL34	VAL33	VAL32
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 VALi	Block valid enable for encryption/decryption Set bit *i* to enable encryption/decryption for the corresponding 32-KByte block *i*. Set by software; cleared by software; cleared by POR (Power-On Reset). 0b - Disable 1b - Enable

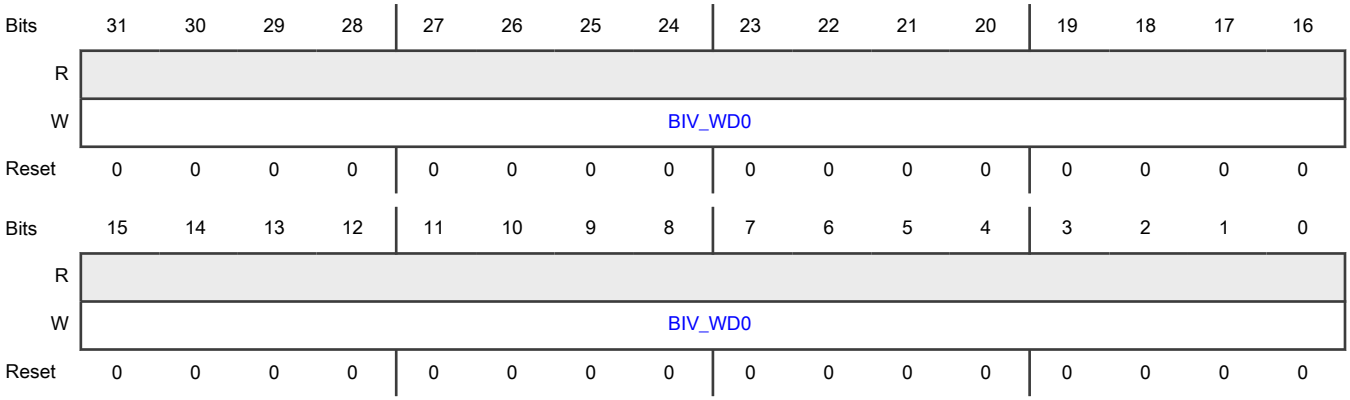
8.5.1.16 Block Initial Vector for Memory Context 2 (BIVCTX2_WD0)

Offset

Register	Offset
BIVCTX2_WD0	68h

Function
For a given CTX*n*, the two combined BIVCTX*n*_WD*m* registers contain the 64-bit initial vector used as the nonce to encrypt/decrypt this block.

Diagram



Fields

Field	Function
31-0 BIV_WD0	Block Initial Vector Word0 Contains bits 31 to 0 of the block initial vector. Set by software; cleared by software; cleared by POR (Power-On Reset).

8.5.1.17 Block Initial Vector for Memory Context 2 (BIVCTX2_WD1)

Offset

Register	Offset
BIVCTX2_WD1	6Ch

Function
For a given CTX*n*, the two combined BIVCTX*n*_WD*m* registers contain the 64-bit initial vector used as the nonce to encrypt/decrypt this block.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R																
W	BIV_WD1															
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	BIV_WD1															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 BIV_WD1	Block Initial Vector Word1 Contains bits 63 to 32 of the block initial vector. Set by software; cleared by software; cleared by POR (Power-On Reset).

8.5.1.18 Bitmap of Valid Control for Memory Context 3 (VMAPCTX3_WD0)

Offset

Register	Offset
VMAPCTX3_WD0	70h

Function

Each VMAPCTX*n*_WD*m* register contains a 32-bit bitmap that controls whether the corresponding 32-KByte block of flash (out of a maximum size of 2 MBytes) is encrypted/decrypted within Memory Context *n* using KEY*n*.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	VAL31	VAL30	VAL29	VAL28	VAL27	VAL26	VAL25	VAL24	VAL23	VAL22	VAL21	VAL20	VAL19	VAL18	VAL17	VAL16
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	VAL15	VAL14	VAL13	VAL12	VAL11	VAL10	VAL9	VAL8	VAL7	VAL6	VAL5	VAL4	VAL3	VAL2	VAL1	VAL0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 VALi	Block valid enable for encryption/decryption Set bit *i* to enable encryption/decryption for the corresponding 32-KByte block *i*. Set by software; cleared by software; cleared by POR (Power-On Reset). 0b - Disable 1b - Enable

8.5.1.19 Bitmap of Valid Control for Memory Context 3 (VMAPCTX3_WD1)

Offset

Register	Offset
VMAPCTX3_WD1	74h

Function

Each VMAPCTX*n*_WD*m* register contains a 32-bit bitmap that controls whether the corresponding 32-KByte block of flash (out of a maximum size of 2 MBytes) is encrypted/decrypted within Memory Context *n* using KEY*n*.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	VAL63	VAL62	VAL61	VAL60	VAL59	VAL58	VAL57	VAL56	VAL55	VAL54	VAL53	VAL52	VAL51	VAL50	VAL49	VAL48
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	VAL47	VAL46	VAL45	VAL44	VAL43	VAL42	VAL41	VAL40	VAL39	VAL38	VAL37	VAL36	VAL35	VAL34	VAL33	VAL32
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 VALi	Block valid enable for encryption/decryption Set bit *i* to enable encryption/decryption for the corresponding 32-KByte block *i*. Set by software; cleared by software; cleared by POR (Power-On Reset). 0b - Disable 1b - Enable

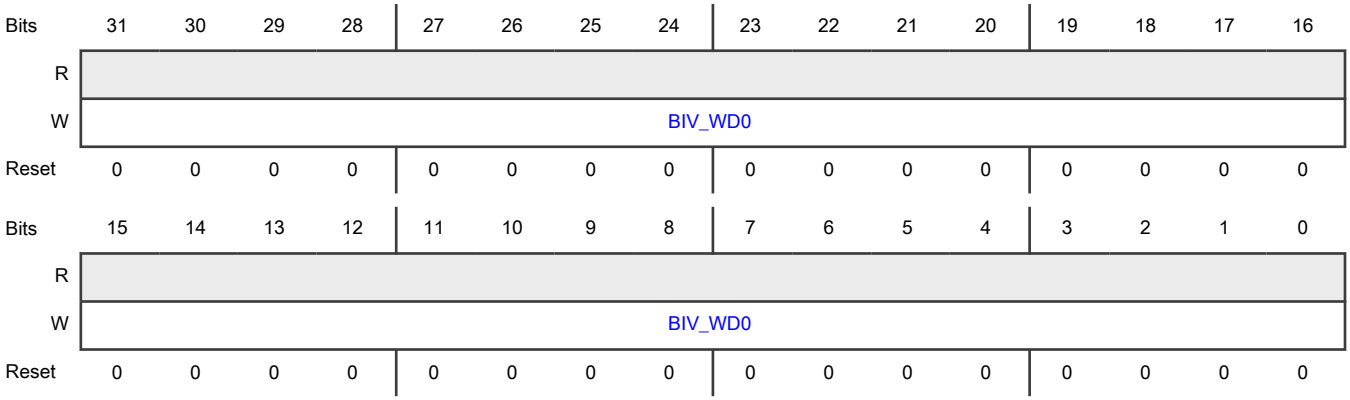
8.5.1.20 Block Initial Vector for Memory Context 3 (BIVCTX3_WD0)

Offset

Register	Offset
BIVCTX3_WD0	78h

Function
For a given CTX*n*, the two combined BIVCTX*n*_WD*m* registers contain the 64-bit initial vector used as the nonce to encrypt/decrypt this block.

Diagram



Fields

Field	Function
31-0 BIV_WD0	Block Initial Vector Word0 Contains bits 31 to 0 of the block initial vector. Set by software; cleared by software; cleared by POR (Power-On Reset).

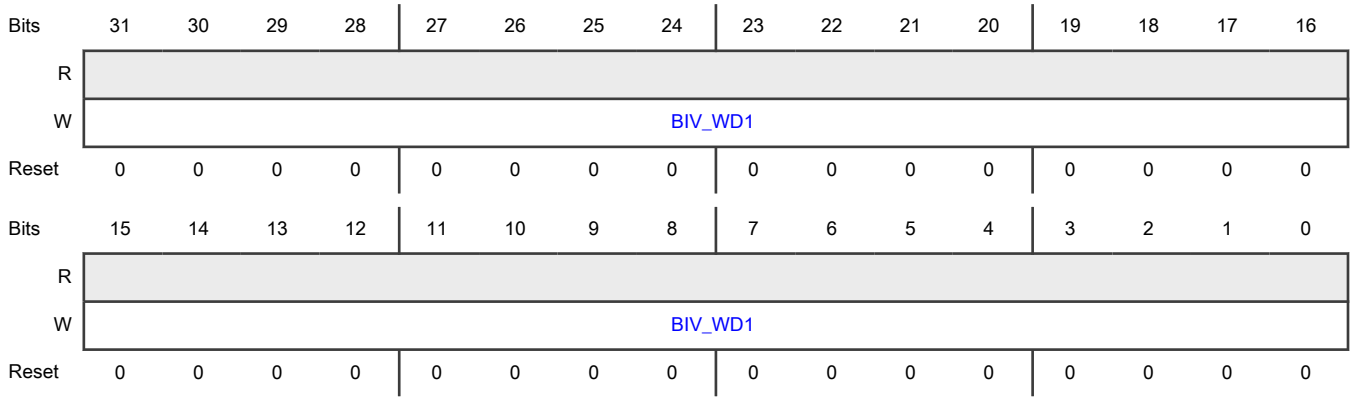
8.5.1.21 Block Initial Vector for Memory Context 3 (BIVCTX3_WD1)

Offset

Register	Offset
BIVCTX3_WD1	7Ch

Function
For a given CTX*n*, the two combined BIVCTX*n*_WD*m* registers contain the 64-bit initial vector used as the nonce to encrypt/decrypt this block.

Diagram



Fields

Field	Function
31-0 BIV_WD1	Block Initial Vector Word1 Contains bits 63 to 32 of the block initial vector. Set by software; cleared by software; cleared by POR (Power-On Reset).

8.5.1.22 Memory Region 0, Masked Key Word 0 (MR0MASKEDKEYWORD0)

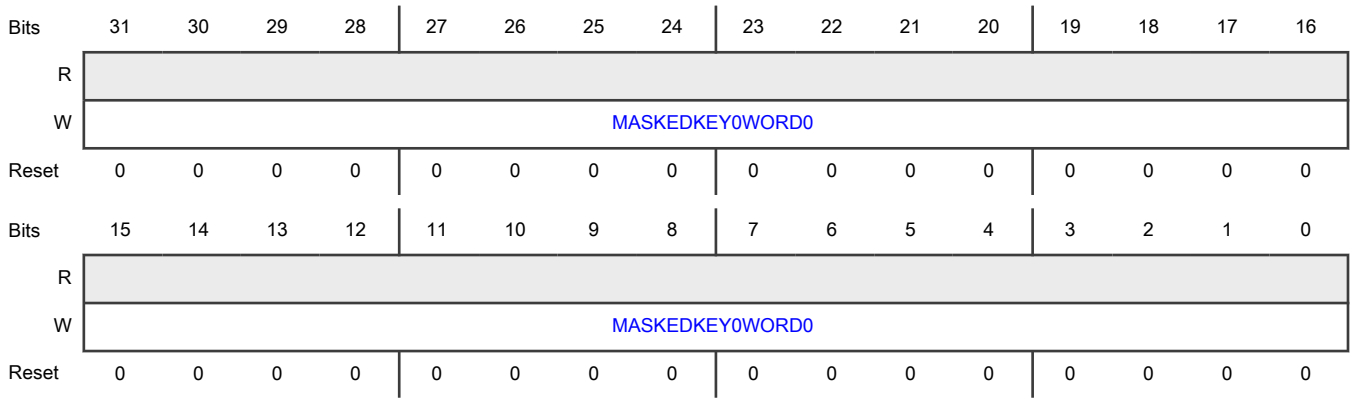
Offset

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



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)

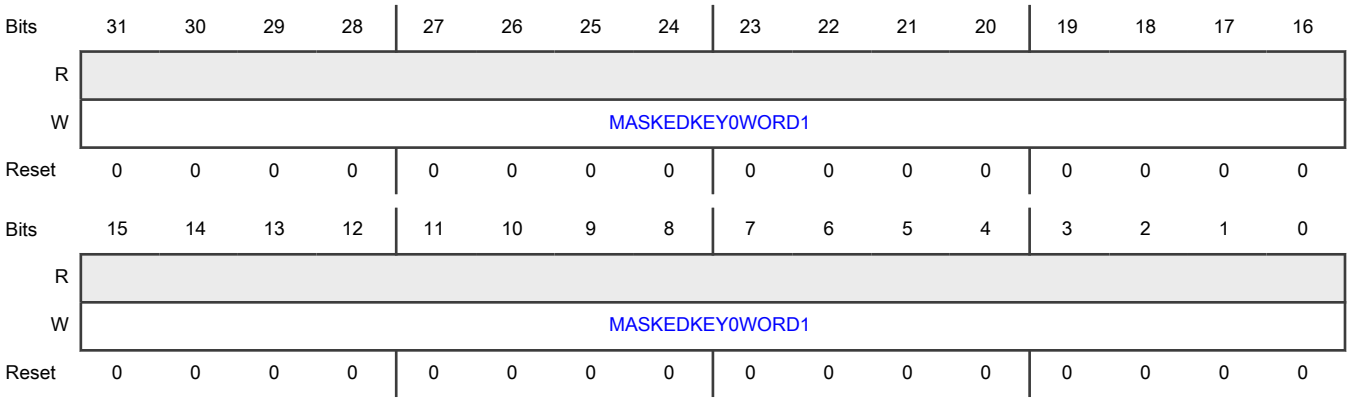
8.5.1.23 Memory Region 0, Masked Key Word 1 (MR0MASKEDKEYWORD1)

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



Fields

Field	Function
31-0 MASKEDKEY0WORD1	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)

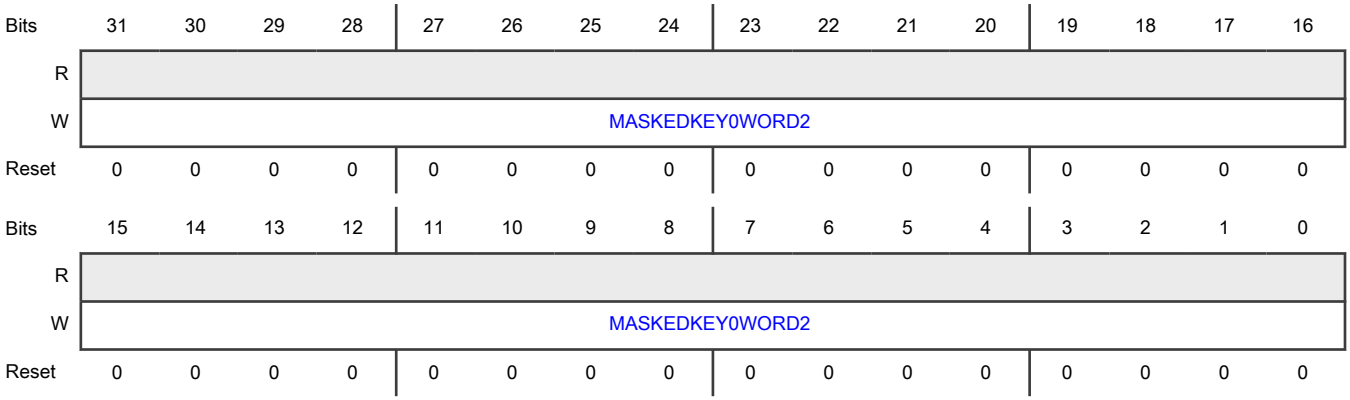
8.5.1.24 Memory Region 0, Masked Key Word 2 (MR0MASKEDKEYWORD2)

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



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)

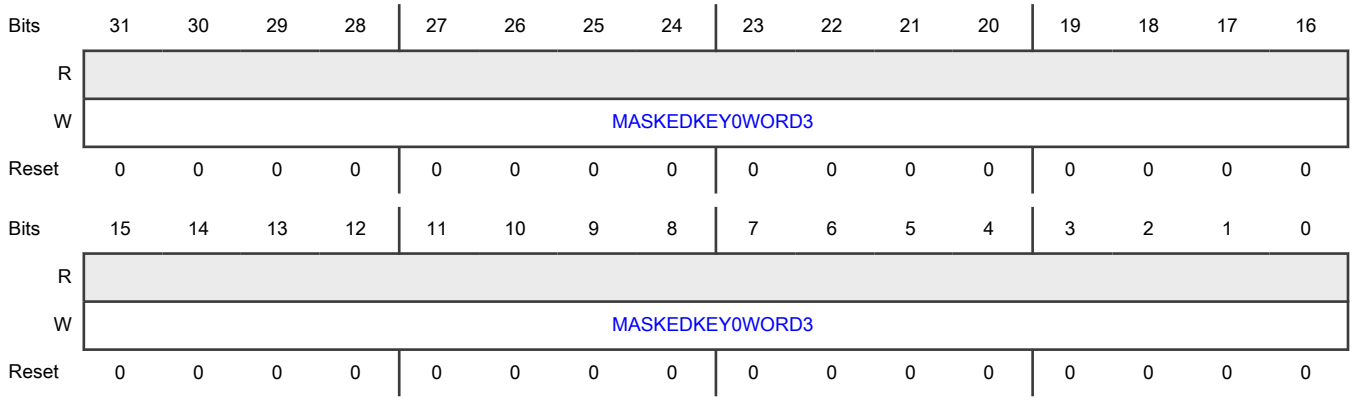
8.5.1.25 Memory Region 0, Masked Key Word 3 (MR0MASKEDKEYWORD3)

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



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)

8.5.1.26 Memory Region 0, Mask for Key Word 0 (MR0MASKFORKEYWORD0)

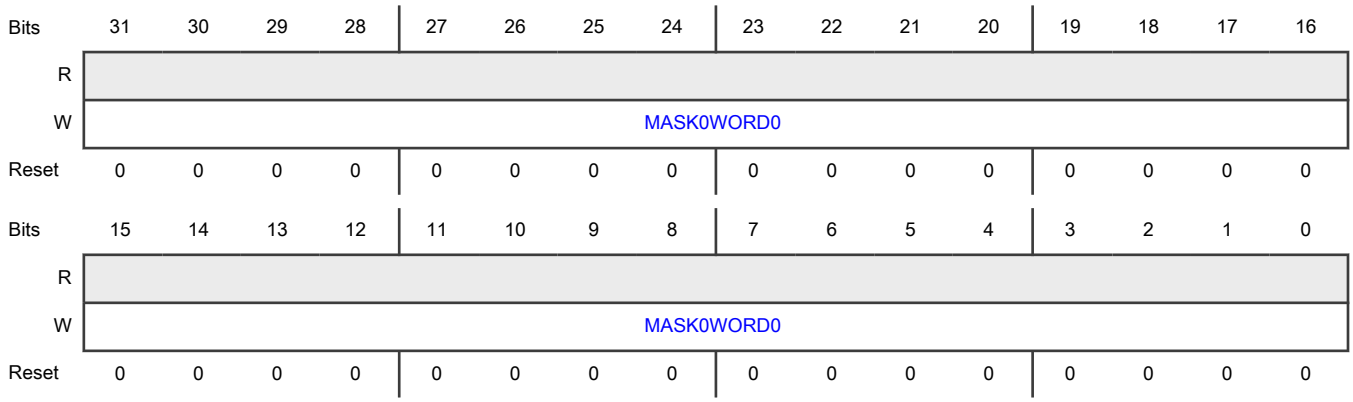
Offset

Register	Offset
MR0MASKFORKEYWORD0	110h

Function

Contains bits [31:0] of the 128-bit mask used to generate the PRINCE key for this region.

Diagram



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**)

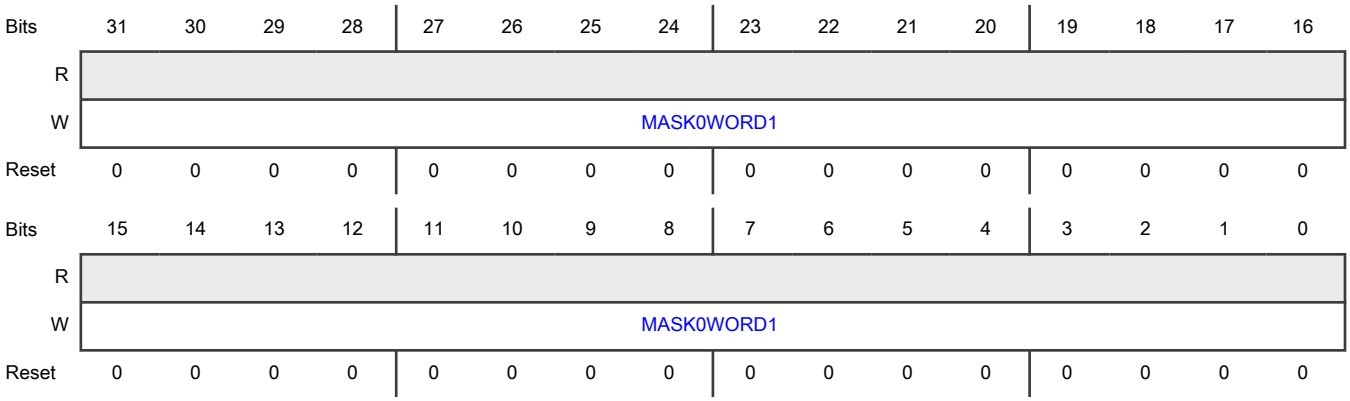
8.5.1.27 Memory Region 0, Mask for Key Word 1 (MR0MASKFORKEYWORD1)

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



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**)

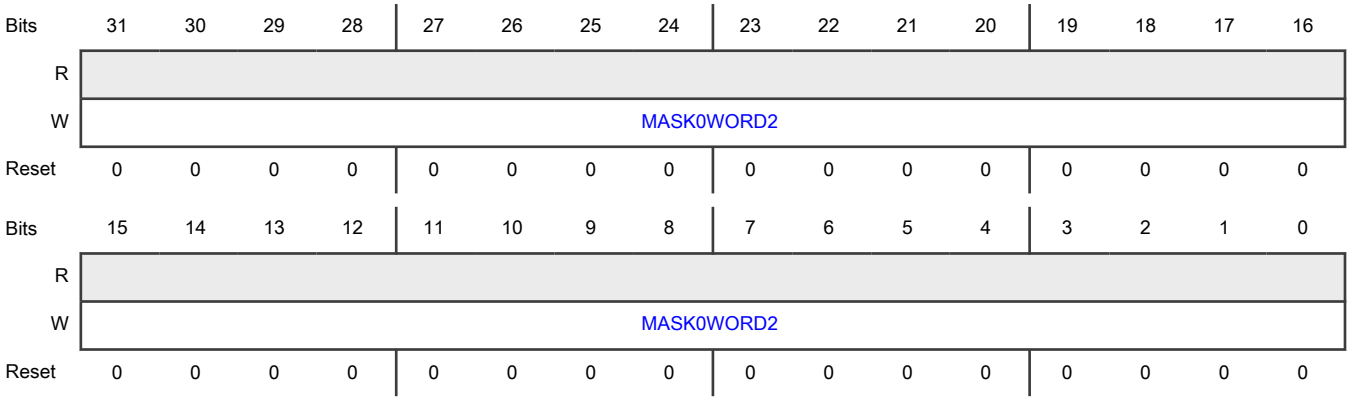
8.5.1.28 Memory Region 0, Mask for Key Word 2 (MR0MASKFORKEYWORD2)

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



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**)

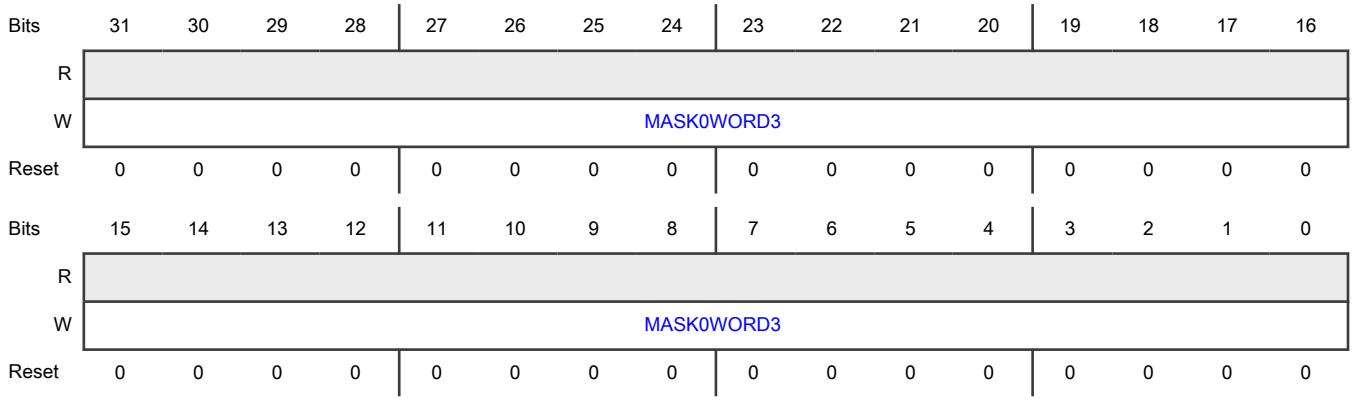
8.5.1.29 Memory Region 0, Mask for Key Word 3 (MR0MASKFORKEYWORD3)

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



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**)

8.5.1.30 Memory Region 0 Start Address (MR0STARTADDR)

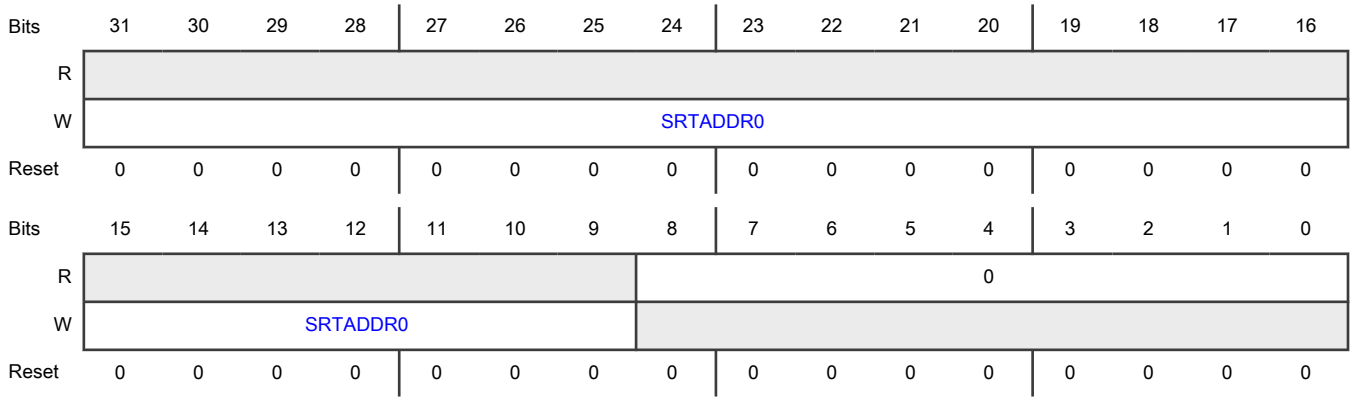
Offset

Register	Offset
MR0STARTADDR	120h

Function

Contains the start address for this memory region.

Diagram



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.

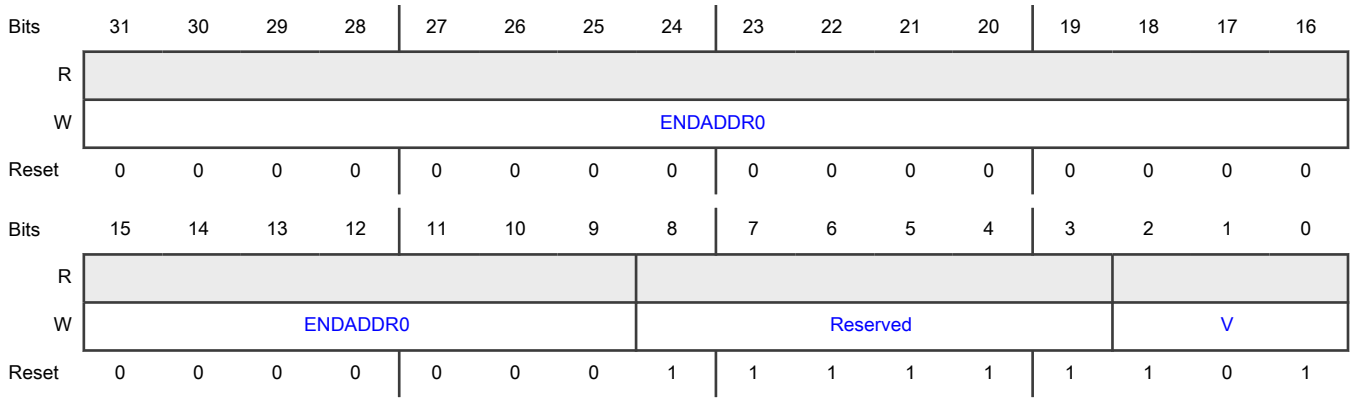
8.5.1.31 Memory Region 0 End Address (MR0ENDADDR)

Offset

Register	Offset
MR0ENDADDR	124h

Function
Contains the end address and the valid indicator for this memory region.

Diagram



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

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 0 is valid. Subsequent reads return 111b. 101b - Memory Region 0 is not valid.

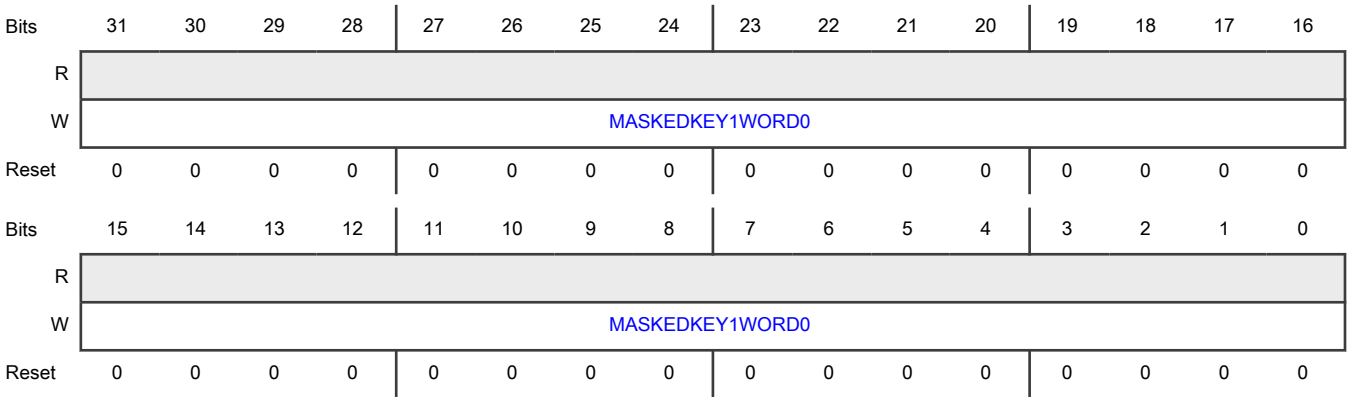
8.5.1.32 Memory Region 1, Masked Key Word 0 (MR1MASKEDKEYWORD0)

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



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)

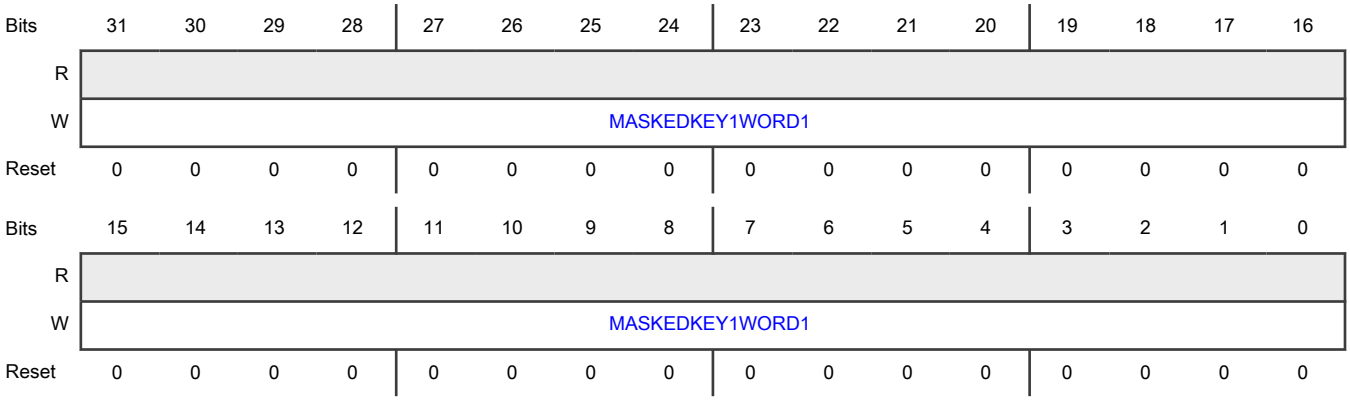
8.5.1.33 Memory Region 1, Masked Key Word 1 (MR1MASKEDKEYWORD1)

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



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)

8.5.1.34 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)

8.5.1.35 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)

8.5.1.36 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**)

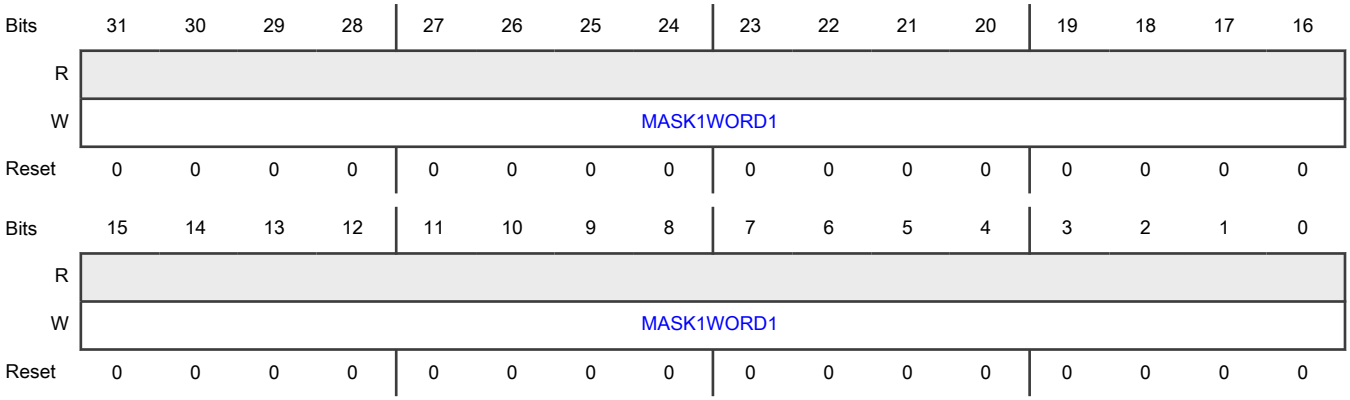
8.5.1.37 Memory Region 1, Mask for Key Word 1 (MR1MASKFORKEYWORD1)

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



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**)

8.5.1.38 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**)

8.5.1.39 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**)

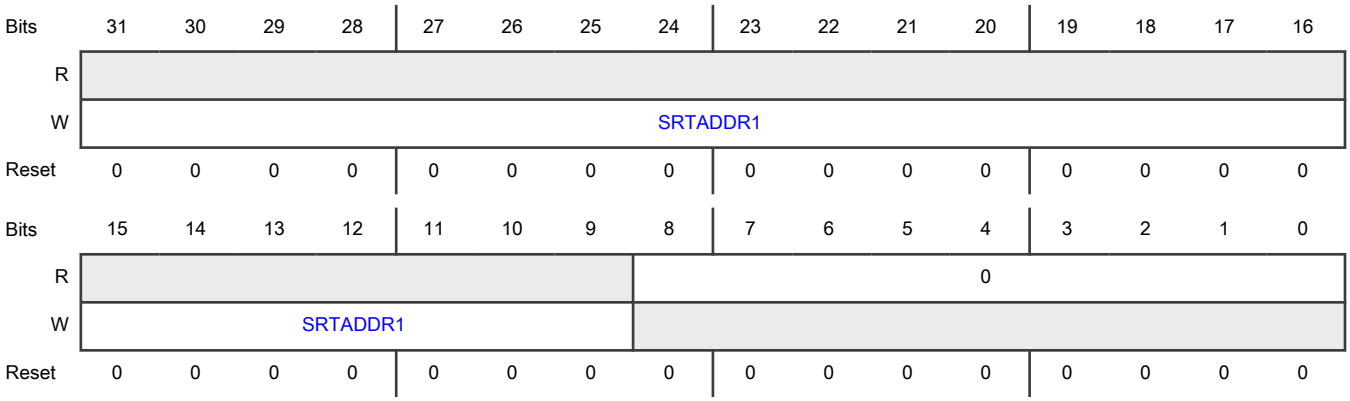
8.5.1.40 Memory Region 1 Start Address (MR1STARTADDR)

Offset

Register	Offset
MR1STARTADDR	160h

Function
Contains the start address for this memory region.

Diagram



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.

8.5.1.41 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.

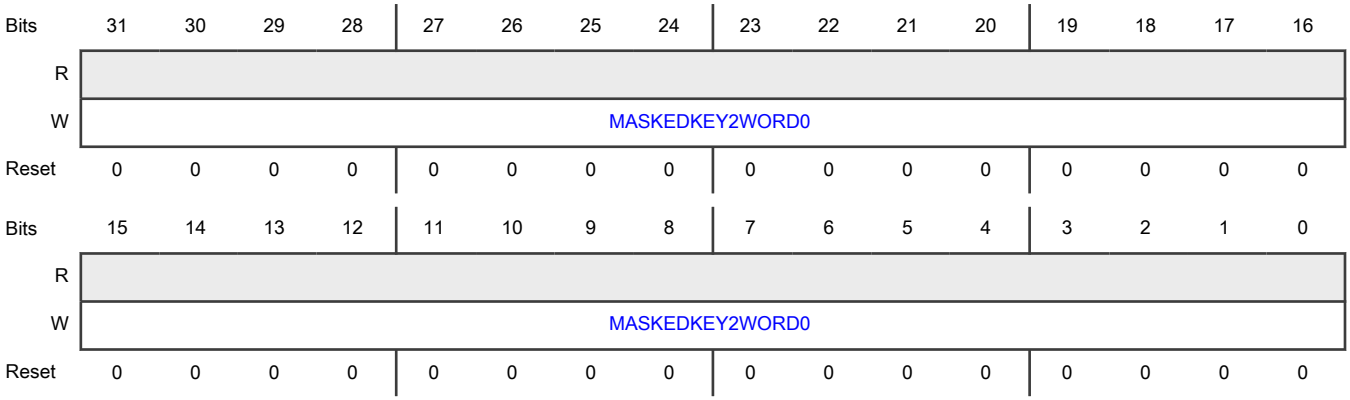
8.5.1.42 Memory Region 2, Masked Key Word 0 (MR2MASKEDKEYWORD0)

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



Fields

Field	Function
31-0 MASKEDKEY2WORD0	Masked Key Word 2 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)

8.5.1.43 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 MASKEDKEY2WORD1	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)

8.5.1.44 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)

8.5.1.45 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)

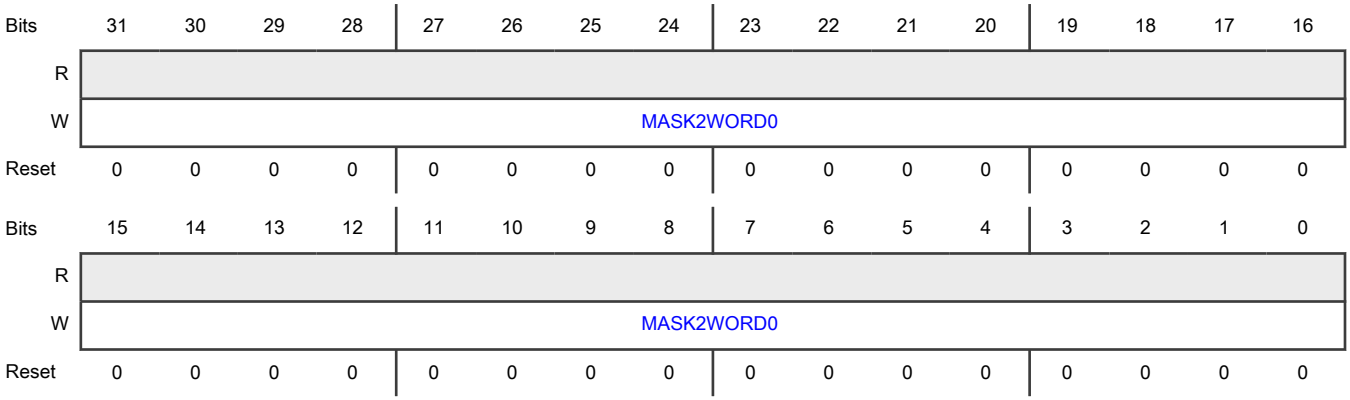
8.5.1.46 Memory Region 2, Mask for Key Word 0 (MR2MASKFORKEYWORD0)

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



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**)

8.5.1.47 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**)

8.5.1.48 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**)

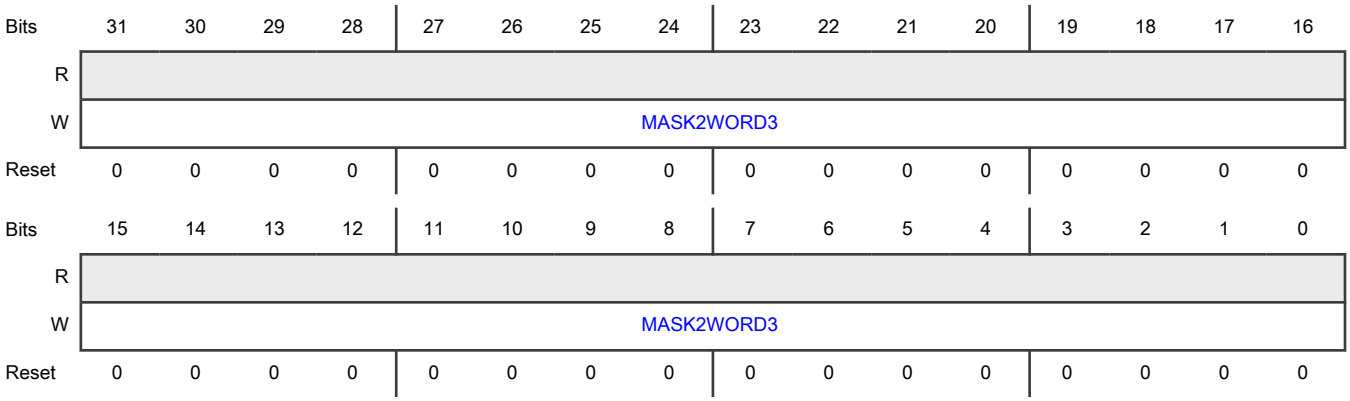
8.5.1.49 Memory Region 2, Mask for Key Word 3 (MR2MASKFORKEYWORD3)

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



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**)

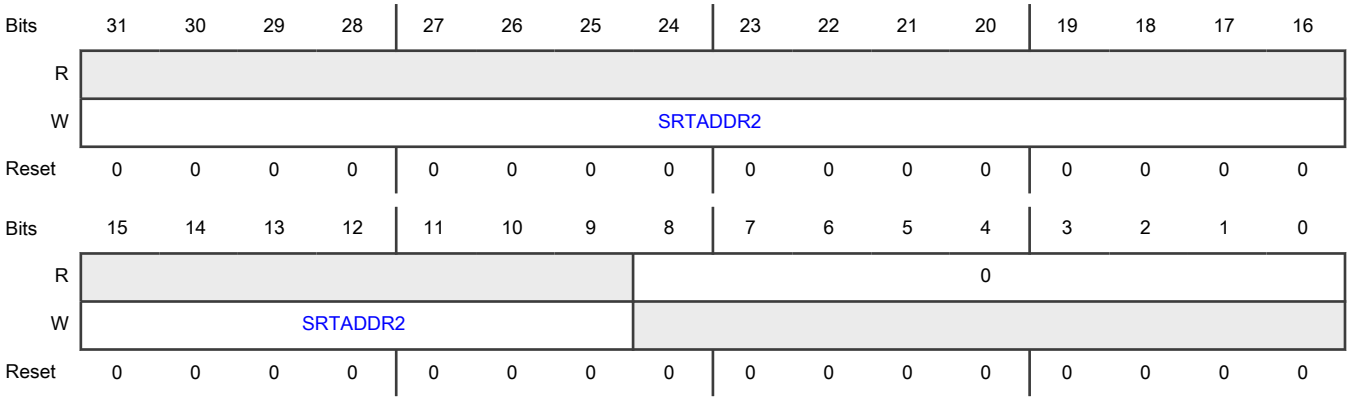
8.5.1.50 Memory Region 2 Start Address (MR2STARTADDR)

Offset

Register	Offset
MR2STARTADDR	1A0h

Function
Contains the start address for this memory region.

Diagram



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.

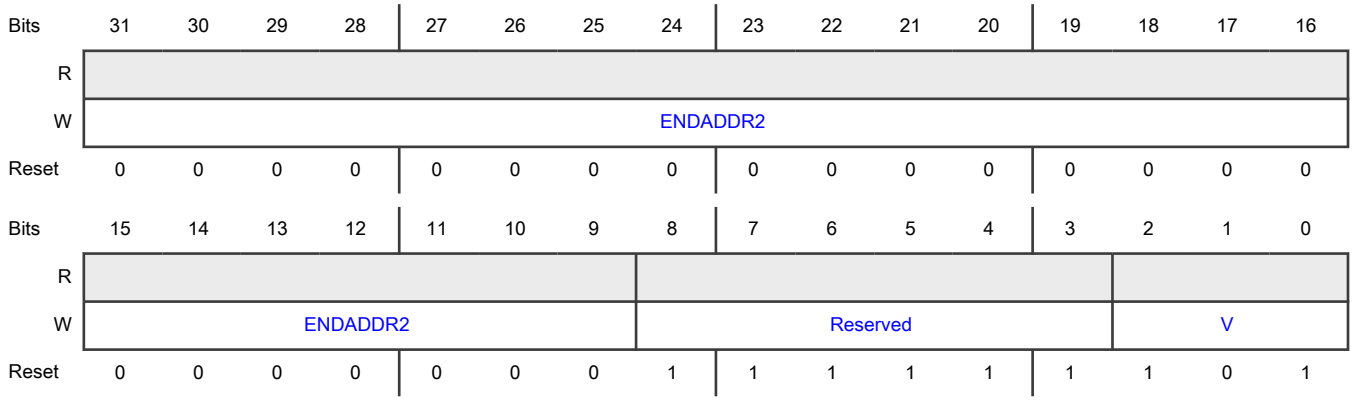
8.5.1.51 Memory Region 2 End Address (MR2ENDADDR)

Offset

Register	Offset
MR2ENDADDR	1A4h

Function
Contains the end address and the valid indicator for this memory region.

Diagram



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 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.

8.5.1.52 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)

8.5.1.53 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)

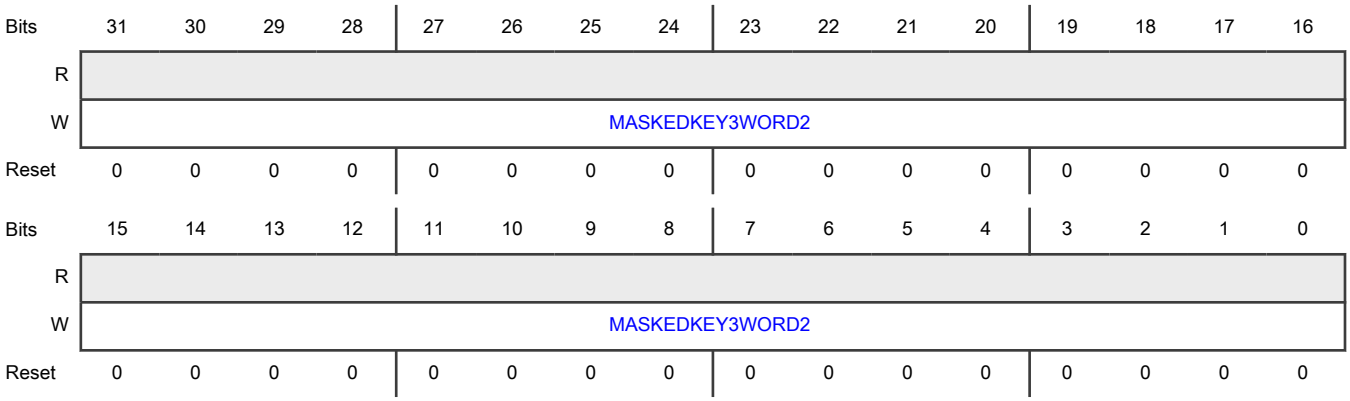
8.5.1.54 Memory Region 3, Masked Key Word 2 (MR3MASKEDKEYWORD2)

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



Fields

Field	Function
31-0 MASKEDKEY3WORD2	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)

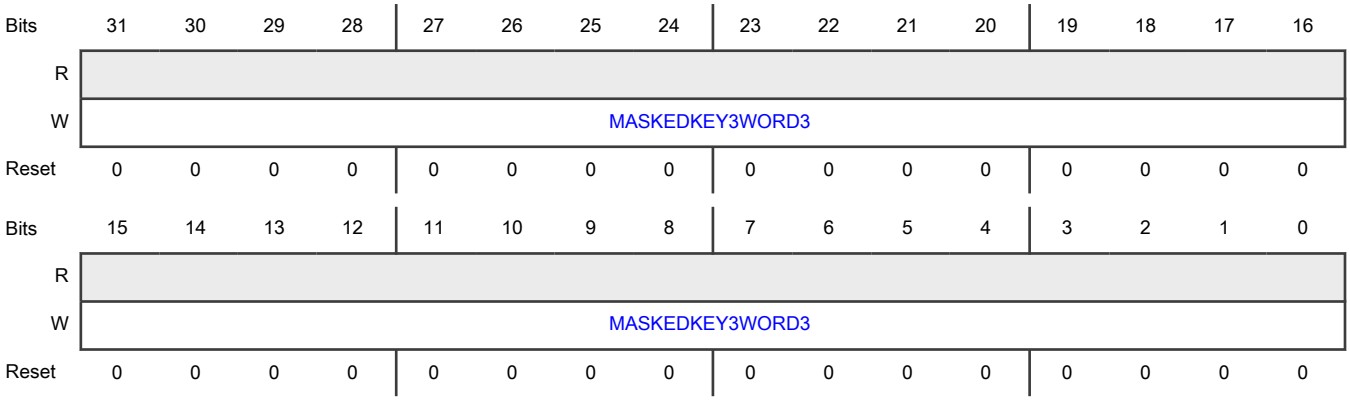
8.5.1.55 Memory Region 3, Masked Key Word 3 (MR3MASKEDKEYWORD3)

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



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)

8.5.1.56 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**)

8.5.1.57 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**)

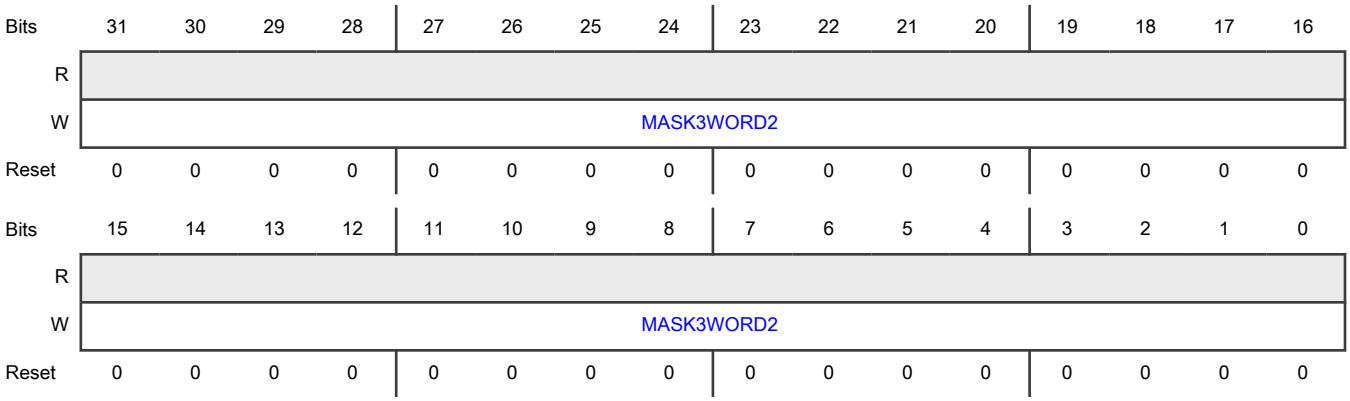
8.5.1.58 Memory Region 3, Mask for Key Word 2 (MR3MASKFORKEYWORD2)

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



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**)

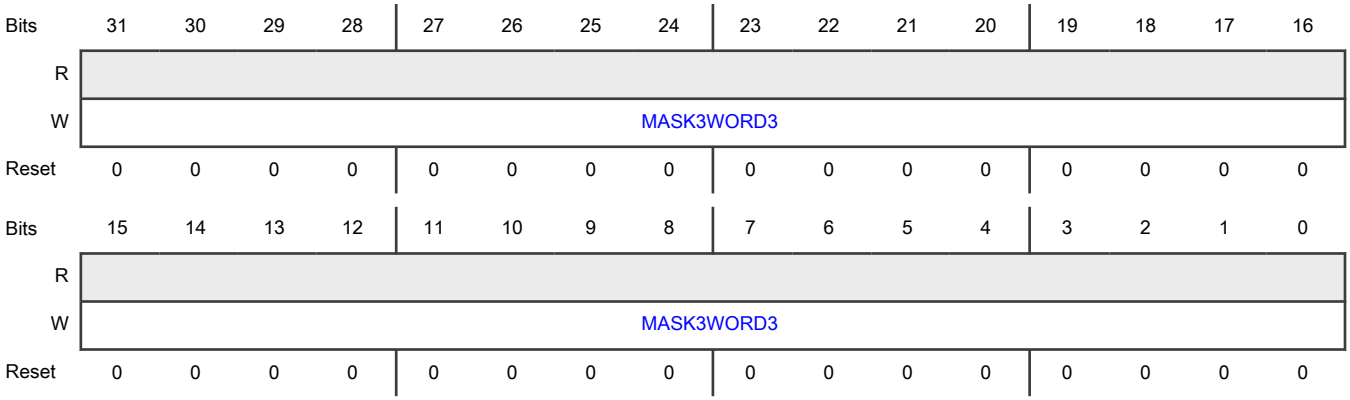
8.5.1.59 Memory Region 3, Mask for Key Word 3 (MR3MASKFORKEYWORD3)

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



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**)

8.5.1.60 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.

8.5.1.61 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 9
ROM Bootloader

9.1 Introduction

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.

9.2 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 0x1480 0000. The following diagram shows the memory map in-use during ROM execution. The first 16 KB of Tightly Coupled Memory - System (STCM) (0x30000000 – 0x30004000) needs to be reserved for ROM bootloader execution and some ROM API execution. Please note that 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.

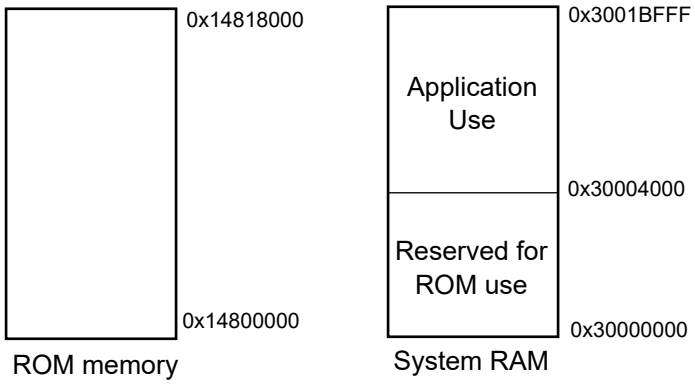


Figure 18. ROM bootloader memory usage

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.
- Low power wakeup path.

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.

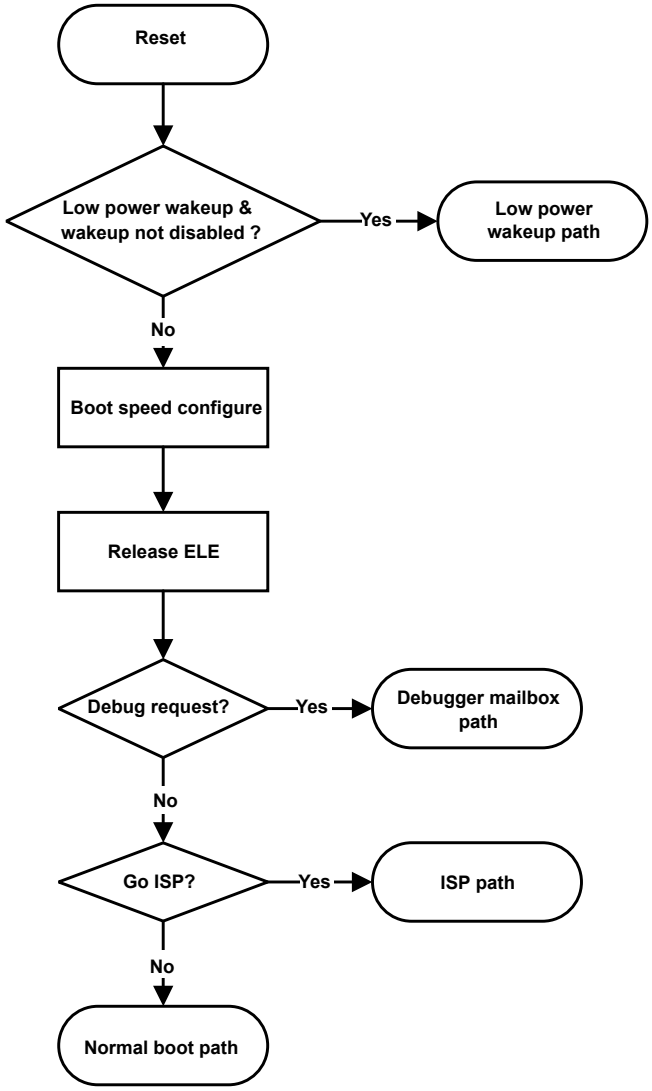


Figure 19. Top-level boot flow

9.2.1 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.

Table 75. Lifecycle states

LIFECYCLE	Lifecycle state [0:7]	CM33 image authentication required to boot?	Radio core (CM3) locked if radio authentication fails	Debug authentication required to debug CM33?	Debug authentication required to debug radio core (CM3)?	Able to access radio flash from CM33 after exiting ROM?
0000_0000	Blank	NA	NA	NA	NA	NA

Table continues on the next page...

Table 75. Lifecycle states (continued)

LIFECYCLE	Lifecycle state [0:7]	CM33 image authentication required to boot?	Radio core (CM3) locked if radio authentication fails	Debug authentication required to debug CM33?	Debug authentication required to debug radio core (CM3)?	Able to access radio flash from CM33 after exiting ROM?
0000_0111	OEM Open	No (Boot even if authentication fails)	Yes	No	Yes	No
0000_1111	OEM Secure World Closed	Yes	Yes	Yes[1]	Yes	No
0001_1111	OEM Closed	Yes	Yes	Yes	Yes	No
1001_1111	OEM Locked	Yes	Yes	NA	NA	No
0011_1111	OEM Return	NA	NA	No	Yes	No

1. For secure world only.

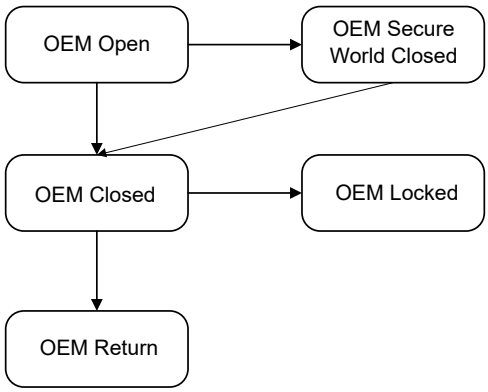


Figure 20. Lifecycle transitions

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. These fuse fields are shown in the table below.

Table 76. Additional fuse fields of interest

Name	Fuse index	Description
LIFECYCLE	0x0A	Lifecycle state. Corresponding product state can be determined from Table 75.
DBG_EN_LOCK	0x0B	Debug Enable Lock 0b - The debug access control registers are not write-locked when jumping to customer code. 1b - The debug access control registers are write-locked before jumping to customer code.
DBG_AUTH_DIS	0x0C	Debug Authentication Disabled

Table continues on the next page...

Table 76. Additional fuse fields of interest (continued)

Name	Fuse index	Description
		0b - Debug Authentication enabled. 1b - Debug Authentication disabled.
TZM_EN	0x0D	Trust Zone Mode Enable 0b - TZ-M is disabled by default. 1b - TZ-M is enabled.
SERIAL_DIS	0x11	Serial Download Disabled 0b - ISP path is enabled. 1b - ISP path is disabled.
WAKEUP_DIS	0x12	Wakeup Disabled 0b - Boot-ROM LP wakeup is enabled. 1b - Boot-ROM LP wakeup is disabled.
CUST_PROD_OEMFW_AUTH_PUK_REVOKE[3:0]	0x13	Key revocation indicators of CUST_PROD_OEMFW_AUTH_PUK.
DBG_AUTH_VU[15:0]	0x15	These Debug Authentication Vendor Usage fuses are used by customers to restrict debug certificates.
IMG_KEY_REVOKE[15:0]	0x16	These fuses are used to revoke image signing keys separately from root keys.
CUST_PROD_OEMFW_AUTH_PUK	0x1f	256-bit RoTKTH (customer trusted root key) used for CM33 main flash image authentication.
CUST_PROD_OEMFW_ENC_SK	0x20	256-bit encryption key used to protect confidentiality of OEM firmware, required for firmware updates using sb3 (SB3KDK).
DCFG_CC_SOCU_L1[16:0]	0x22	Debug authentication configuration for OEM1 customer.
DCFG_CC_SOCU_L2[16:0]	0x23	Debug authentication configuration for OEM2 customer.
SOC_VER_CNT	0x24	512-bits of version counter for various software components in SoC. Each software component version counter has dedicated fuse index 0x25 to 0x29.
CM33_S_VER_CNT	0x25	64-bit monotonic counter indicating the version of the secure OEM CM33 firmware. The number of set bits equals the version number.
CM33_NS_VER_CNT	0x26	256-bit monotonic counter indicating the version of the non-secure OEM CM33 firmware. The number of set bits equals the version number.
RADIO_VER_CNT	0x27	128-bit monotonic counter indicating the version of the radio firmware. The number of set bits equals the version number.
SNT_VER_CNT	0x28	32-bit monotonic counter indicating the version of the ELE loadable firmware. The number of set bits equals the version number.
CM33_BOOTLOADER_VER_CNT	0x29	Reserved for future usage of additional NXP software deliverable.

Table continues on the next page...

Table 76. Additional fuse fields of interest (continued)

Name	Fuse index	Description
GD_EN	0x30	• 0b - Core VDD glitch detection does not generate a hardware reset. • 1b - Boot ROM configures/enables Core VDD Glitch detection based reset.
CM33_S_VER_CNT_VIRTUAL	0x2A	64-bit virtual counter indicating the version of the secure OEM CM33 firmware. The integer value of this field indicates the number of set bits in CM33_S_VER_CNT and that number equals the version number.
CM33_NS_VER_CNT_VIRTUAL	0x2B	256-bit virtual counter indicating the version of the non-secure OEM CM33 firmware. The integer value of this field indicates the number of set bits in CM33_NS_VER_CNT and that number equals the version number.
RADIO_VER_CNT_VIRTUAL	0x2C	128-bit virtual counter indicating the version of the radio firmware. The integer value of this field indicates the number of set bits in RADIO_VER_CNT and that number equals the version number.
SNT_VER_CNT_VIRTUAL	0x2D	32-bit virtual counter indicating the version of the ELE loadable firmware. The integer value of this field indicates the number of set bits in SNT_VER_CNT and that number equals the version number.
CM33_BOOTLOADER_VER_CNT_VIRTUAL	0x2E	Reserved for future usage of additional NXP software deliverable.
HVD_EN	0x2F	• 0b - Core VDD high-voltage event does not generate hardware reset. • 1b - Boot ROM configures/enables Core VDD High voltage detection based reset.

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

- nboot API (See #unique_235 for more details about fuse programming using nboot API),
- ISP fuse-program and fuse-read command,
- sbloader “programFuses” command in sb3.1 file

9.2.2 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 77. 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

Table continues on the next page...

Table 77. User IFR allocation (continued)

Sector	Start address	End address	Name	Description
2	0x02004000	0x2005FFF	CMAC table	Used to save hashes of multiple boot components.
3	0x02006000	0x2007FFF	OTACFG	Used for Over-the-Air update

9.2.2.1 ROM bootloader configuration

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

Table 78. ROM configuration fields

Offset	Size (bytes)	Configuration Field	Description
0x0000	1	Restore 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

Table continues on the next page...

Table 78. ROM configuration fields (continued)

Offset	Size (bytes)	Configuration Field	Description
	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 • 00 = Normal boot (32 MHz) • 10 = Fast Boot (96 MHz) • 01, 11 = Normal boot (default) bit 7:3 – Reserved
	15	Reserved	Reserved
0x0060	1	Secure Boot Failure Options	Bit 0 – Secure Boot failure mode • 0 = Infinite sleep • 1 = Serial download Bit 1 – Secure Boot failure alert pin • 0 = Secure Boot failure alert pin is enabled • 1 = Secure Boot failure alert pin is disabled Bit 7:2 – Reserved
	1	Secure Boot Failure Alert Pin Selection	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, should align with 32 KB. ROM Bootloader does search from 0x80 and the first not 0xFFFFFFFF 32-bit will be the boot address. So, it's better to program from 0xB0 to 0x80, users will have 4 chances to update boot address.
0x00C0	64	Reserved	Reserved
0x0100	1	Peripherals Enable	Bit 0 – LPUART1 peripheral for ISP

Table continues on the next page...

Table 78. ROM configuration fields (continued)

Offset	Size (bytes)	Configuration Field	Description
			• 0 = LPUART1 disabled • 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 = CAN disabled • 1 = CAN 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 Slave Address	If 0xFF, defaults to 0x10 for LPI2C slave address. If not 0xFF, use this value as the 7-bit LPI2C slave 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

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 79. 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	32/32/16 MHz	1.0 V
10: Fast Boot	FRO_192M	96/96/24 MHz	1.1 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.

9.2.2.2 Token

Token needs to be programmed to the SECURE_PHANTOM_CONFIG fuse to change the ownership of radio firmware. By default, radio firmware is owned by NXP.

If developer at “OEM Open” lifecycle state requires an unrestricted access to radio flash for their own firmware development and debugging purposes, token is mandatory. Token contents must be programmed to SECURE_PHANTOM_CONFIG fuse, 0xddccbbaa (OEM1) or 0xaabbccdd (OEM2). If programmed token is valid, OEM_Enablement_Token fuse contents are used for NBU firmware authentication. Following a reset, specific decisions about locking radio core, blocking radio access, and debug authentication for CM3 core are made by boot ROM as described in Lifecycle states.

9.2.2.3 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. Note that, Buffer size of hash table to be reprogrammed must be at max. 48*4 bytes (to save flash programming time) but phrase-by-phrase programming can also be done as boot ROM progresses through various stages.

```
enum _cmac_table_index
{
kCmacIndexPrim = 0x0U,
kCmacIndexOEM = 0x1U,
kCmacIndexNBU = 0x2U,
kCmacIndexReserved = 0x3U,
kCmacIndexLast = 0x4U,
};

typedef struct{
uint32_t update_needed;
uint32_t reserved0[3];
uint8_t imgHash[NBOOT_CMAC_SIZE_IN_BYTES];
}cmac_info_t;

typedef struct {
cmac_info_t components[kCmacIndexLast]; /*!< Upto 4 components: 2 images (dual), 1 radio, 1 extended
bootloader (TBD) */
}cmac_table_t;
```

Table 80.

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

Table continues on the next page...

Table 80. (continued)

Offset	Size (bytes)	Field description	Description
0x0050	16	16 byte CMAC for radio image	Hash of radio (NBU) flash image data
0x0060	4	Reserved	Reserved
	12	Reserved	Reserved
0x0070	16	Reserved	Reserved

9.2.2.4 Over-the-air (OTA) update configuration

Table 81. 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};

9.2.3 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. Boot ROM prepares the system for execution of application code residing in internal flash.

Please 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 21 shows the flow of normal boot.

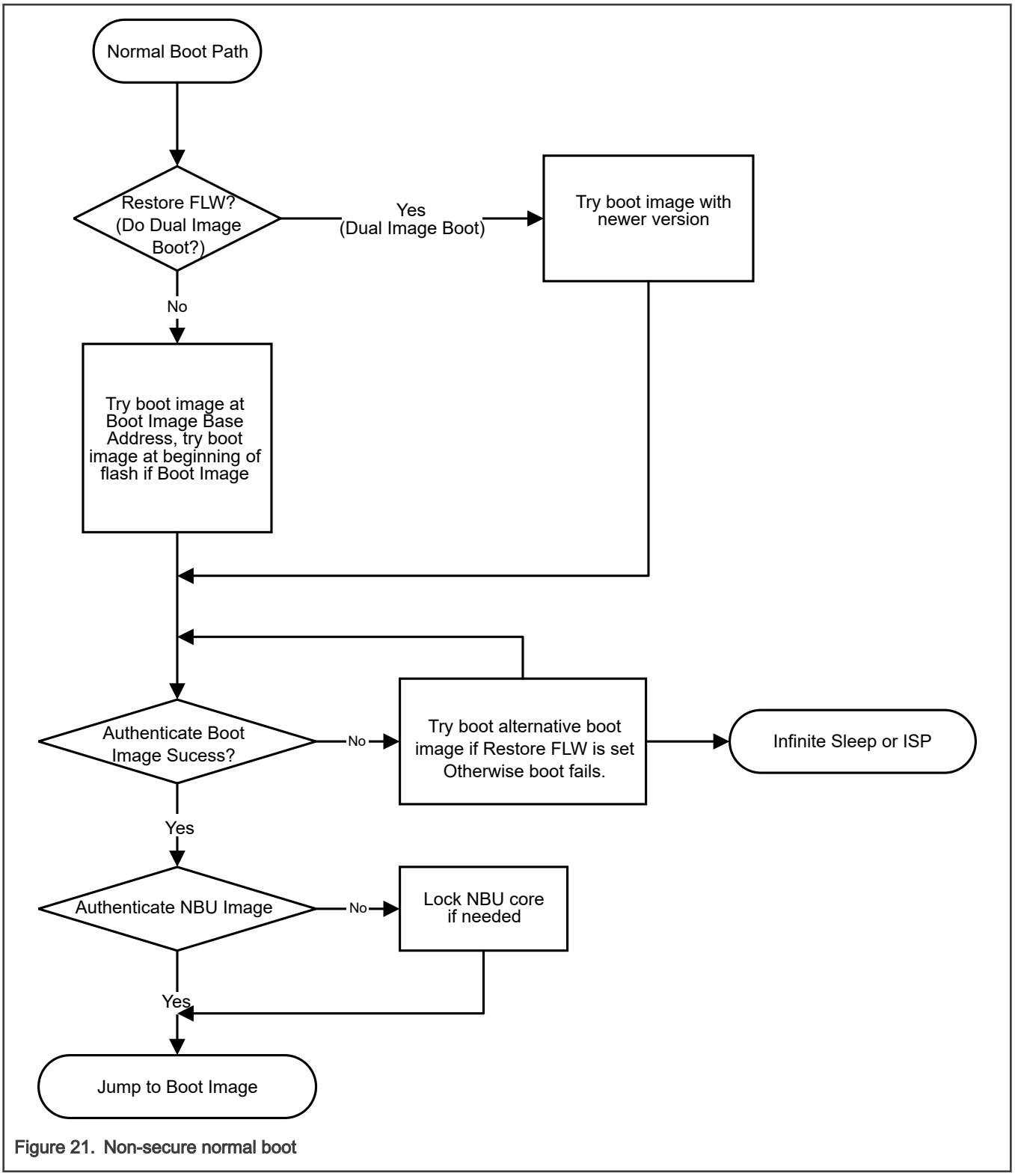


Figure 21. Non-secure normal boot

9.2.3.1 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 address 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.

9.2.4 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 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.

9.2.5 Debugger mailbox path

Debugger Mailbox (DM) is a register-based mailbox operating over Serial Wire Debug Port (SWD). It can be accessed via a Debug Access port (DM-AP) which is always enabled. The external world can send and receive data to/from the ROM bootloader via Debugger Mailbox. One important feature of Debugger Mailbox is that it supports NXP Debug Authentication Protocol, which allows users to enable debug access for CM33 when security is required. Meanwhile, users can utilize Debugger Mailbox to initiate a debugging session when there is no valid boot image.

The communication to the Debugger Mailbox is initiated by the debugger. It does so by setting a re-synchronization request and then resets the chip. ROM bootloader can observe the request and go to Debugger Mailbox Path.

9.2.5.1 Quick guide of using debugger mailbox

To make ROM bootloader enter the Debugger Mailbox, the debugger needs to select DM-AP and set the RESYNCH_REQ bit and the CHIP_RESET_REQ bit to 1 in the CSW register. Following a successful initial re-synchronization and reset, communication by the debugger to the device is achieved using 32-bit DM-AP command writes to the REQUEST register in the Debug Mailbox (DM-AP Commands are shown in table below) The debugger can read results of communications via the RETURN register. Response codes are shown in DM-AP return codes.

The example below shows how to enter debug mailbox and do the communication:

```
// 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
// Write DM-AP Command START DM-AP (1) to REQUEST register (1)
WriteAP 1 1
// Read the return code of START DM-AP command from RETURN register (2), 0x0 means success
ReadAP 2
// Write Get Lifecycle (2) to REQUEST register (1)
WriteAP 1 2
// Read the value of current lifecycle from RETURN register (2)
ReadAP 2
```

When doing debug authentication, Debug Auth Start command needs to be sent first and DAC needs to be read back. Then sending Debug Auth Response command with DAR. Pseudo code below shows how to use debug authentication commands. See Debug authentication for more details.

```
// Assume already in Debugger Mailbox
// Write Debug Auth Start (0x10) to REQUEST register (1)
WriteAP 1 0x10
// Read the data count of DAC
ReadAP 2
// Assume reading back 0x82000000, then the length of DAC is 0x200 word (4-byte)
// Send the ACK_TOKEN with the length of remaining data (word) to receive
WriteAP 1 0x0200A5A5
// Read DAC data
ReadAP 2
WriteAP 1 0x01FFA5A5
ReadAP 2
…
WriteAP 1 0x0001A5A5
ReadAP 2
// Get the whole DAC, generate DAR using DAC, assume length of DAR is 0x400 word (4-byte)
// Write Debug Auth Response (0x11) with DAR length to REQUEST register (1)
WriteAP 1 0x04000011
// Read the ACK_TOKEN from RETURN register (2), should be 0x0400A5A5
```

```
ReadAP 2
// Send DAR data, use 0x00000000 as an example of data
WriteAP 1 0x00000000
// Read the ACK_TOKEN, should be 0x03FFA5A5
ReadAP 2
// Send next DAR data
WriteAP 1 0x00000000
ReadAP 2
…
WriteAP 1 0x00000000
// Complete sending DAR when ACK_TOKEN is 0x0000A5A5
ReadAP 2
```

9.2.5.1.1 DM-AP commands

Commands for the DM-AP are listed below. These would be written to the REQUEST register. Please note that BulkErase/ISP Mode/Set FA Mode can only be issued after debug authentication is successful. These commands are not available in OEM Open.

Table 82. DM-AP commands

Command	ID	Parameter/Response	Description
Start DM-AP	0x01	*Parameters*: None *Response*: 32-bit status	Cause the device to enter DM-AP command mode. This must be done prior to Get Lifecycle command. This command is provided for backwards compatibility and does not need to be used for most commands.
Get Lifecycle	0x02	*Parameters*: None *Response*: 32-bit value	Return the value of current lifecycle.
Bulk Erase	0x03	*Parameters*: None *Response*: 32-bit status	Erase the entire CM33 Program Flash (IFR0 not included).
Exit DM-AP	0x04	*Parameters*: None *Response*: 32-bit status	Cause the device to exit DM-AP command mode. The device returns to normal boot path.
Enter ISP Mode	0x05	*Parameters*: dataWordCount: 0x1 data[0]: ISP mode enum. 0xffffffff – Auto detection 0x1 - LPUART 0x2 - LPI2C 0x4 - LPSPI 0x8 - CAN Others - Reserved *Response*: 32-bit status	Enter specified ISP path.

Table continues on the next page...

Table 82. DM-AP commands (continued)

Command	ID	Parameter/Response	Description
Set FA Mode	0x06	*Parameters*: data[]: FA Request *Response*: 32-bit status	Set the part permanently in Fault Analysis (FA or RMA) mode bit in PFR field, and return the part to NXP for FA/RMA testing. ROM erases customer sensitive assets (key codes) stored in PFR.
Start Debug Session	0x07	*Parameters*: None *Response*: 32-bit status	This command is used to indicate ROM the intention of connecting debugger. ROM bootloader enables debug access (if no debug authentication required) and enters while(1) loop.
Debug Auth Start	0x10	*Parameters*: dataWordCount: 0x0 *Response*: data[] : DAC	Start Debug Authentication Protocol. ROM responds to debugger with *DAC (Debug Authentication Challenge)* message.
Debug Auth Response	0x11	*Parameters*: data[] : DAR *Response*: 32-bit status	Debug Authentication Response.

9.2.5.1.2 DM-AP return codes

Return codes for DM-AP commands are listed below. These commands can be read from the RETURN register. See table below.

Table 83. DM-AP return codes

Return code	Descriptions
0x0000 0000	Command succeeded.
0x0010 0001	Debug mode not entered. This is returned if other commands are sent prior to the “Enter DM-AP” command.
0x0010 0002	Command not recognized. A command was received other than the ones defined above.
0x0010 0003	Command failed.

9.2.5.2 Debugger Mailbox protocol

ROM Bootloader implements debug mailbox protocol to interact with host debug systems over the SWD interface. The protocol has following features:

- Request/response based.
- Support for relatively large command and response data.
- All commands and responses are 32-bit word aligned.
- Supports data above 32 bits by using an ACK_TOKEN that moderates the transfer in 32-bit value chunks.
- Requests and responses use the same basic structure.

9.2.5.2.1 Mailbox registers

The registers in the debug mailbox are shown in Table below. These registers are readable by the CPU and are intended primarily to allow on-chip ROM routines to implement requests from an external debugger.

Table 84. Register overview: DBGMailbox (base address = 0x58000000)

Name	Access	Offset	Description	Reset value
CSW	R/W	0x000	Command and status word	0x0
REQ	R/W	0x004	Request from the debugger to the device	0x0
RETURN	R/W	0x008	Return value from the device to the debugger	0x0
ID	RO	0x0FC	Identification register	0x002A0000

9.2.5.2.1.1 Command and Status Word Register (CSW)

The CSW register contains command and status bits to facilitate communication between the debugger and the device.

Table 85. Command and Status Word register (CSW, offset = 0x000)

Bit	Symbol	Description	Reset Value
0	RESYNCH_REQ	The debugger sets this bit to request a re-synchronization.	0x0
1	REQ_PENDING	A request is pending for the debugger: a value is waiting to be read from the REQUEST register.	0x0
2	DBG_OR_ERR	When 1, a debug overrun has occurred: a REQUEST value has been overwritten by the debugger before it was read by the device.	0x0
3	AHB_OR_ERR	When 1, an AHB overrun has occurred: a RETURN value has been overwritten by the device before it was read by the debugger.	0x0
4	SOFT_RESET	Setting this write-only bit in the CSW register of the DM-AP by an external debugger resets the DM-AP state machine and its registers. While setting RESYNCH_REQ only resets the DP and CPU handshake state machine.	0x0
5	CHIP_RESET_REQ	This write-only bit causes the device (but not the DM-AP) to be reset by generating SYSRESET_REQ.	0x0
31:6	Reserved		-

9.2.5.2.1.2 Request value register

The REQUEST register is used by a debugger to send action requests to the device.

Table 86. Request value register (REQUEST, offset = 0x004)

Bit	Symbol	Description	Reset Value
31:0	REQUEST	Request value. Reads as 0 when no new request is present. Cleared by the device. Can be read back by the debugger in order to confirm communication.	0x0

9.2.5.2.1.3 Return value register

The RETURN register provides any response from the device to the debugger.

Table 87. Return value register (RETURN, offset = 0x008)

Bit	Symbol	Description	Reset Value
31:0	RET	Return value. This is any response from the device to the debugger. If no new data is present, a debugger read will be stalled until new data is available.	0x0

9.2.5.2.1.4 Identification register

The ID register provides an identification of the DM-AP interface.

Table 88. Identification register (ID, offset = 0x0FC)

Bit	Symbol	Description	Reset Value
31:0	ID	Identification value.	0x002A0000

9.2.5.2.2 Mailbox commands

9.2.5.2.2.1 Request

The first word transmitted in a request is a header word containing the command ID and number of following data words. Following the header are the number of 32-bit words specified in the header.

Table 89. 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 as follows:

```
struct dm_request {
  uint16_t commandID;
  uint16_t dataWordCount;
  uint32_t data[]; };
```

9.2.5.2.2.2 Response

The first word transmitted in a response is a header word containing the command status and number of following data words.

- To support legacy command and response value, Bit_31 in header will be used to indicate that the response follows new protocol structure.

Table 90. Response register byte description

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	commandStatus[7:0]	commandStatus[15:8]	dataWordCount[7:0]	dataWordCount[14:8] new_protocol[15]
1	*data…*			

The C structure definition for a response is as follows:

```
struct dm_response {
    uint16_t commandStatus;
    uint16_t dataWordCount;
    uint32_t data[];
};
```

9.2.5.2.2.3 ACK_TOKEN

- When command has parameters the debugger should wait for ACK_TOKEN (sent through DBG_MB_RETURN register) before sending next 32-bit value.
- Similarly when response packet has data to send back to debugger, ROM will wait for debugger to send ACK_TOKEN (sent through DBG_MB_REQUEST register) before sending next 32-bit value.
- Upper 16 bits are set by receiving end with number of remaining words expected.
- Lower 16 bits are always set to 0xA5A5.

Table 91. ACK_TOKEN register byte description

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	0xA5	0xA5	remainCount[7:0]	remainCount[15:8]

The C structure definition for a ACK_TOKEN is as follows:

```
struct dm_ack_token {
    uint16_t token; /* always set to 0xA5A5 */
    uint16_t remainCount; /* count of remaining word */};
```

9.2.5.2.2.4 Error handling

If an overrun occurs from either side of the communication, the appropriate error flag is set in the CSW. The state machine hardware will prevent further communication in any direction once an overrun has occurred in that direction, so if such an error occurs, the debugger will need to start with a new re-synchronization request in order to clear the error flag.

9.2.5.3 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. Thus, 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 ROM bootloader 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 scheme is a challenge-response scheme and assures that debugger in possession of required debug credentials only can successfully authenticate over debug interface and access restricted parts of the device. This protocol is divided into steps as shown in description and figure below.

1. The debugger initiates the Debug Mailbox message exchange by setting the RESYNCH_REQ bit and CHIP_RESET_REQ bit in the CSW register of DM-AP.
2. The debugger waits (minimum 30ms) for the devices to restart and enter debug mailbox request handling loop.

3. The debugger sends Debug Authentication Start command (command code 0x10) to the device.
4. The device responds back with Debug Authentication Challenge (DAC) packet based on the debug access rights pre-configured in CMPA fields, 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.

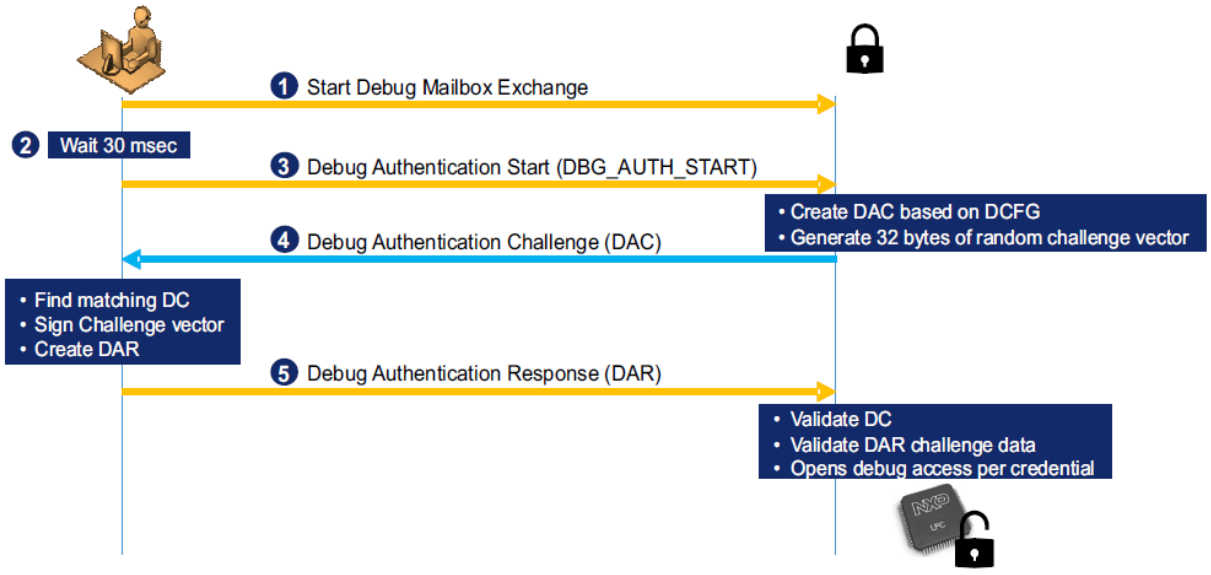


Figure 22. Debug authentication flow

9.2.5.3.1 Debug access control configuration

The ROM bootloader handles the device side of 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.
- DCFG_VENDOR_USAGE: 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 field for revocation of already issued debug certificates.

9.2.5.3.1.1 Protocol Version (DCFG_VER)

This device supports Debug authentication protocol version 2.1, which are defined based on the secp384r1 curve.

NOTE

Both debug authentication certificates and image signing certificates use same Root of Trust keys (RoTK).

9.2.5.3.1.2 Root of Trust Identifier (DCFG_ROTID)

The Root of Trust Identifier used in debug authentication protocol is composed of two elements.

- A 256-bit cryptographic hash (SHA256 or leading 32 bytes of SHA384, according root certificates EC types, secp256r1->SHA256 or secp384r1->SHA384) over the Root of Trust Keys hash table. This is the same as Root of Trust Key Table Hash (RoTKTH) field referred in Security Features of Boot ROM. RoTKTH is a 32-byte SHA256 or 48-byte SHA384 hash of up to four RoTKs.
- A 32-bit field containing revocation bits for the four Root of Trust keys in the table.

9.2.5.3.1.3 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 16) in CC_SOCU.

This device-specific constraint, if enabled, is in addition to all the other constraints defined and enforced by the authentication protocol

9.2.5.3.1.4 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 is sub-divided into multiple debug domains to allow finer access control. Table 93 shows debug domains and their corresponding control bit position in DCFG_CC_SOCU. Which is logically composed of two components:

- SOCU_PIN (bit 7-0 in CC_SOCU): A bitmask that specifies which debug domains are predetermined by device configuration.
- SOCU_DFLT (bit 15-8 in CC_SOCU): 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 92. Access restriction levels

Restriction Level	SOCU_PIN [n]	SOCU_DFLT [n]	Description
0	1	1	Access to the sub-domain is always enabled. This setting is provided for module use case scenario where DCFG_CC_SOCU_NS would be used to define further access restrictions before final deployment of the product.
1	0	0	Access to the sub-domain is disabled at start up. But the access can be enabled through debug authentication process by providing appropriate Debug Credential (DC) certificate.
2	0	1	Illegal setting. Part may lock-up if this setting is selected.

Table continues on the next page...

Table 92. Access restriction levels (continued)

Restriction Level	SOCU_PIN [n]	SOCU_DFLT [n]	Description
3	1	0	Access to the sub-domain is permanently disabled and cannot be reversed. This setting offers the highest level of restriction.

Debug domains supported by this device are shown in table below.

Table 93. CC_LIST_Table

Bit[1]	Symbol	Description
0	PIN_NIDEN	Controls non-Invasive debugging of TrustZone for Arm8-M defined non-secure domain of CM33.
1	PIN_DBGEN	Controls invasive debugging of TrustZone for Arm8-M defined non-secure domain of CM33.
2	PIN_SPNIDEN	Controls non-Invasive debugging of TrustZone for Arm8-M defined secure domain of CM33.
3	PIN_SPIDEN	Controls invasive debugging of TrustZone for Arm8-M defined secure domain of CM33.
4	PIN_NBU_DBGEN	Controls debugging of NBU (radio, CM3)
5	PIN_FA_CMD_EN	Controls whether DM-AP command, Set FA Mode (command code: 0x06), can be issued through after authentication.
6	PIN_ISP_CMD_EN	Controls whether ISP boot flow DM-AP command (command code: 0x05) can be issued after authentication.
7	PIN_ME_CMD_EN	Controls whether DM-AP command, Bulk Erase (command code: 0x02), can be issued through after authentication.
8	DFLT_NIDEN	Controls non-Invasive debugging of TrustZone for Arm8-M defined non-secure domain of CM33.
9	DFLT_DBGEN	Controls invasive debugging of TrustZone for Arm8-M defined non-secure domain of CM33.
10	DFLT_SPNIDEN	Controls non-Invasive debugging of TrustZone for Arm8-M defined secure domain of CM33.
11	DFLT_SPIDEN	Controls invasive debugging of TrustZone for Arm8-M defined secure domain of CM33.
12	DFLT_NBU_DBGEN	Controls debugging of NBU (radio, CM3)
13	DFLT_FA_CMD_EN	Controls whether DM-AP command, Set FA Mode (command code: 0x06), can be issued through after authentication.
14	DFLT_ISP_CMD_EN	Controls whether ISP boot flow DM-AP command (command code: 0x05) can be issued after authentication.

Table continues on the next page...

Table 93. CC_LIST_Table (continued)

Bit[1]	Symbol	Description
15	DFLT_ME_CMD_EN	Controls whether DM-AP command, Bulk Erase (command code: 0x02), can be issued through after authentication.
16	UUID_CHECK	Used for UUID matching required flag.
17:31	Reserved	Reserved

1. Bits [7:0] is SOCU pinned value, and bits [15:8] is SOCU default value.

9.2.5.3.1.5 DCFG_VENDOR_USAGE

This field can be used to define vendor specific debug policy use case such as Debug Credential (DC) certificate revocations or department identifier or model identifier. During Debug Authentication Response (DAR) processing the device checks that the value specified in Vendor Usage field of DC matches exactly the value programmed in DBG_AUTH_VU fields of device configurations. During debug authentication this value has to mach cc_vu value in debug credentials. For example user can define DBG_AUTH_VU for every product and then issue debug credential just for specific product.

9.2.5.3.2 Debug Authentication Challenge (DAC)

The debug authentication protocol begins with a DEBUG_AUTH_START request, issued by the debugger to the device, which replies by Authentication Challenge (DAC) message, issued by the device to the debugger. The DAC carries information about current device state to the debugger, inputs from DAC are necessary to find existing or request generation of new Debug Credential (DC). Protocol version 2.x uses ECC for digital signatures: minor version 1 (“2.1”) points to secp384r1.

Structure of DAC message is shown in following Figure.

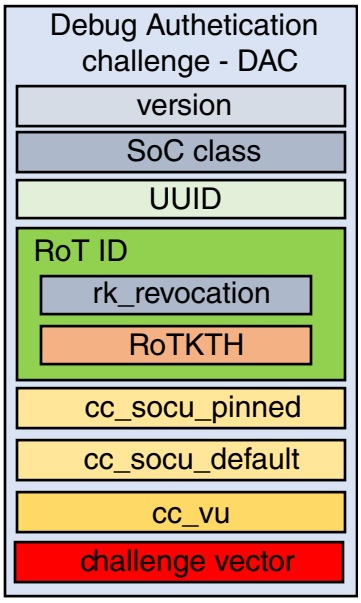


Figure 23. Debug Authentication Challenge (DAC)

- **version** – uint32_t, little endian

  Fixed to value 0x02000100u "2.1" (major =2, minor = 1) -> secp384r1 used as RoTKTH

- **SoC class** – uint32_t, little endian

  Fixed to 0x5u for this device, identify the device family

- **UUID** – uint32_t[4], little endian

  Unique ID of device.

- **RoT ID** – object

  Identifier of root of trust settings in device.

  — **rk_revocation** – uint32_t, little endian

  1 bit per RoTK (up to 4 RoTK can be defined for device, bit is 1 – RoTK is revoked (not usable for debug auth), bit is 0 – RoTK can be used.

  rk_revocation[bits 31:4] -> reserved for future use.

  rk_revocation[bit 3] -> bit value 0 = RoTK[3] OK / bit value 1 = RoTK[3] revoked

  rk_revocation[bit 2] -> bit value 0 = RoTK[2] OK / bit value 1 = RoTK[2] revoked

  rk_revocation[bit 1] -> bit value 0 = RoTK[1] OK / bit value 1 = RoTK[1] revoked

  rk_revocation[bit 0] -> bit value 0 = RoTK[0] OK / bit value 1 = RoTK[0] revoked

  Refer to Keys and Certificate block for more information about RoTK.

  — **RoTKTH** – uint8_t[32]

  RoTKTH (Root of Trust Key Table Hash) value is stored in device fuses (Refer to Keys and Certificate block for more information about RoTKTH). If RoTKTH size is 48 bytes (RoTKTH is SHA384 digest of the RoT Key Table), then this field value carries only leading 32 bytes of RoTKTH full value.

- **cc_socu_pinned** – uint32_t, little endian

  Actual value of CC_SOCU_PINNED in device, which is based on actual device lifecycle, DCFG_CC_SOCU_L1 and DCFG_CC_SOCU_L2 fuses combination. For bit representation of CC_SOCU_PINNED please refer to Table 92.

- **cc_socu_default** – uint32_t, little endian

  Actual value of CC_SOCU_DEFAULT, which is based on actual device lifecycle, DCFG_CC_SOCU_L1 and DCFG_CC_SOCU_L2 fuses combination. For bit representation please refer to Table 92.

- **cc_vu** – uint32_t, little endian

  Vendor usage value, custom value provided by vendor, stored in DBG_AUTH_VU fuse in device, value needs to be the same as value specified present later in debug certificate (DC), used for pairing of corresponding DC.

- **challenge vector** – uint8_t[32]

  Random number generated for each request to be signed in debug authentication response (DAR) to avoid reusing of DAR.

9.2.5.3.3 Debug Credential (DC)

Prior to generating debug credential, it is necessary to generate EC keypair (secp384r1) for debugger known as Debug Credential Key (DCK). The public key part of DCK (DCK_{public}) is used to represent the identity of the debugger through the creation of a DC, which binds that public key to some usage attributes (UUID, CC_SOCU), and is then authorized/signed by the vendor's Roots of Trust key ($RoTK_{private}$) using ECDSA algorithm. If RoTK is based on secp384r1then DCK also needs to be based on secp384r1. Combining of EC types is not allowed in DC.

The private part of DCK ($DCK_{private}$) will be used later for signing of Debug Authentication Response (DAR). Refer to Keys and Certificate block for more information about RoTK.

The DC structure is illustrated in the following figure.

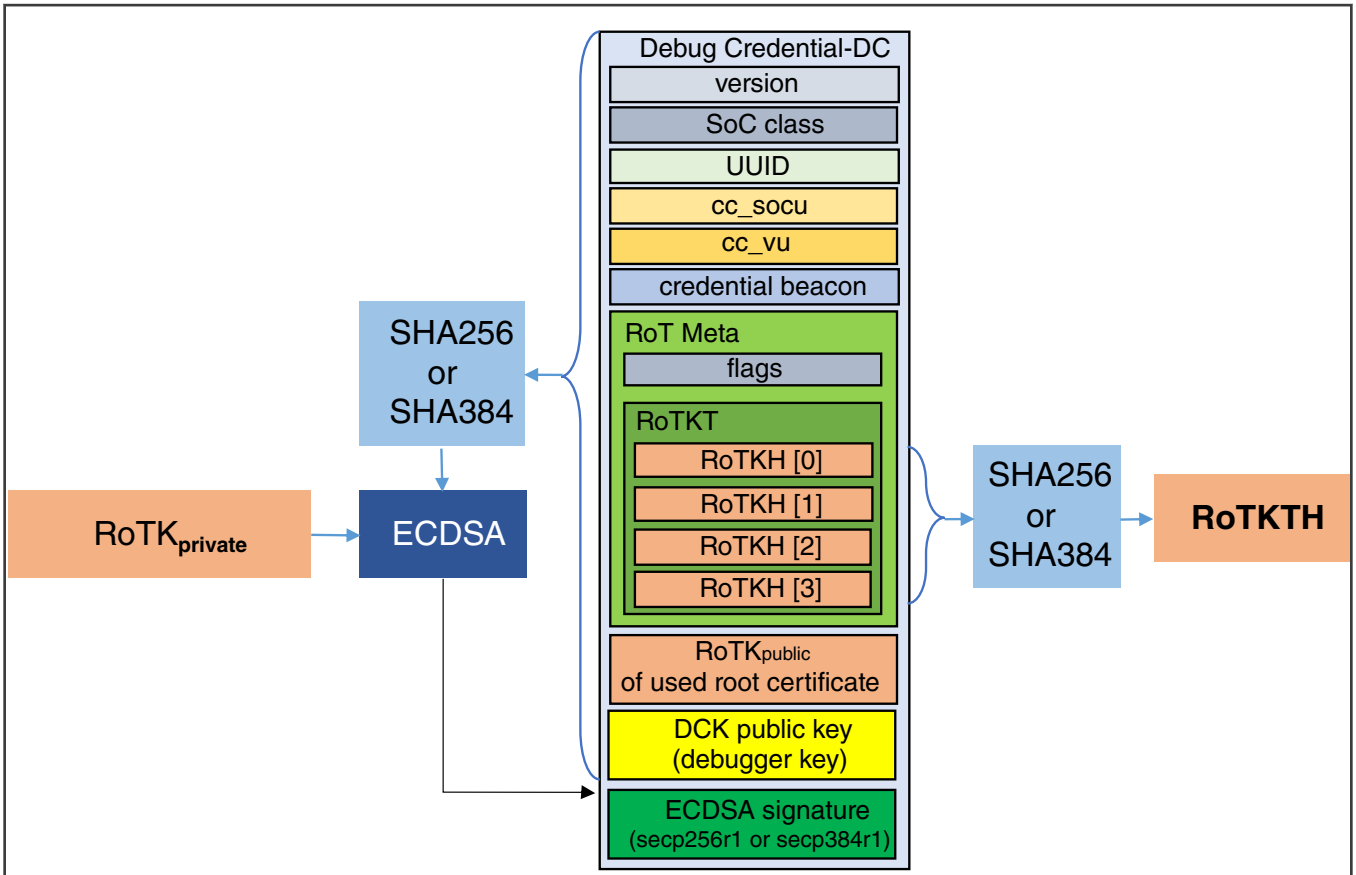


Figure 24. Debug Credential (DC)

The data structure is represented as a packed binary concatenation of its component fields as shown in the list below:

- **version** – uint32_t, little endian

  Fixed to value 0x02000100u “2.1” (major =2, minor = 1) -> secp384r1 used as RoTK

- **SoC class** – uint32_t, little endian

  Fixed to 0x5u, identify the device family.

- **UUID** – uint32_t[4], little endian

  Unique ID of device. Can be set to 0 when debug credential is used for SoC class matching (one debug credential can be used to open debug ports on all devices with same CTRK and same SoC class, UUID matching in device have to be disabled – default). If UUID matching in device is enabled, then UUID value needs to correspond with UUID value of the device received in DAC.

- **cc_socu** – uint32_t, little endian

  Proposed value of SOCU, will be validated and combined on device with current lifecycle, DCFG_CC_SOCU_L1 and DCFG_CC_SOCU_L2 fuses values to have final state of debug ports, which will be applied on device. Refer to Table 92 for Bit details.

- **cc_vu** – uint32_t, little endian

  Vendor usage value need to be matched with value of DBG_AUTH_VU in device fuses (received on DAC).

- **credential beacon** – uint32_t, little endian

  Value loaded to the beacon register after successful debug authentication. Value is used by application to control debug ports. Mask 0xFFFFu is applied to the value of credential beacon and is connected with authentication beacon value from DAR as following formula: beacon = (credentialBeacon & 0xFFFFu) | ((authenticationBeacon & 0xFFFFu) << 16). If final

beacon value is non zero, debug ports are not enabled by ROM bootloader itself and debug ports configuration is delegated to booted application, which can control debug ports according final beacon value available in register (e.g. third party secondary bootloader will open debug just before jump to loaded application, debug of secondary bootloader can be enabled only by special beacon value of credential beacon).

- **RoT meta** – object

  Device root of trust meta data object.

  - **flags** – uint32_t, little endian

    Configuration details of RoT meta object, needs to be matched with RoT ID received in DAC.

    - **flags[bit 31]**: CA flag, always 1 in DC.
    - **flags[bits 30:12]**: Reserved for future use.
    - **flags[bits 11:8]**: Used root cert number [0-3] (specify RoTK used to DC signature), used only when more than 1 RoTK specified for device.
    - **flags[bits 7:4]**: Number of records in RoTKT [1-4], when equal to 1, then RoTKT not present in DC.
    - **flags[bits 3:0]**: Reserved for future use.

  - **RoTKT** – object

    Root of Trust keys Table, optional object (in case when is specified only one RoTK for device, RoTKT is not present, 1 to 4 RoTK can be specified for device), so if RoTKT is present in DC, then RoTKT table contains at least two RoTKH and maximally 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 update container and signed boot image. Refer to Keys and Certificate block for more information about RoTK and RoTKTH.

- **$RoTK_{public}$** – 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 DC, field size depends on EC type (secp256r1 or secp384r1)

- **DCK_{public}** – uint8_t[64] or uint8_t[96]

  X and Y coordinates of public key of DCK, field size depends on EC type of DCK key (secp256r1 or secp384r1). DCK and RoTKs must be defined on same EC type, combination of EC types is not allowed.

- **ECDSA signature** – uint8_t[64] or uint8_t[96]

  SHA256 or SHA384 of all Debug Credential data (as described on previous figure) used for ECDSA signature. Hash algorithm selected according to EC type of RoTK (secp256r1 -> SHA256 or secp384r1 -> SHA384). Signature coordinates (r,s) stored in big-endian.

9.2.5.3.4 Debug Authentication Response (DAR)

Debugger will find existing or request generation of new corresponding DC according to the inputs from DAC message. The $DCK_{private}$ key is used as input for calculation of ECDSA signature of the corresponding DC including device UUID and Challenge vector from received DAC (Challenge vector is not part of DAR, but it is used as last input to hash of DAR). For signature used SHA256 or SHA384 hash algorithm based on DCK EC type (secp256r1 or secp384r1), if secp256r1-> SHA256, if secp384r1->SHA384. The DAR structure and generation process is described in following figure.

Because regular debuggers usually don't offer feature for ECDSA signing, some PC tool communicating with debugger is needed. NXP offers tool nxpcertgen.exe for generation of DC and nxpdebugmbox.exe for full debug authentication process. Usage of mentioned tools is described in separate user manual related to them.

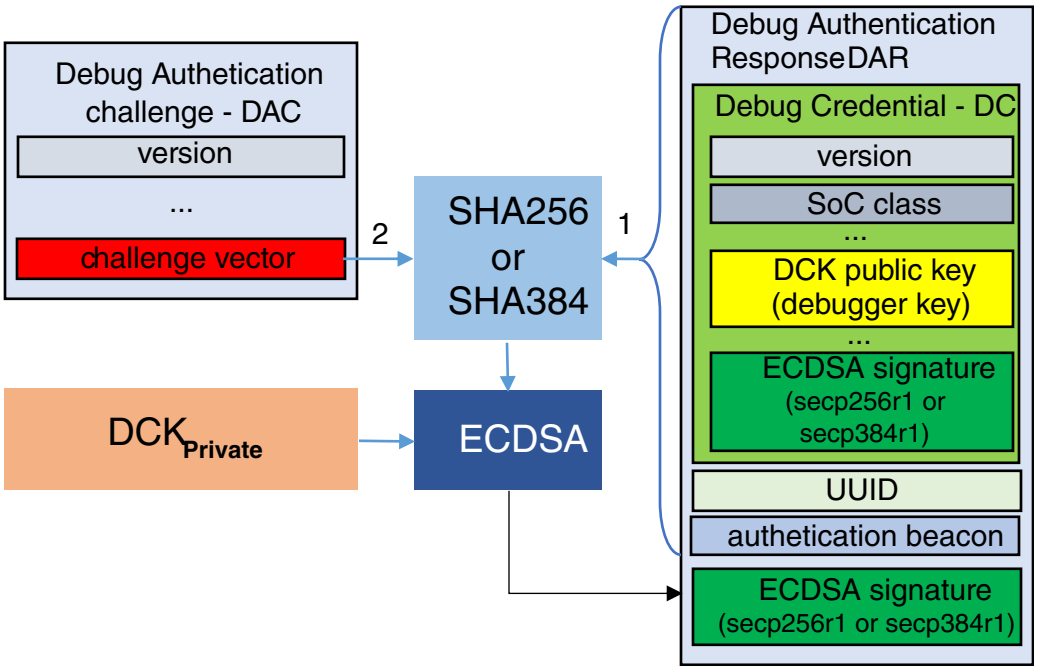


Figure 25. Debug Authentication Response (DAR)

- **Debug credential** – object

  Matched debug credential based on data received from DAC message.

- **authentication beacon** – uint32_t, little endian

  Value loaded to the beacon register after successful debug authentication. Value is used by running application to control debug ports. Mask 0xFFFFu is applied to the value and is connected with credential beacon value from DC as following formula: beacon. Beacon = (credentialBeacon & 0xFFFFu) | ((authenticationBeacon & 0xFFFFu) << 16). If final beacon value is non zero, debug ports are not enabled by ROM bootloader itself and debug ports configuration is delegated to booted application, which can control debug ports according final beacon value available in register (e.g. third party secondary bootloader will open debug just before jump to loaded application, debug of secondary bootloader can be protected by special beacon value).

- **UUID** – uint32_t[4], little endian

  Unique ID of device, must be same as UUID value provided by device in DAC message

9.2.5.3.5 Device processing the DAR

The device BootROM will process DAR received from debugger. As a part of the validation step, device will:

1. Verify DC: Validate DC version, SoCC, UUID, RoT, VU, and DC signature.
2. Verify that the DAR has a valid signature that binds it to the CV from the DAC.

If all the steps are successfully completed, it can be deduced that:

- The debugger possesses the private key corresponding to the vendor/RoT-signed credential.
- The credential satisfies the constraints specified in the device configuration.
- The response of the debugger to the challenge from the device 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 DAR, if authentication is successful, Debug Access will be granted. If authentication fails, no special response is issued but further debug access request will be ignored, and device will enter in a failure loop (infinite loop waiting for debug attach)

9.2.5.3.6 Debug authentication use cases

9.2.5.3.6.1 Return Material Analysis (RMA) use case

The diagram below shows RMA flow using debug authentication scheme, where a debug credential certificate is issued for each field technician.

1. Vendor generates RoT key pairs and programs the device with SHA256 hash of RoT public key hashes before shipping
2. Field technician generates his own key pair and provides public key to vendor for authorization.
3. Vendor attests the field technician's public key. In the debug credential certificate, vendor assigns the access rights.
4. End customer having issues with a locked product takes it to field technician.
5. Field technician uses his credentials to authenticate with device and un-locks the product for debugging.

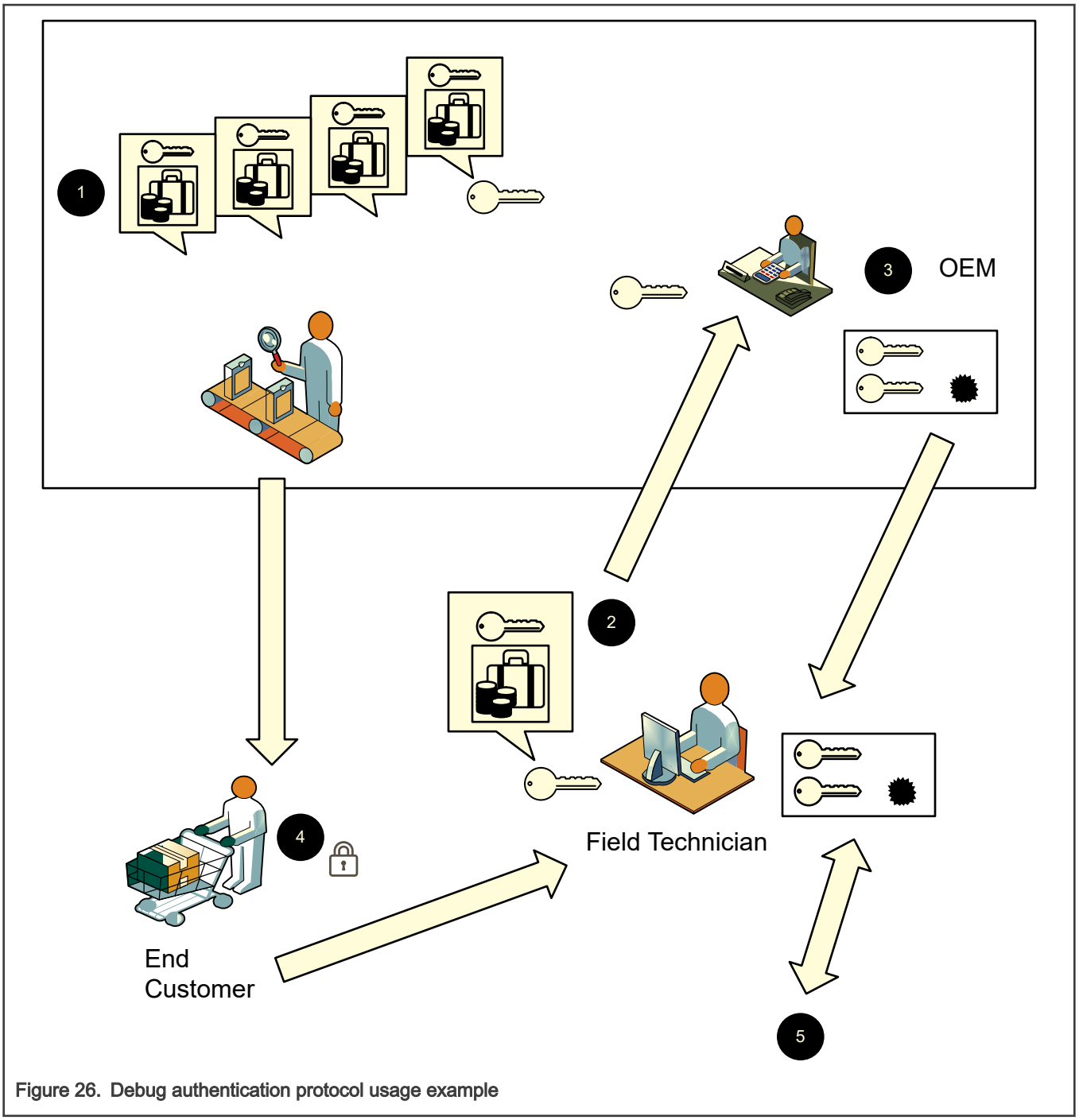


Figure 26. Debug authentication protocol usage example

9.2.5.3.6.2 Module use case with OEM tier1 and tier2 Lifecycle states

The CC_SOCU_PIN_NS & CC_SOCU_DFLT_NS is provided to allow module-maker and OEM using the module to implement tiered protection approach.

- The module maker who is referred as Tier-1 developer can develop secure code and define access rights to his module using CC_SOCU_PIN & CC_SOCU_DFLT.
- Configuration can be such that debug access to secure module is blocked but non-secure debug is always allowed.
- Once the module is ready, Tier-1 developer can release the module to OEM (a.k.a. tier-2 developer), but block debug access to secure mode and enable debug access to non-secure mode.

- Tier-2 developer can develop non-secure module and extend access rights configuration to that module using CC_SOCU_DFLT_NS and CC_SOCU_PIN_NS.

9.2.5.3.7 Fault Analysis (FA) mode

The ROM bootloader offers an FA Mode (Set FA Mode) command handler to enable to delete sensitive information (for example, Keys) before handling over the device to OEM for fault analysis. ROM allows Set FA Mode command only when corresponding flag, FA_CMD_EN, is set. Users must pass an FA request message to Set_FA_Mode command to enter FA mode.

Set FA Mode command will perform the following:

- Erase all OEM Keys in fuse
- Erase the internal flash
- Flush all temporary key registers
- Move the lifecycle to OEM Return

9.2.6 Low-power wakeup path

ROM bootloader goes to low-power wakeup path if the register SPC0->WAKEUP is set to a non-zero value, WAKEUP_DIS is 0 indicating wakeup path is not disabled and Sticky NPX Configuration for Waking up from Deep Power Down is not set. Please note that ROM bootloader does not configure/service/disable Watchdog for low power wakeup path. And ELE is not released for this path. Figure below shows the flow of low power wakeup path.

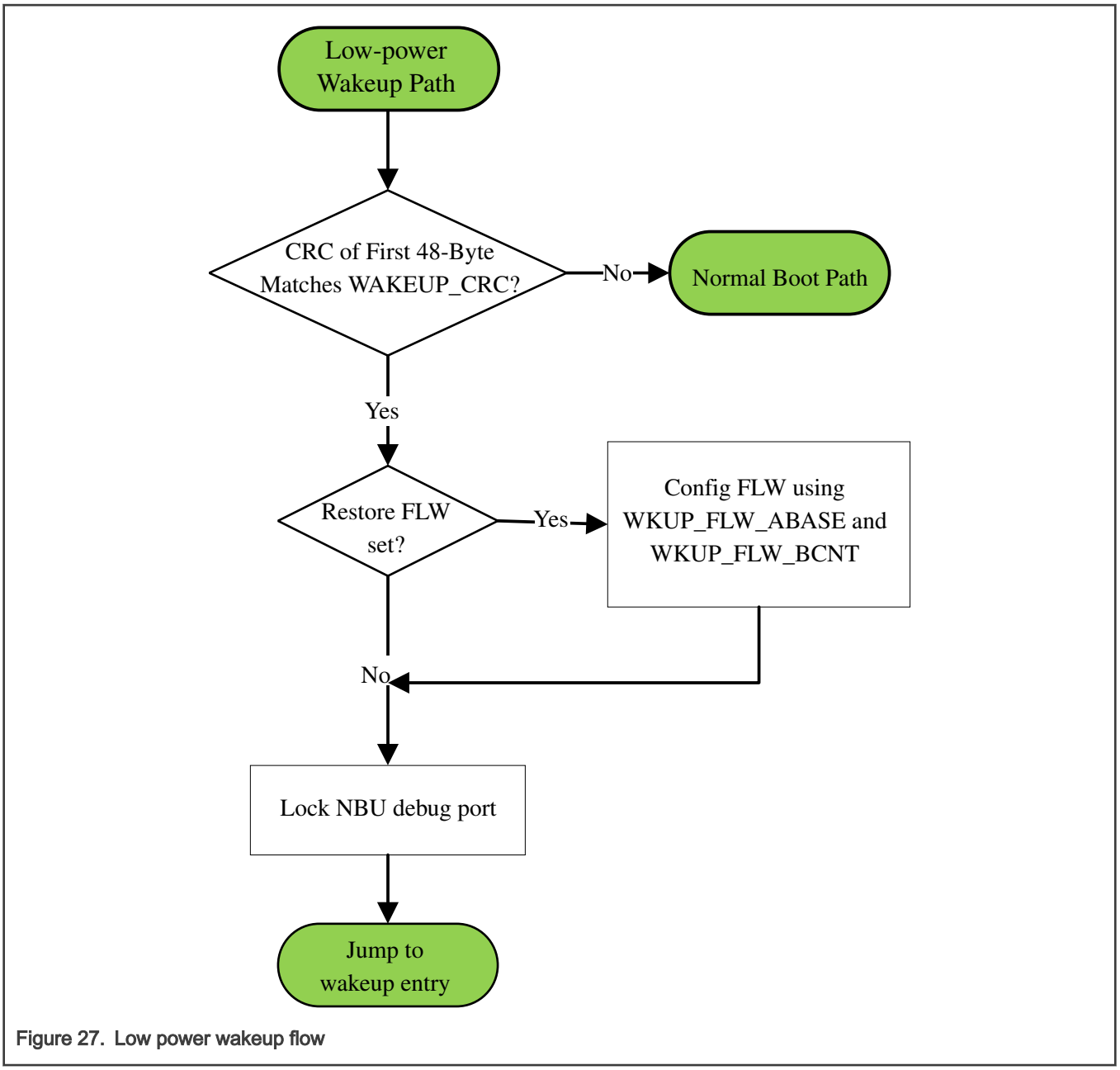


Figure 27. Low power wakeup flow

If WAKEUP_CRC in REGFILE1 (REG0 of REGFILE1) is set correctly, matching the CRC for first 48 bytes of wakeup image, ROM bootloader jumps to the wakeup process entry stored in SPC0->WAKEUP. Otherwise, ROM bootloader exits low power wakeup path and go to normal boot path. Please note that CM3 debug port is locked for low power wakeup path.

In addition, if Flash Logic Window needs to be configured for low power wakeup path, Restore FLW Flag, WAKEUP_FLW_ABASE (REG1 of REGFILE1) and WAKEUP_FLW_BCNT (REG2 of REGFILE1) need to be set correctly. ROM bootloader will configure the FLW using the value from WAKEUP_FLW_ABASE and WAKEUP_FLW_BCNT registers. Please note that Wini_applicatiohalting_executing_immediately_following_rom_AKEUP_FLW_BCNT has three fields, including Valid bit (bit 31), Lock bit (bit 30) and Block Count (bit 0 to bit 14).

9.2.7 Radio firmware update feature

Boot ROM can perform a firmware update remotely. It can be used for updating main flash (CM33) as well as radio flash (CM3) firmware.

Use case example of remote radio firmware update:

1. Main flash (CM33) 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 master mode and pin assignment is showed in table below.

Table 94. 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
LPSPI1_SOUT	PTB3	2
LPSPI1_DATA2	PTB5	2
LPSPI1_DATA3	PTB4	2
LPSPI1_SS0	PTB0	2

9.2.8 Initiate a debug session

When ROM bootloader is executing (i.e., instructions are fetched from the ROM memory address range), the debug access port (AP) of CM33 is disabled irrespective of device life-cycle state or token settings. This mechanism is referred as 'Boot-ROM protection' in this document. Thus, the method to initiate a debug session will vary depending on the device state and intended debug scenario. The scenarios described in the rest of these section are as follows:

- Flash does not contain a valid, bootable image. If the flash does not contain a valid image, the ROM bootloader will proceed to ISP mode and wait to be connected via one of its serial interfaces; in ISP mode the debug interface is disabled.
- ISP mode initiated because the ISP pin was asserted on the device at reset.
- Connection to a device running a valid application, with the intent to update flash with a new application.
- Connection to a device running a valid application in flash, with the intent to debug without updating flash (also called a "debug attach").

9.2.8.1 Debug session with invalid flash image or ISP mode

When the device boots, there may be no valid image in the boot media, at which point the ROM bootloader enters the In-System Programming (ISP) path, and debug access is disabled for security reasons. Another scenario is where the device may be placed into ISP mode because the ISP has been asserted as the device leaves reset.

To start a debug session under these scenarios, users need to enter Debugger Mailbox and send the DM-AP command Start Debug Session. Upon receiving the command, the ROM bootloader ensures any unwanted peripheral interrupts are disabled and secrets managed before enabling debug access. After configuring debug access, the ROM enters a while (1) loop. For lifecycle which does not require debug authentication, once the Start Debug Session has been successfully executed, the AP for CM33 and CM3 will be accessible and can used to set breakpoints, etc. as with other Cortex-M devices. For lifecycle which requires debug authentication, DM-AP commands Debug Auth Start and Debug Auth Response need to be done successfully before sending Start Debug Session command to ensure AP for CM33 or CM3 can be accessible.

Example below shows how a debug session is initiated at lifecycle which debug authentication is not required:

```
// 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
// 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
```

9.2.8.2 Debug session with valid application in flash

In this scenario, ROM bootloader configures debug access before jumping to the application in flash. If the device is at some lifecycle debug authentication is not required and debug has not been disabled by an application already in flash, the APs of CM33 and radio (CM3) are accessible, a debug session can be initialized without any other operation. If debug authentication is required, entering Debugger Mailbox, doing the debug authentication successfully and exiting Debugger Mailbox need to be done before ROM bootloader booting the application. By doing so, users can initialize a debug session after ROM bootloader jumping to the application in flash.

9.2.9 Clock

9.2.9.1 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.

9.2.9.2 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

9.3 Security Features of Boot ROM

The Secure part of ROM boot loader 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.

9.3.1 Function description

9.3.1.1 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.

9.3.1.2 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.

9.3.1.3 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. Refer to Additional fuse fields of interest 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.

9.3.1.4 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 many nested sub-modules are required, 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.

9.3.1.5 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

9.3.2 Keys

There are four types of secure boot keys.

- Root of Trust Key (RoTK)
  - Public part of RoTK ($RoTK_{public}$)
  - Private part of RoTK ($RoTK_{private}$)

- 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.

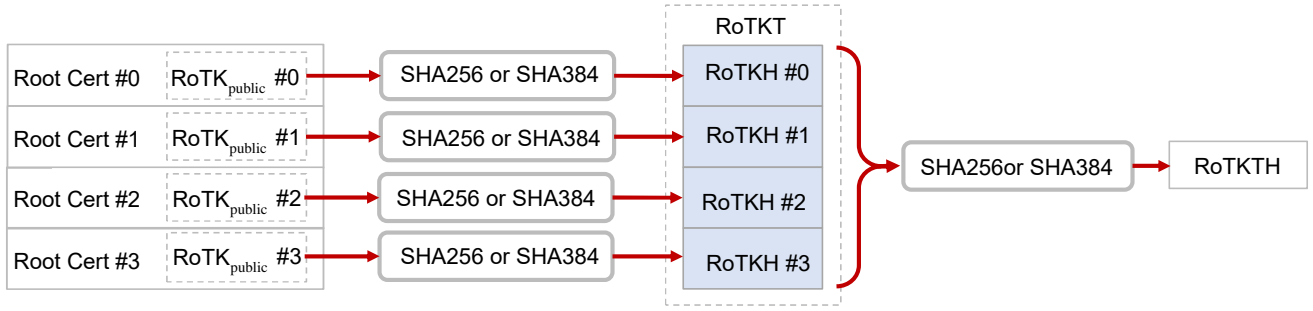


Figure 28. RoTKTH calculation

If only 1 RoTK is specified for RoTKTH, then the RoTKTH value is directly SHA256 or SHA384 of RoTK$_{public}$. If more than 1 root certificates is specified, RoTKTH is calculated as hash of hashes of RoTK$_{public}$, as shown in Figure 28. 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.

9.3.3 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.

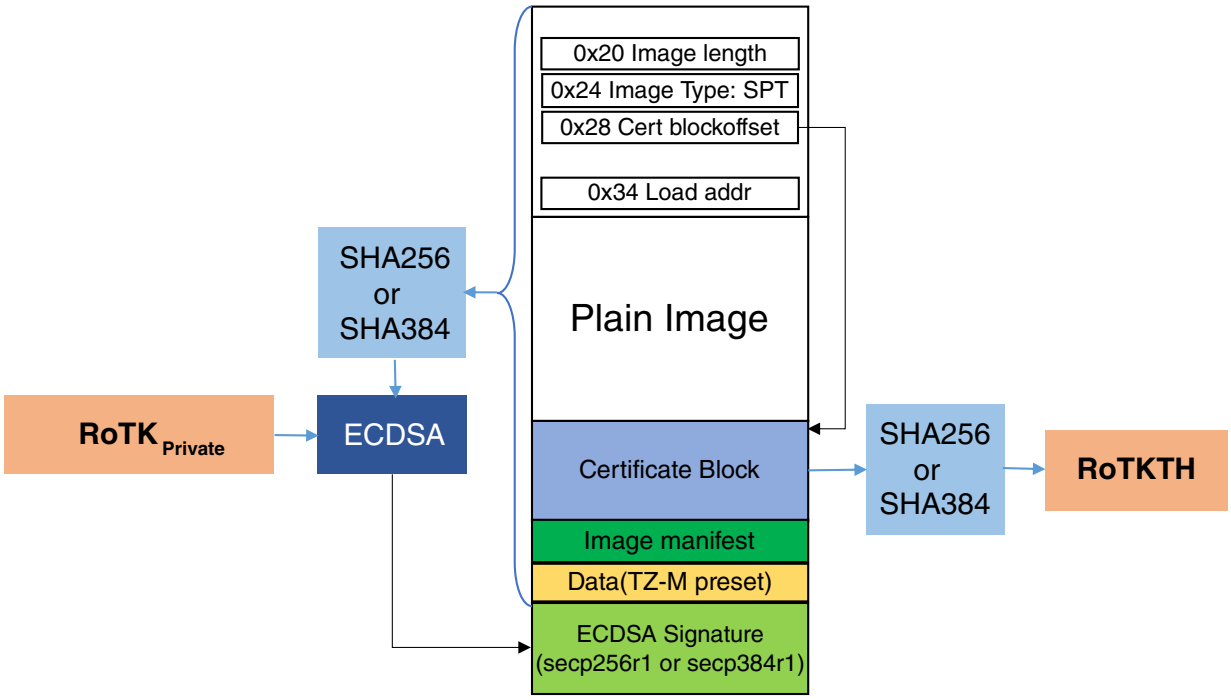


Figure 29. nboot signed image format

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

Table 95. Boot image vector table

Offset	Field	Size (bytes)	Description
0x00	Arm cortex M33 specific	32	Arm specific data
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 96. Details of imageType (word at offset 0x24)

Bit 31 -- bit 16	Reserved	shall be set to 0

Table continues on the next page...

Table 96. Details of imageType (word at offset 0x24) (continued)

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 6	Image subtypes	Used to distinguish between “XIP plain signed NXP” (image type 0x8) for CM33 and NBU core: 0x0: CM33 XIP plain signed NXP 0x1: NBU XIP plain signed NXP Other values and combinations with image type are reserved.
Bit 5 -- bit 0	Image Type	0x0: plain image 0x4: Xip plain signed 0x5: Xip plain with CRC 0x6: SB3 manifest 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).

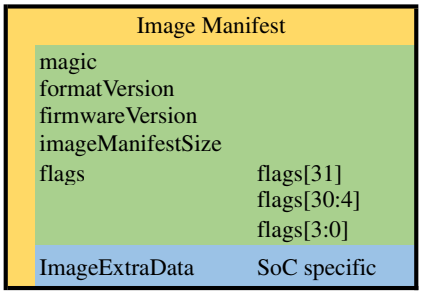


Figure 30. Image Manifest

Table 97. Image Manifest

Image Manifest	Offset	Data Type (little endian)	Description
magic	0x00u	uint32_t	Fixed 4-byte string "imgm" without the trailing NULL, 0x6D676D69u.

Table continues on the next page...

Table 97. Image Manifest (continued)

Image Manifest	Offset	Data Type (little endian)	Description
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.

9.3.3.1 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.

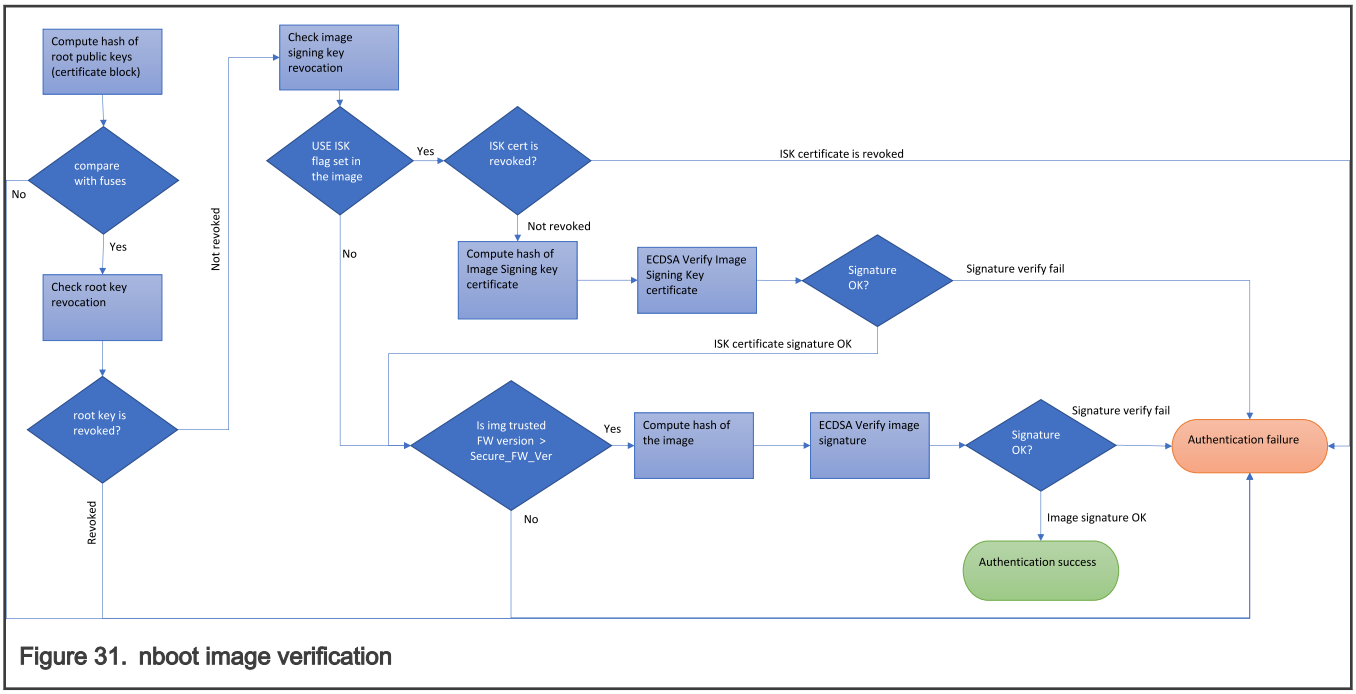


Figure 31. nboot image verification

9.3.3.2 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

9.3.4 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.

9.3.4.1 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.

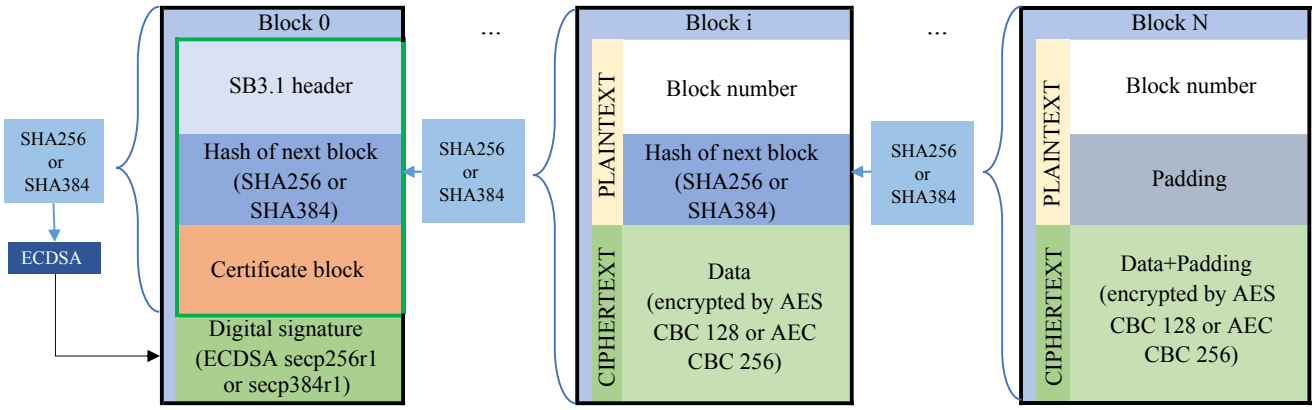


Figure 32. SB3.1 structure

9.3.4.2 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 34) 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 33 and Figure 34)) by use of SHA256 if secp256r1 is selected for ECDSA digital signature, or SHA384 if secp384r1 is selected for ECDSA digital signature.

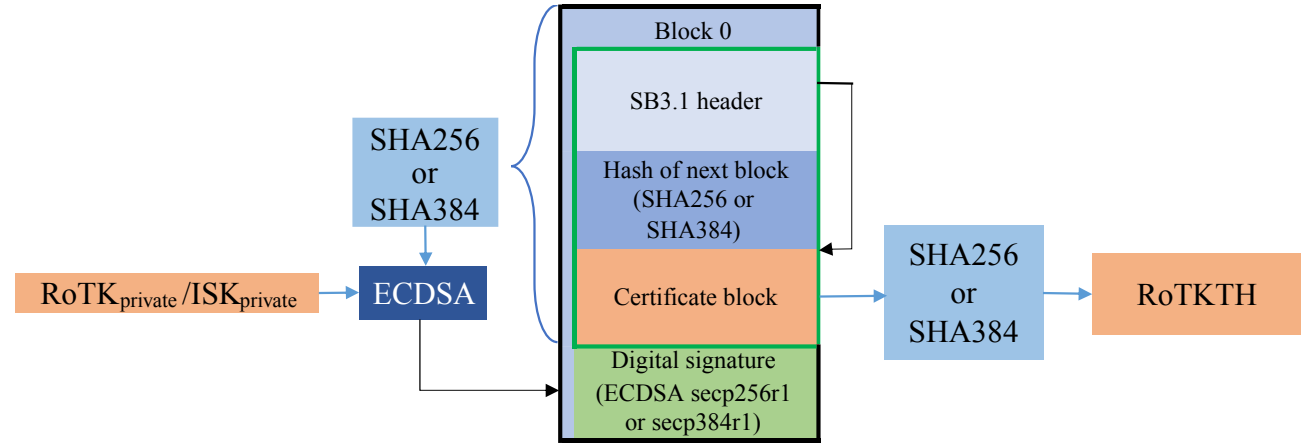


Figure 33. SB3.1 Block 0

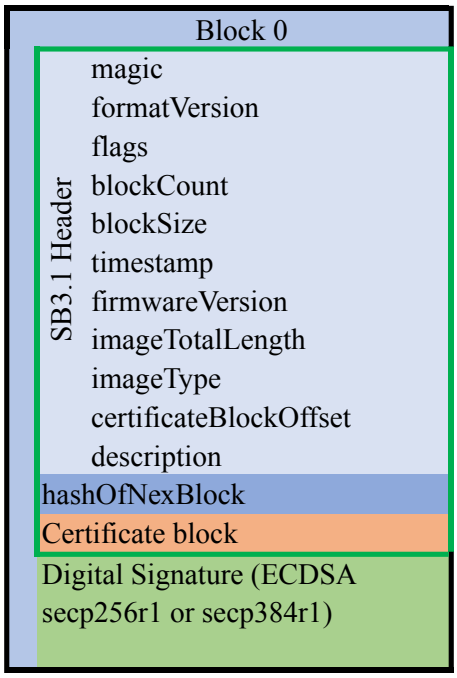


Figure 34. SB3.1 Block 0 full structure

- **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.

9.3.4.3 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.

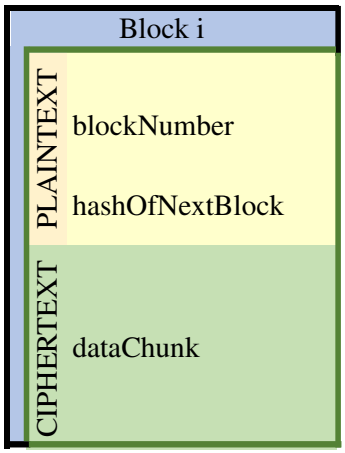


Figure 35. Block i

- **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.

9.3.4.4 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 98. 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 99. 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: "0E000000000000000000000000".
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 99. 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: "8b71ad27000000000000000000000000000000000c00100210000010000000001"
   - Data2: "8b71ad27000000000000000000000000000000000c00100210000010000000002"
   - Res1 = PRF(PCK, Data1) = "68fd9ef140290488eca5736aa9f4b4a5"
   - Res2 = PRF(PCK, Data2) = "cf437c8618809047ec1d46f70523481a"
2. FW_KBLK(3) 256bit derivation
   - FW_KDK: "68fd9ef140290488eca5736aa9f4b4a5cf437c8618809047ec1d46f70523481a"
   - Data1: "030000000000000000000000000000000000000000c01000210000010000000001"
   - Data2: "030000000000000000000000000000000000000000c01000210000010000000002"
   - Res1 = PRF(FW_KDK, Data1) = "4b2afc98b4ca03fc0de090be76d3beb2"
   - Res2 = PRF(FW_KDK, Data2) = "729fb4b3149b3ea05f414a2dd0a193ce"
   - FW_KBLK(3): "4b2afc98b4ca03fc0de090be76d3beb2729fb4b3149b3ea05f414a2dd0a193ce"

9.3.4.5 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.

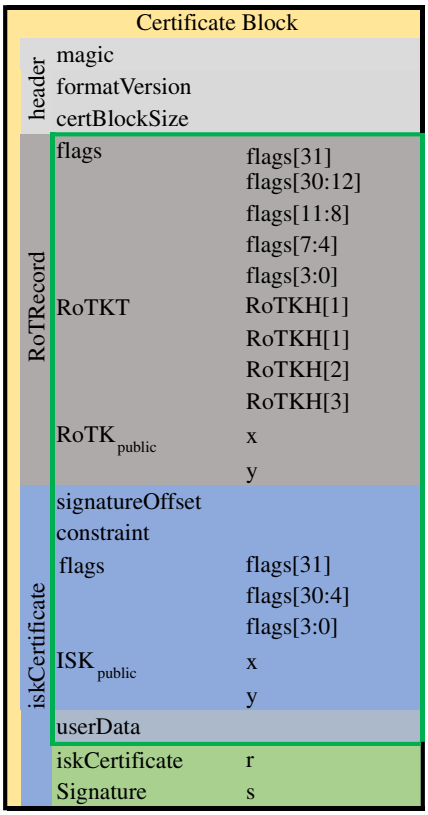


Figure 36. Certificate Block

- **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.
    - f**lags[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.

9.3.4.6 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.

9.3.4.7 SB3.1 processing

The following figure shows SB3.1 processing flow.

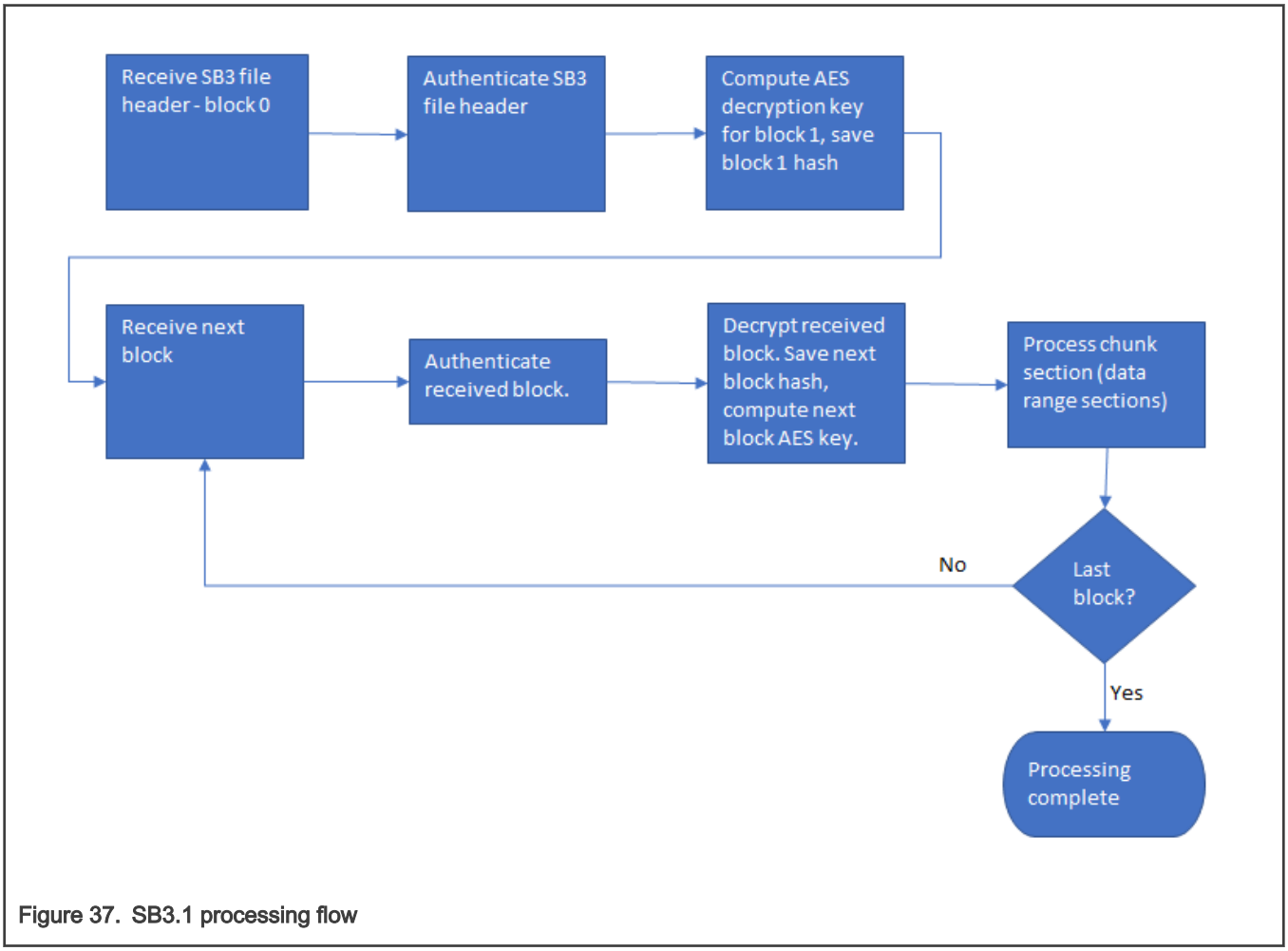


Figure 37. SB3.1 processing flow

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

9.3.5 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.

9.3.5.1 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 100. 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

9.3.6 ROM Trustzone preset data support

9.3.6.1 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.

9.3.6.1.1 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.

9.3.6.1.2 Trustzone preset data structure

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

```
typedef struct _tzm_secure_config
{
    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_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*/
    /* 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; /*!< MBC 0, Global Access Control Register 0*/
    uint32_t mbc0_memn_glbac1; /*!< MBC 0, Global Access Control Register 1*/
    uint32_t mbc0_memn_glbac2; /*!< MBC 0, Global Access Control Register 2*/
    uint32_t mbc0_memn_glbac3; /*!< MBC 0, Global Access Control Register 3*/
    uint32_t mbc0_memn_glbac4; /*!< MBC 0, Global Access Control Register 4*/
    uint32_t mbc0_memn_glbac5; /*!< MBC 0, Global Access Control Register 5*/
    uint32_t mbc0_memn_glbac6; /*!< MBC 0, Global Access Control Register 6*/
    uint32_t mbc0_memn_glbac7; /*!< MBC 0, Global Access Control Register 7*/
    /* SHARED BY SECTION 4, 5, 6 - START */
```

```
/* SECTION 4 - START */
uint32_t mbc0_0_mem0_blk_cfg_w0; /*!< MBC 0, Domain 0, Slave 0, Configuration Word Register 0*/
uint32_t mbc0_0_mem0_blk_cfg_w1; /*!< MBC 0, Domain 0, Slave 0, Configuration Word Register 1*/
uint32_t mbc0_0_mem0_blk_cfg_w2; /*!< MBC 0, Domain 0, Slave 0, Configuration Word Register 2*/
uint32_t mbc0_0_mem0_blk_cfg_w3; /*!< MBC 0, Domain 0, Slave 0, Configuration Word Register 3*/
uint32_t mbc0_0_mem1_blk_cfg_w0; /*!< MBC 0, Domain 0, Slave 1, Configuration Word Register 0*/
uint32_t mbc0_0_mem2_blk_cfg_w0; /*!< MBC 0, Domain 0, Slave 2, Configuration Word Register 0*/
uint32_t mbc0_0_mem3_blk_cfg_w0; /*!< MBC 0, Domain 0, Slave 3, Configuration Word Register 0*/
uint32_t mbc0_0_mem3_blk_cfg_w1; /*!< MBC 0, Domain 0, Slave 3, Configuration Word Register 1*/
/* SECTION 4 - END */

/* SECTION 5 - START */
uint32_t mbc0_1_mem0_blk_cfg_w0; /*!< MBC 0, Domain 1, Slave 0, Configuration Word Register 0*/
uint32_t mbc0_1_mem0_blk_cfg_w1; /*!< MBC 0, Domain 1, Slave 0, Configuration Word Register 1*/
uint32_t mbc0_1_mem0_blk_cfg_w2; /*!< MBC 0, Domain 1, Slave 0, Configuration Word Register 2*/
uint32_t mbc0_1_mem0_blk_cfg_w3; /*!< MBC 0, Domain 1, Slave 0, Configuration Word Register 3*/
uint32_t mbc0_1_mem1_blk_cfg_w0; /*!< MBC 0, Domain 1, Slave 1, Configuration Word Register 0*/
uint32_t mbc0_1_mem2_blk_cfg_w0; /*!< MBC 0, Domain 1, Slave 2, Configuration Word Register 0*/
uint32_t mbc0_1_mem3_blk_cfg_w0; /*!< MBC 0, Domain 1, Slave 3, Configuration Word Register 0*/
uint32_t mbc0_1_mem3_blk_cfg_w1; /*!< MBC 0, Domain 1, Slave 3, Configuration Word Register 1*/
/* SECTION 5 - END */

/* SECTION 6 - START */
uint32_t mbc0_2_mem0_blk_cfg_w0; /*!< MBC 0, Domain 2, Slave 0, Configuration Word Register 0*/
uint32_t mbc0_2_mem0_blk_cfg_w1; /*!< MBC 0, Domain 2, Slave 0, Configuration Word Register 1*/
uint32_t mbc0_2_mem0_blk_cfg_w2; /*!< MBC 0, Domain 2, Slave 0, Configuration Word Register 2*/
uint32_t mbc0_2_mem0_blk_cfg_w3; /*!< MBC 0, Domain 2, Slave 0, Configuration Word Register 3*/
uint32_t mbc0_2_mem1_blk_cfg_w0; /*!< MBC 0, Domain 2, Slave 1, Configuration Word Register 0*/
uint32_t mbc0_2_mem2_blk_cfg_w0; /*!< MBC 0, Domain 2, Slave 2, Configuration Word Register 0*/
uint32_t mbc0_2_mem3_blk_cfg_w0; /*!< MBC 0, Domain 2, Slave 3, Configuration Word Register 0*/
uint32_t mbc0_2_mem3_blk_cfg_w1; /*!< MBC 0, Domain 2, Slave 3, Configuration Word Register 1*/
/* SECTION 6 - END */

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

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

/* SECTION 8 - START */
uint32_t mbc1_1_mem0_blk_cfg_w0; /*!< MBC 1, Domain 1, Slave 0, Configuration Word Register 0*/
uint32_t mbc1_1_mem1_blk_cfg_w0; /*!< MBC 1, Domain 1, Slave 1, Configuration Word Register 0*/
uint32_t mbc1_1_mem2_blk_cfg_w0; /*!< MBC 1, Domain 1, Slave 2, Configuration Word Register 0*/
uint32_t mbc1_1_mem3_blk_cfg_w0; /*!< MBC 1, Domain 1, Slave 3, Configuration Word Register 0*/
/* SECTION 8 - END */

/* SECTION 9 - START */
```

```
    uint32_t mbc1_2_mem0_blk_cfg_w0; /*!< MBC 1, Domain 2, Slave 0, Configuration Word Register 0*/
    uint32_t mbc1_2_mem1_blk_cfg_w0; /*!< MBC 1, Domain 2, Slave 1, Configuration Word Register 0*/
    uint32_t mbc1_2_mem2_blk_cfg_w0; /*!< MBC 1, Domain 2, Slave 2, Configuration Word Register 0*/
    uint32_t mbc1_2_mem3_blk_cfg_w0; /*!< MBC 1, Domain 2, Slave 3, Configuration Word Register 0*/
    /* SECTION 9 - END */

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

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

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

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

```
    uint32_t mbc2_2_mem1_blk_cfg_w0; /*!< MBC 2, Domain 2, Slave 1, Configuration Word Register 0*/
    uint32_t mbc2_2_mem2_blk_cfg_w0; /*!< MBC 2, Domain 2, Slave 2, Configuration Word Register 0*/
    uint32_t mbc2_2_mem2_blk_cfg_w1; /*!< MBC 2, Domain 2, Slave 2, Configuration Word Register 1*/
    /* SECTION 12 - END */

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

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

    /* SECTION 14 - START */
    uint32_t mrc0_1_rgd0_w0; /*!< MRC 0, Domain 1, Region 0, Memory Region Descriptor Register 0*/
    uint32_t mrc0_1_rgd0_w1; /*!< MRC 0, Domain 1, Region 0, Memory Region Descriptor Register 1*/
    uint32_t mrc0_1_rgd1_w0; /*!< MRC 0, Domain 1, Region 1, Memory Region Descriptor Register 0*/
    uint32_t mrc0_1_rgd1_w1; /*!< MRC 0, Domain 1, Region 1, Memory Region Descriptor Register 1*/
    uint32_t mrc0_1_rgd2_w0; /*!< MRC 0, Domain 1, Region 2, Memory Region Descriptor Register 0*/
    uint32_t mrc0_1_rgd2_w1; /*!< MRC 0, Domain 1, Region 2, Memory Region Descriptor Register 1*/
    uint32_t mrc0_1_rgd3_w0; /*!< MRC 0, Domain 1, Region 3, Memory Region Descriptor Register 0*/
    uint32_t mrc0_1_rgd3_w1; /*!< MRC 0, Domain 1, Region 3, Memory Region Descriptor Register 1*/
    uint32_t mrc0_1_rgd4_w0; /*!< MRC 0, Domain 1, Region 4, Memory Region Descriptor Register 0*/
    uint32_t mrc0_1_rgd4_w1; /*!< MRC 0, Domain 1, Region 4, Memory Region Descriptor Register 1*/
    uint32_t mrc0_1_rgd5_w0; /*!< MRC 0, Domain 1, Region 5, Memory Region Descriptor Register 0*/
    uint32_t mrc0_1_rgd5_w1; /*!< MRC 0, Domain 1, Region 5, Memory Region Descriptor Register 1*/
    uint32_t mrc0_1_rgd6_w0; /*!< MRC 0, Domain 1, Region 6, Memory Region Descriptor Register 0*/
    uint32_t mrc0_1_rgd6_w1; /*!< MRC 0, Domain 1, Region 6, Memory Region Descriptor Register 1*/
    uint32_t mrc0_1_rgd7_w0; /*!< MRC 0, Domain 1, Region 7, Memory Region Descriptor Register 0*/
    uint32_t mrc0_1_rgd7_w1; /*!< MRC 0, Domain 1, Region 7, Memory Region Descriptor Register 1*/
    /* SECTION 14 - END */

    /* SECTION 15 - START */
    uint32_t mrc0_2_rgd0_w0; /*!< MRC 0, Domain 2, Region 0, Memory Region Descriptor Register 0*/
    uint32_t mrc0_2_rgd0_w1; /*!< MRC 0, Domain 2, Region 0, Memory Region Descriptor Register 1*/
    uint32_t mrc0_2_rgd1_w0; /*!< MRC 0, Domain 2, Region 1, Memory Region Descriptor Register 0*/
    uint32_t mrc0_2_rgd1_w1; /*!< MRC 0, Domain 2, Region 1, Memory Region Descriptor Register 1*/
    uint32_t mrc0_2_rgd2_w0; /*!< MRC 0, Domain 2, Region 2, Memory Region Descriptor Register 0*/
```

```
    uint32_t mrc0_2_rgd2_w1; /*!< MRC 0, Domain 2, Region 2, Memory Region Descriptor Register 1*/
    uint32_t mrc0_2_rgd3_w0; /*!< MRC 0, Domain 2, Region 3, Memory Region Descriptor Register 0*/
    uint32_t mrc0_2_rgd3_w1; /*!< MRC 0, Domain 2, Region 3, Memory Region Descriptor Register 1*/
    uint32_t mrc0_2_rgd4_w0; /*!< MRC 0, Domain 2, Region 4, Memory Region Descriptor Register 0*/
    uint32_t mrc0_2_rgd4_w1; /*!< MRC 0, Domain 2, Region 4, Memory Region Descriptor Register 1*/
    uint32_t mrc0_2_rgd5_w0; /*!< MRC 0, Domain 2, Region 5, Memory Region Descriptor Register 0*/
    uint32_t mrc0_2_rgd5_w1; /*!< MRC 0, Domain 2, Region 5, Memory Region Descriptor Register 1*/
    uint32_t mrc0_2_rgd6_w0; /*!< MRC 0, Domain 2, Region 6, Memory Region Descriptor Register 0*/
    uint32_t mrc0_2_rgd6_w1; /*!< MRC 0, Domain 2, Region 6, Memory Region Descriptor Register 1*/
    uint32_t mrc0_2_rgd7_w0; /*!< MRC 0, Domain 2, Region 7, Memory Region Descriptor Register 0*/
    uint32_t mrc0_2_rgd7_w1; /*!< MRC 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_icr4;  /*!< GPIO A, Interrupt Control Pin 4 */
    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 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 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 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 */
    /* SECTION 16 - START */

} 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.

9.3.6.2 TZ-M data with specific function

9.3.6.2.1 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.

9.3.6.2.2 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. Please 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 101. 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)

Table continues on the next page...

Table 101. tzm_control variable definition (continued)

Bit	Description
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.

9.3.6.2.3 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 102. cm33_misc_ctrl variable definition

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

9.3.7 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.

9.3.7.1 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 $RoTK_{private}$ 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.

9.3.7.2 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]
- $RoTK_{public}$ does not match with RoTKTH in fuses.
- KeyBlobEncryptionKey does not match with SB3KDK in fuses

9.3.7.3 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.

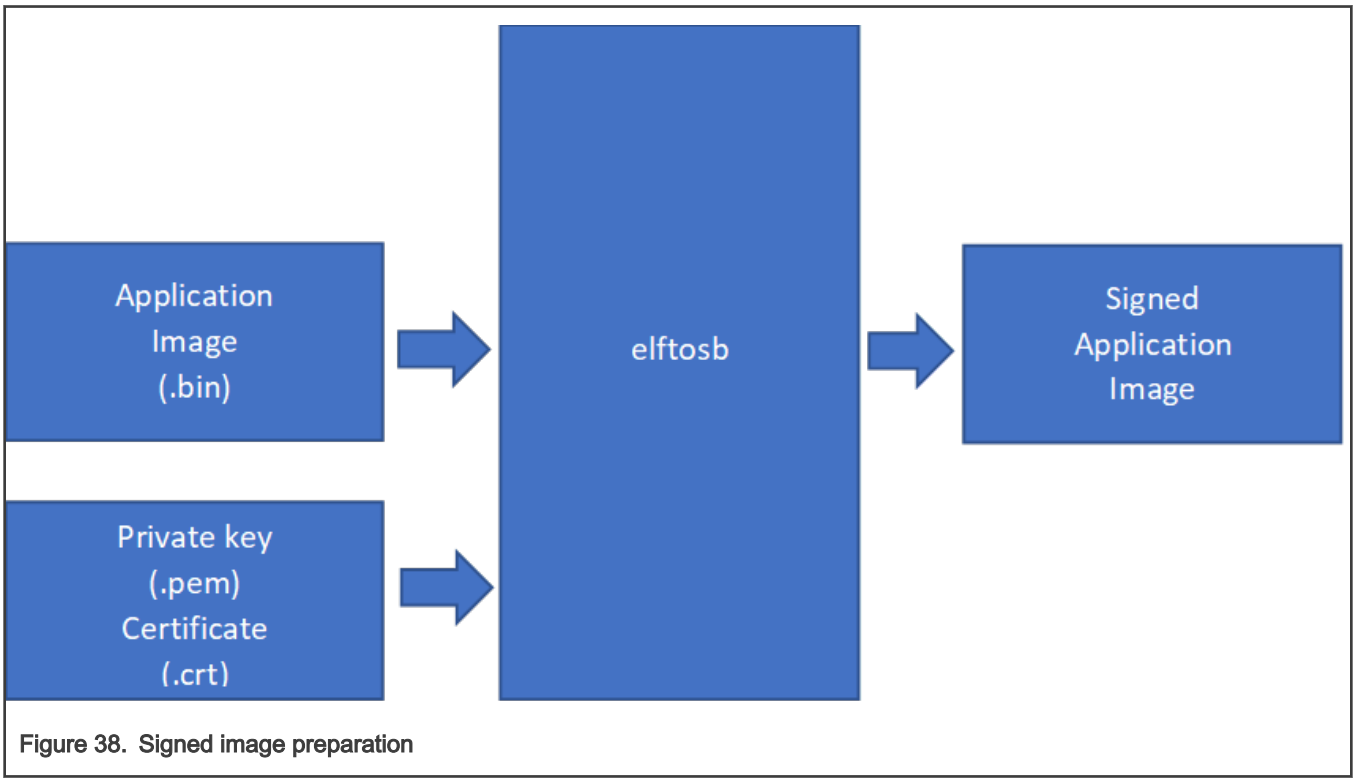


Figure 38. Signed image preparation

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

9.3.7.4 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

9.3.7.5 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 103. Secure boot status code

Status code	Description
(CMC0_BSR [0]) = 1b	NBU FW CMAC authentication was success.

Table continues on the next page...

Table 103. Secure boot status code (continued)

Status code	Description
(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 10
ROM ISP

10.1 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 and it will auto-detect activity on the LPI2C/LPSPI/LPUART or CAN interface. The auto-detect looks for activity on the LPUART, LPI2C, LPSPI and 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.

One method to make ROM bootloader go to ISP path is by pressing 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.

10.2 Available peripherals

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

Table 104. Peripheral instances and pin assignments used by ISP

Peripheral instances	Pin name	Pin assignment	Alt
LPUART1	LPUART1_TX	PTC3	3
	LPUART1_RX	PTC2	3
LPSPI1	LPSPI1_PCS0	PTB0	2
	LPSPI1_SIN	PTB1	2
	LPSPI1_SCK	PTB2	2
	LPSPI1_SOUT	PTB3	2
CAN0	CAN0_TX	PTC4	3
	CAN0_RX	PTC5	3
LPI2C1	LPI2C1_SCL	PTB5	4
	LPI2C1_SDA	PTB4	4
Boot pin	BOOT_CONFIG	PTA4	Default

10.3 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 105. Available ISP commands for different lifecycle

Command	Description	At OEM Open	After OEM Open
Reset	Reset the device	Available	Available

Table continues on the next page...

Table 105. Available ISP commands for different lifecycle (continued)

Command	Description	At OEM Open	After OEM Open
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	CM33 only	NA
flash-erase-all [<memoryID>]	Erase the entire flash memory specified by memoryID	CM33 only	NA
read-memory <addr> <byte_count> [<file>]	Read memory at specified address	CM33 only	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	CM33 only	NA
fill-memory <addr> <byte_count> <pattern> [word | short | byte]	Fill memory with pattern; pattern size can be word(default), short or byte	CM33 only	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

10.3.1 Available properties

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

Table 106. 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.
CRCCheckStatus	no	8	4	The status of the application CRC check.

Table continues on the next page...

Table 106. Supported properties in GetProperty and SetProperty (continued)

Name	Writable	Tag value	Size in Bytes	Description
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
BootStatus	no	20	4	Value of Boot Status Register
LoadableFWVersion	no	21	4	ELE loadable firmware version
FuseProgramVoltage	yes	22	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)
TargetVersion	no	24	4	Target version

10.3.2 Fuse reading and programming

A set of fuses can be read / programmed using fuse-program and fuse-read commands. The index of fuses can be found in Lifecycle and fuses.

Please note that -- set-property 22 1 needs to be used before -- fuse-program command to ensure fuse can be programmed successfully. And -- set-property 22 0 needs to be done after all the fuse programming completes.

10.3.3 Commands with limitation

At OEM Open lifecycle, flash-erase-region/ flash-erase-all/ read-memory/ write-memory/ fill-memory commands can be used to Program Flash (CM33) with no restriction. But for radio flash, the accessibility is controlled by Token. If Token is owned by OEM1,

these commands can be used to read/write/erase radio flash. Otherwise, these commands return error when trying to access radio flash.

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 128 for details about memory ID.

10.3.4 receive-sb-file

receive-sb-file command is used to do image update on CM33 flash or radio (CM3) 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. Please note that -- set-property 22 1 needs to be used before receive-sb-file command if the sb file contains "programFuses" command. And -- set-property 22 0 needs to be done after receive-sb-file is done.

10.4 In-System Programming 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.

10.4.1 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)

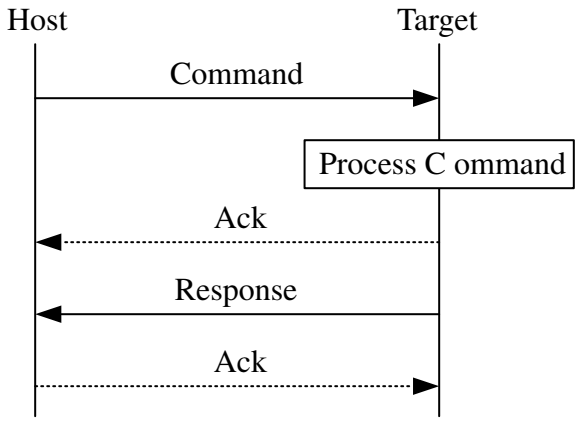


Figure 39. Command with no data phase

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.

10.4.2 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.

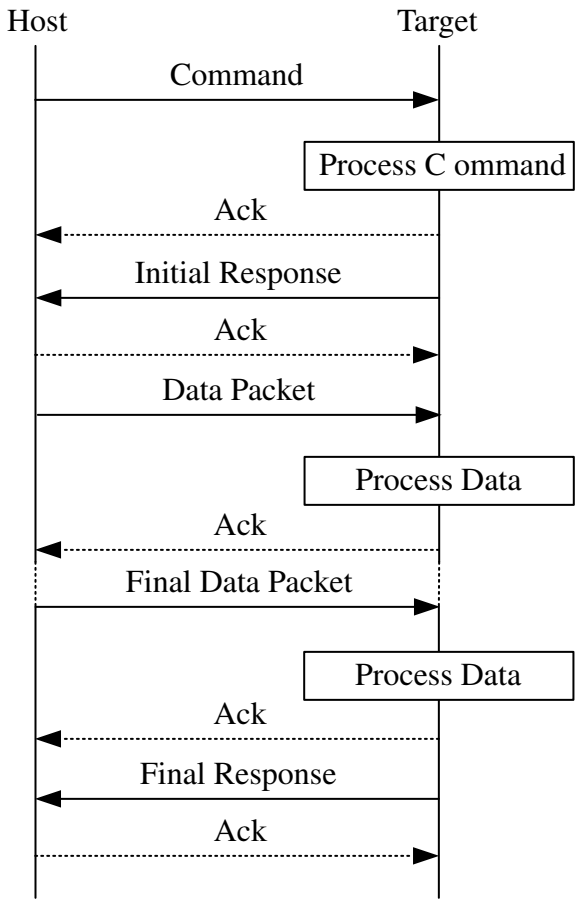


Figure 40. Command with incoming data phase

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.

10.4.3 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).

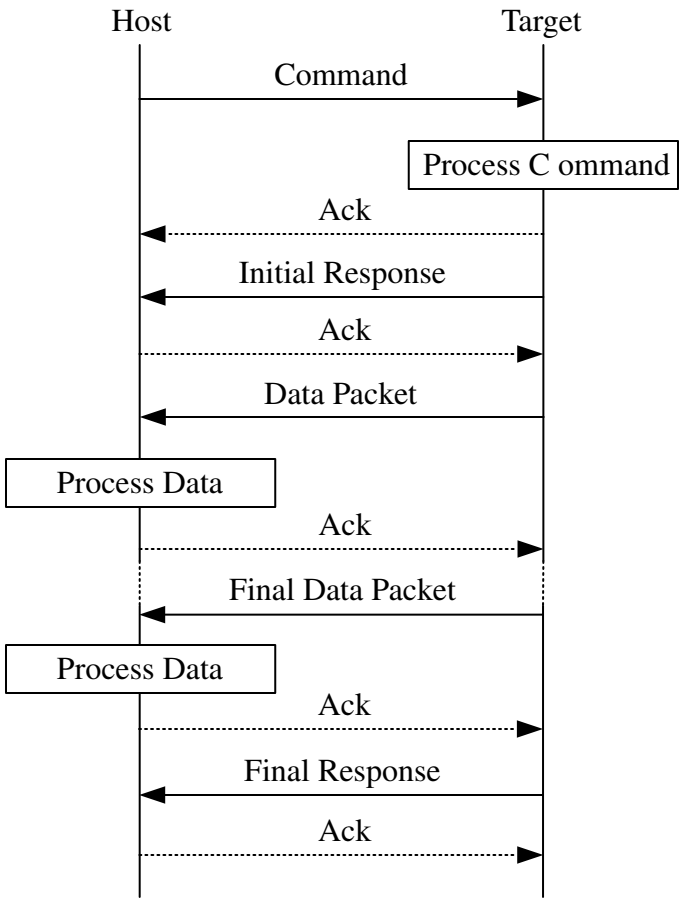


Figure 41. Command with outgoing data phase

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.

10.5 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.

10.5.1 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 107. Ping packet format

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

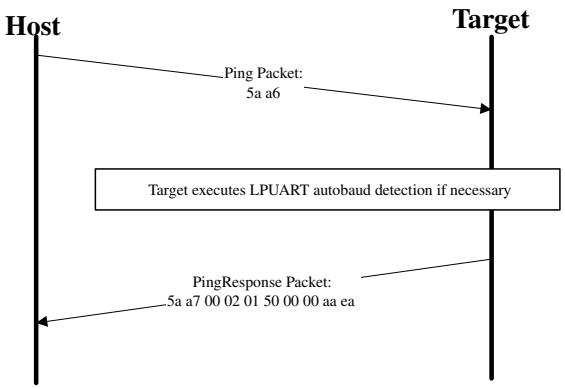


Figure 42. Ping packet protocol sequence

10.5.2 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 108. ping response packet format

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

Table continues on the next page...

Table 108. 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.

10.5.3 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 109. 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 110. 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 111. 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.

10.5.4 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 112. 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;
}
```

10.5.5 Command packet

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

Table 113. 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 114. 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 115. Command tags

Command Tag	Name	Remark
0x01	FlashEraseAll	The command tag specifies one of the commands supported by the bootloader. The valid command tags for the bootloader are listed here.
0x02	FlashEraseRegion	
0x03	ReadMemory	
0x04	WriteMemory	
0x05	FillMemory	
0x06	Reserved	
0x07	GetProperty	
0x08	ReceiveSbFile	
0x09	Execute	
0x0A	Reserved	
0x0B	Reset	
0x0C	SetProperty	
0x0D	Reserved	
0x0E	Reserved	
0x0F	Reserved	
0x10	Reserved	
0x11	Reserved	
0x12	Reserved	
0x13	Reserved	
0x14	FuseProgram	
0x15	Reserved	
0x16	Reserved	
0x17	FuseRead	

Table 116. 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.
0xA3	ReadMemoryResponse	
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)	
0xB5	KeyProvisionResponse	

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.

10.5.6 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
- KeyProvisionResponse

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 117. 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 118. 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 119. 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 120. FlashReadOnceResponse parameters

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

The KeyProvisionResponse packet is sent by the target in response to the host sending a KeyProvision command. The KeyProvisionResponse packet contains the framing packet data and command packet data, with the command/response tag set to a KeyProvisionResponse tag value (0xB5), and the flags field set to kCommandFlag_HasDataPhase (1).

Table 121. KeyProvisionResponse parameters

Byte #	Parameter
0 – 3	Status Code
4 – 7	Data byte count

10.6 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.

10.6.1 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 122. 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)

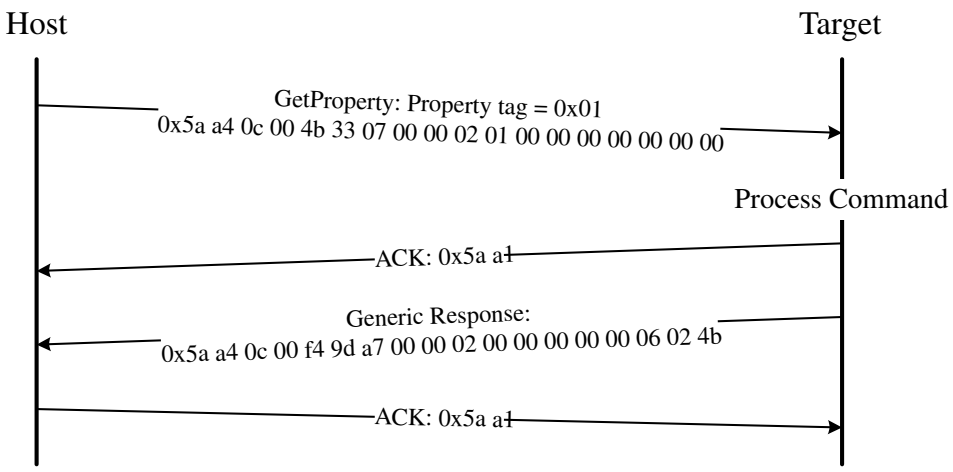


Figure 43. Protocol Sequence for GetProperty Command

Table 123. 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

Table continues on the next page...

Table 123. GetProperty Command Packet Format (Example) (continued)

	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 124. 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

10.6.2 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 125. Parameters for SetProperty Command

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

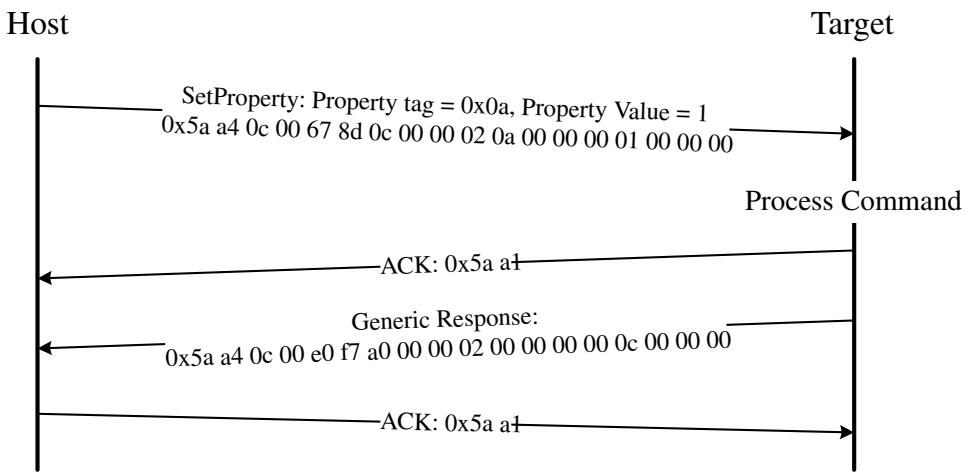


Figure 44. Parameters for SetProperty Command

Table 126. 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 127. SetProperty Response Status Codes

Status Code
kStatus_Success
kStatus_ReadOnly
kStatus_UnknownProperty
kStatus_InvalidArgument

10.6.3 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 128. Parameter for FlashEraseAll Command

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

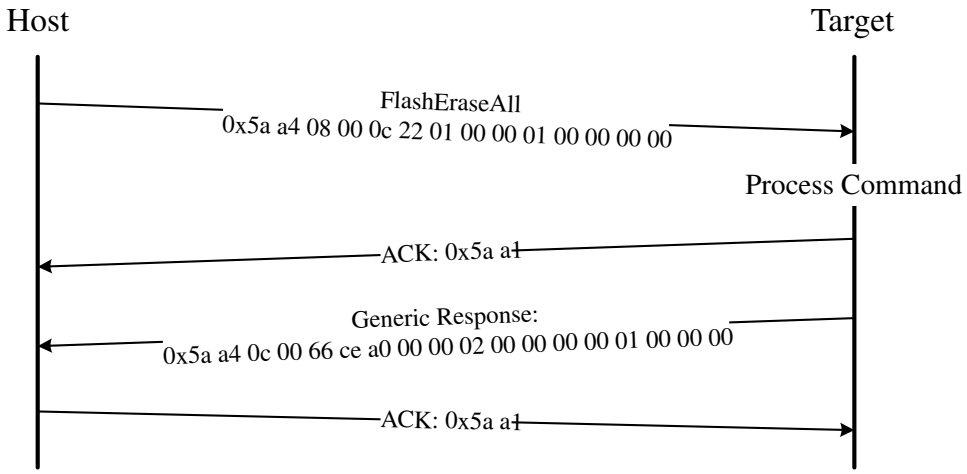


Figure 45. Protocol Sequence for FlashEraseAll Command

Table 129. 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.

10.6.4 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 130. 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.

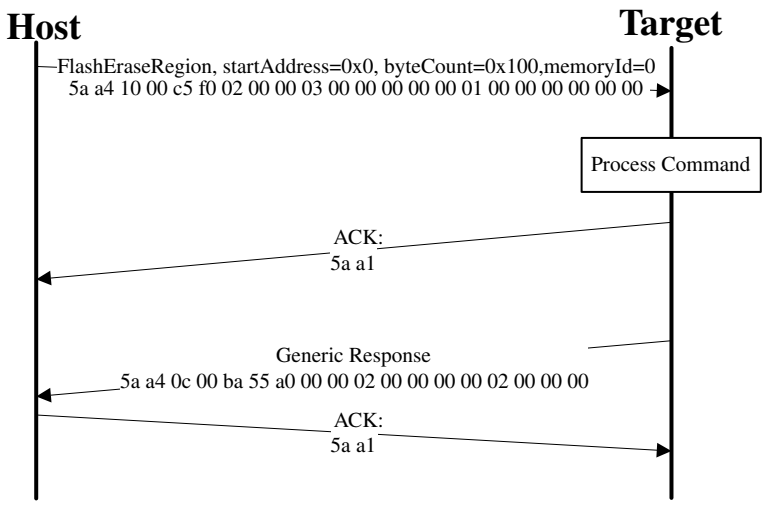


Figure 46. Protocol Sequence for FlashEraseRegion Command

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

Table 131. FlashEraseRegion Response Status Codes

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

10.6.5 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 132. 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

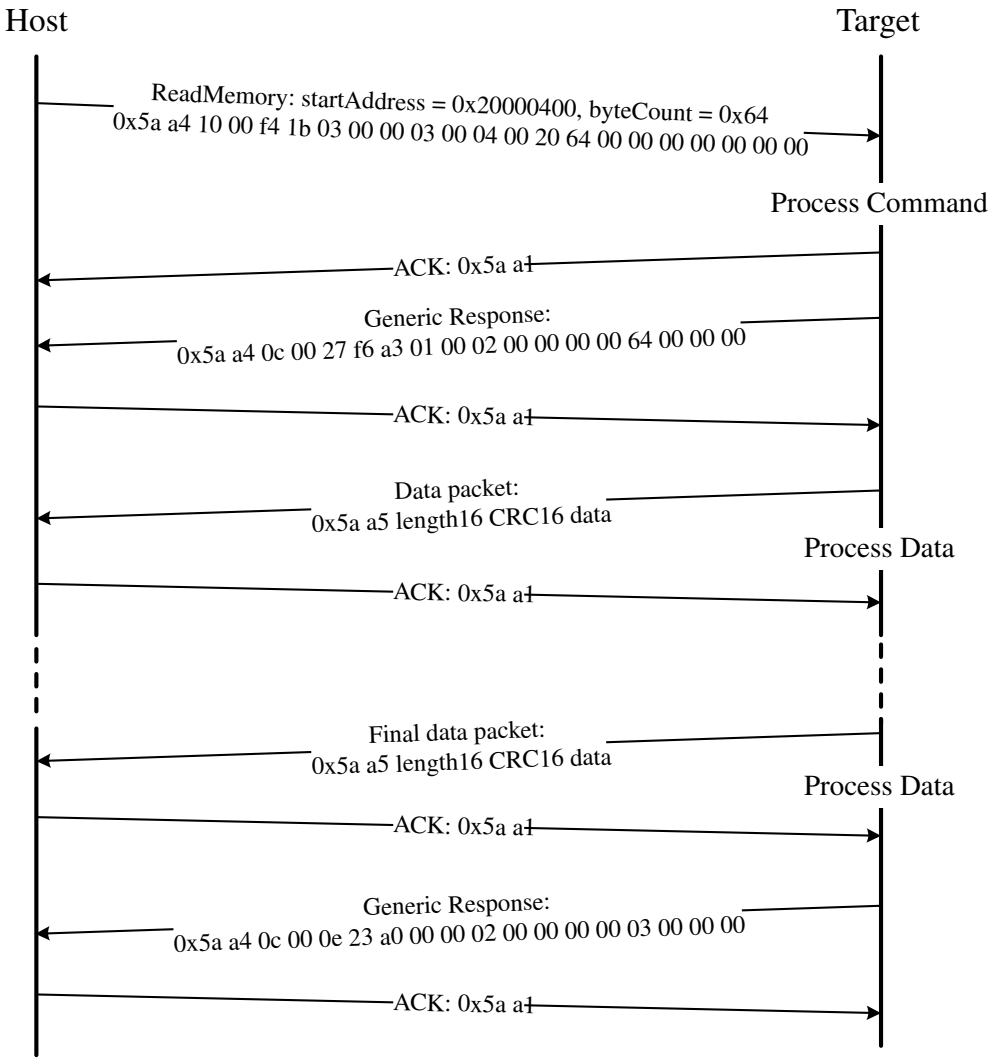


Figure 47. Command sequence for ReadMemory

Table 133. 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 133. 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.

10.6.6 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.
- 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 134. Parameters for WriteMemory Command

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

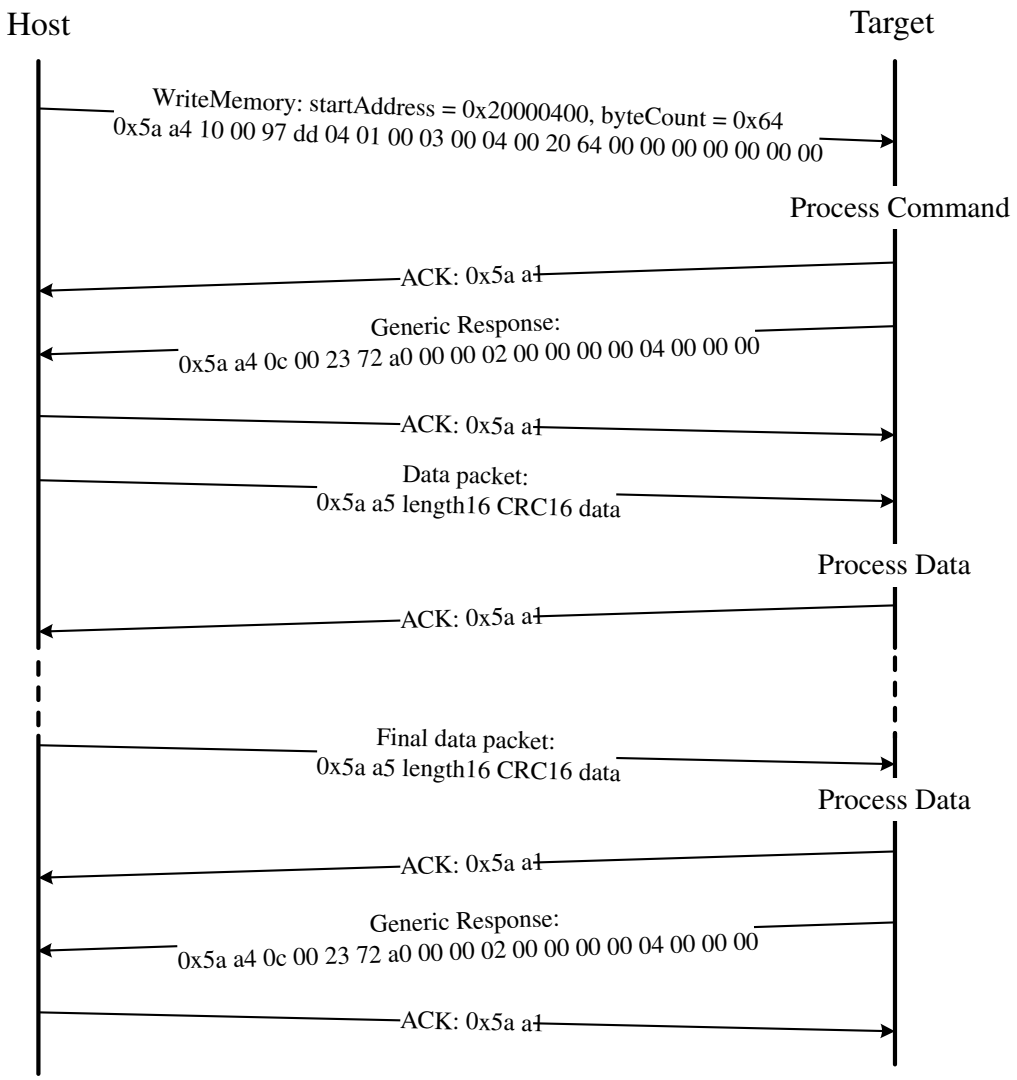


Figure 48. Protocol Sequence for WriteMemory Command

Table 135. 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 135. 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.

10.6.7 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 136. 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.
- 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.

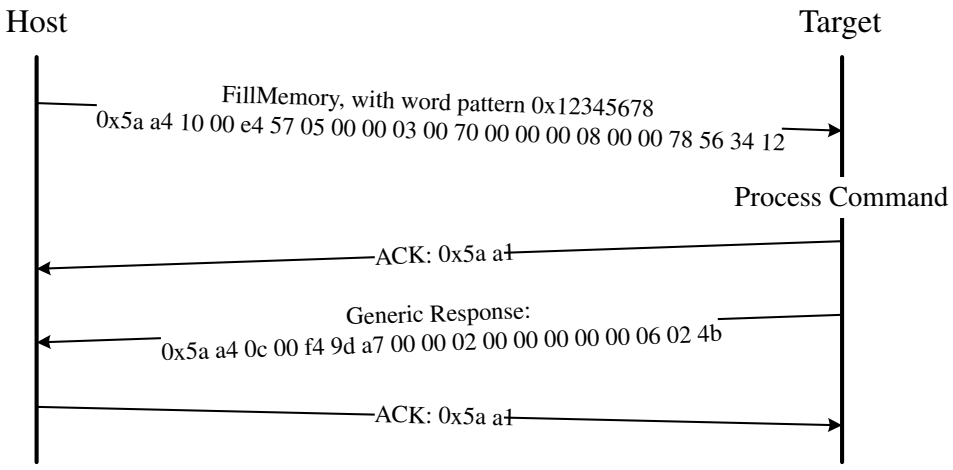


Figure 49. Protocol Sequence for FillMemory Command

Table 137. 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.

10.6.8 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 138. . 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.

10.6.9 Call command

The Call command executes a function that is written in memory at the address sent in the command. The address needs to be a valid memory location residing in accessible flash (internal or external) or in RAM. The command supports the passing of one 32-bit argument. Although the command supports a stack address, at this time the call still takes place using the current stack pointer. After execution of the function, a 32-bit return value is returned in the generic response message.

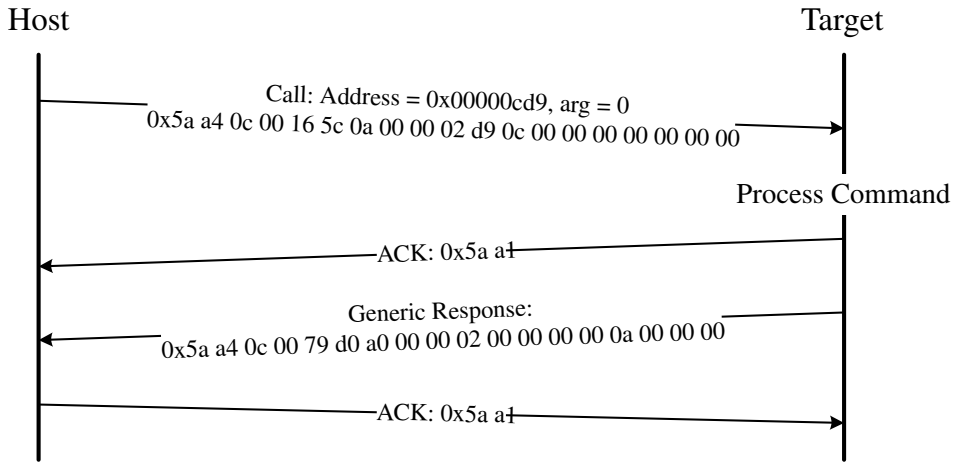


Figure 50. Protocol sequence for Call command

Table 139. Parameters for Call Command

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

Response: The target returns a GenericResponse packet with a status code either set to the return value of the function called or set to kStatus_InvalidArgument (105).

10.6.10 Reset command

The Reset command results in the bootloader resetting the chip.

The Reset command requires no parameters.

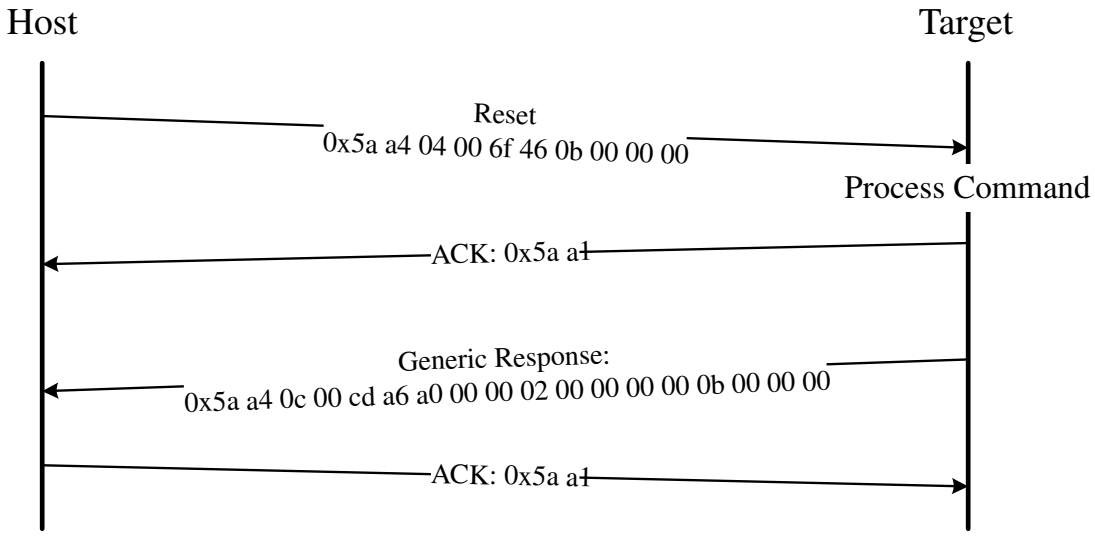


Figure 51. Protocol Sequence for Reset Command

Table 140. Reset Command Packet Format (Example)

Reset	Parameter	Value
Framing packet	start byte	0x5A
	packetType	0xA4, kFramingPacketType_Command
	length	0x04 0x00

Table continues on the next page...

Table 140. Reset Command Packet Format (Example) (continued)

	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.

10.6.11 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 141. . 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.

10.7 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

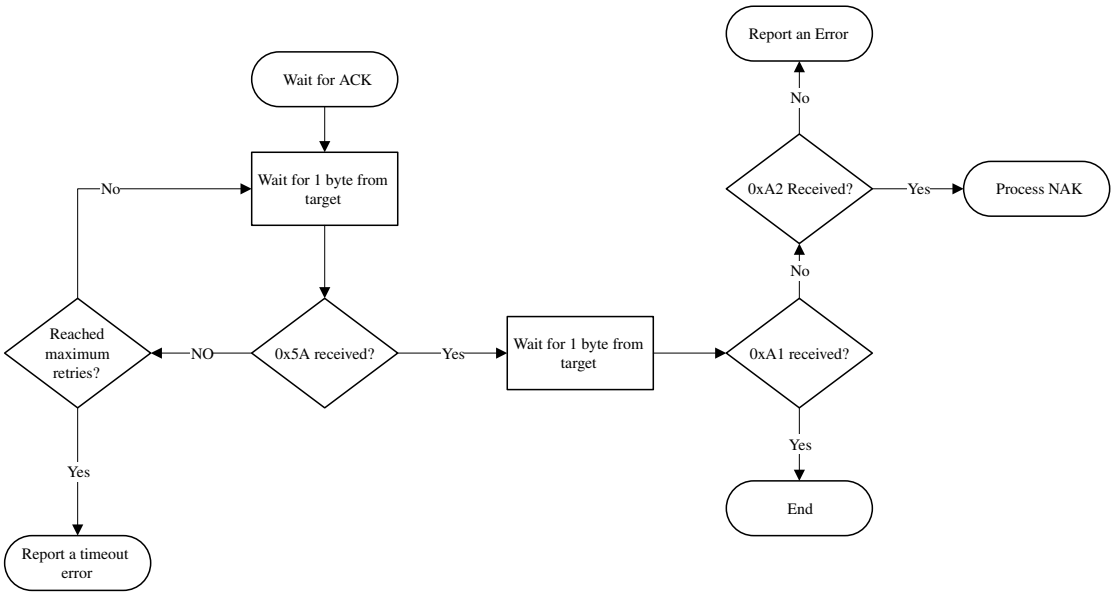


Figure 52. Host reads an ACK from target via LPUART

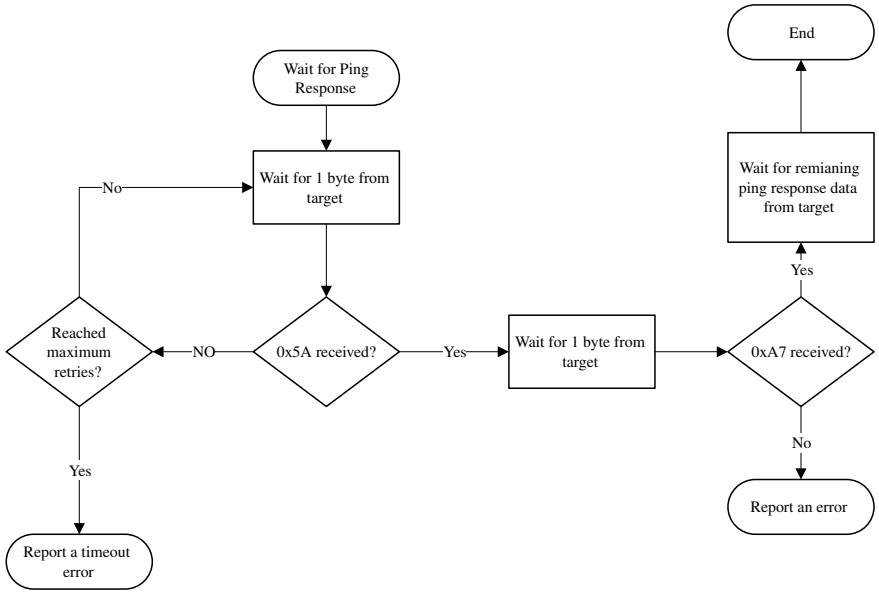


Figure 53. Host reads a ping response from target via LPUART

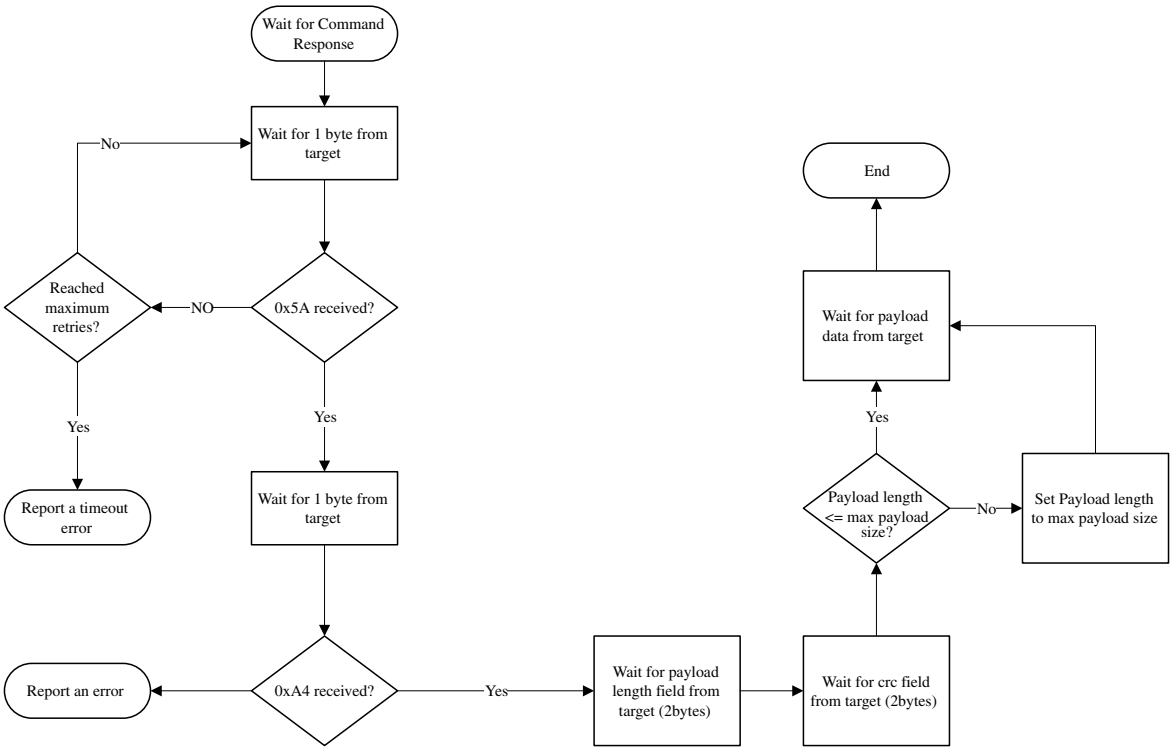


Figure 54. Host reads a command response from target via LPUART

For information on commands and formats, see ISP packet type

10.8 LPI2C ISP

The bootloader supports In-System Programming or serial boot via the LPI2C peripheral, where the LPI2C peripheral serves as the LPI2C slave. The bootloader uses 0x10 as the 7-bit LPI2C slave 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 slave 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 slave address and the direction bit is set to write.
- An outgoing packet is read by the host with a selected LPI2C slave 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.

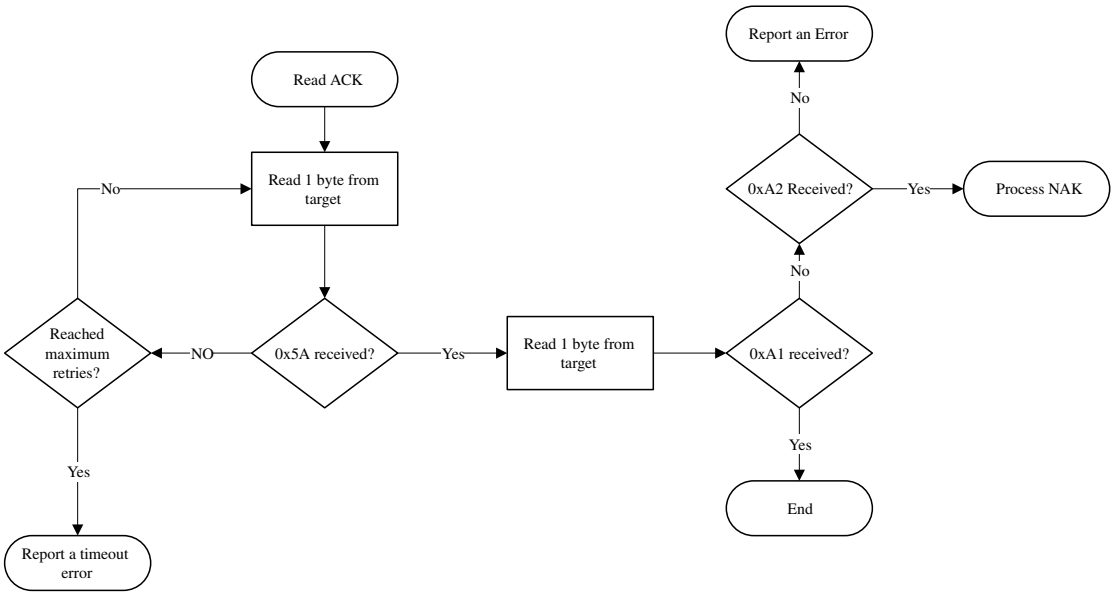


Figure 55. Host reads ACK packet from target via LPI2C

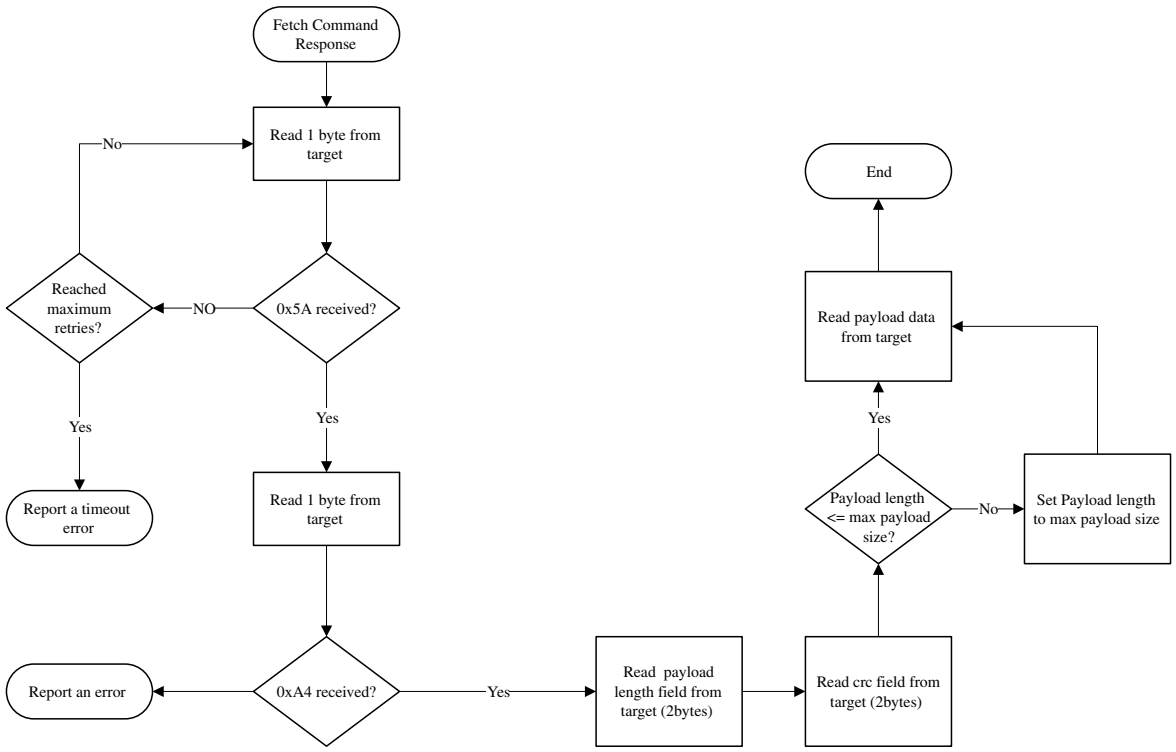


Figure 56. Host reads response from target via LPI2C

For information on commands and formats, see ISP packet type

10.9 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 slave 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.

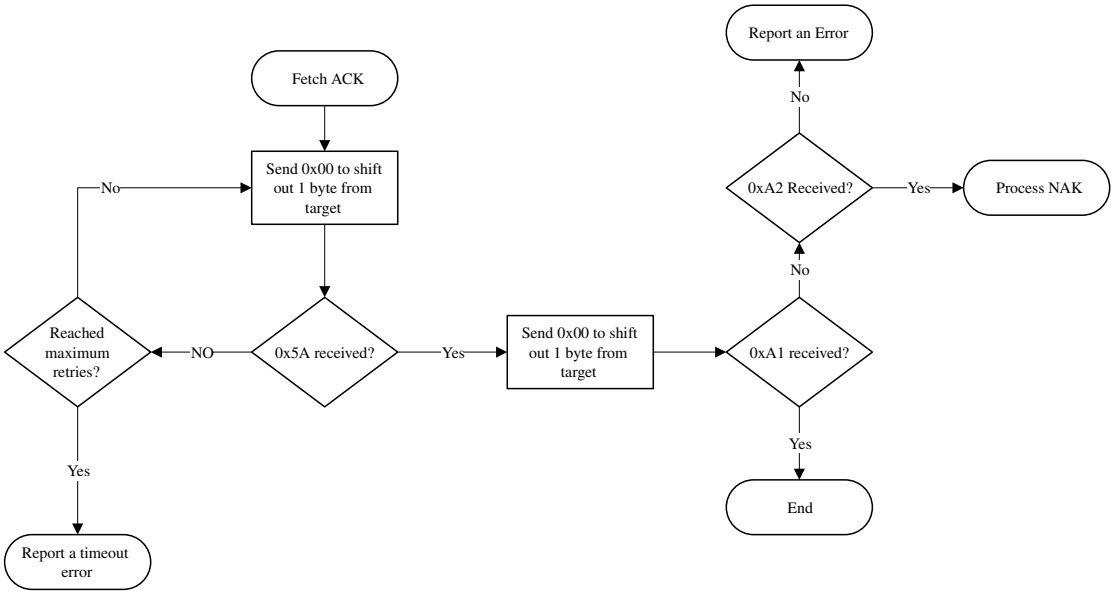


Figure 57. Host reads ACK from target via LPSPI

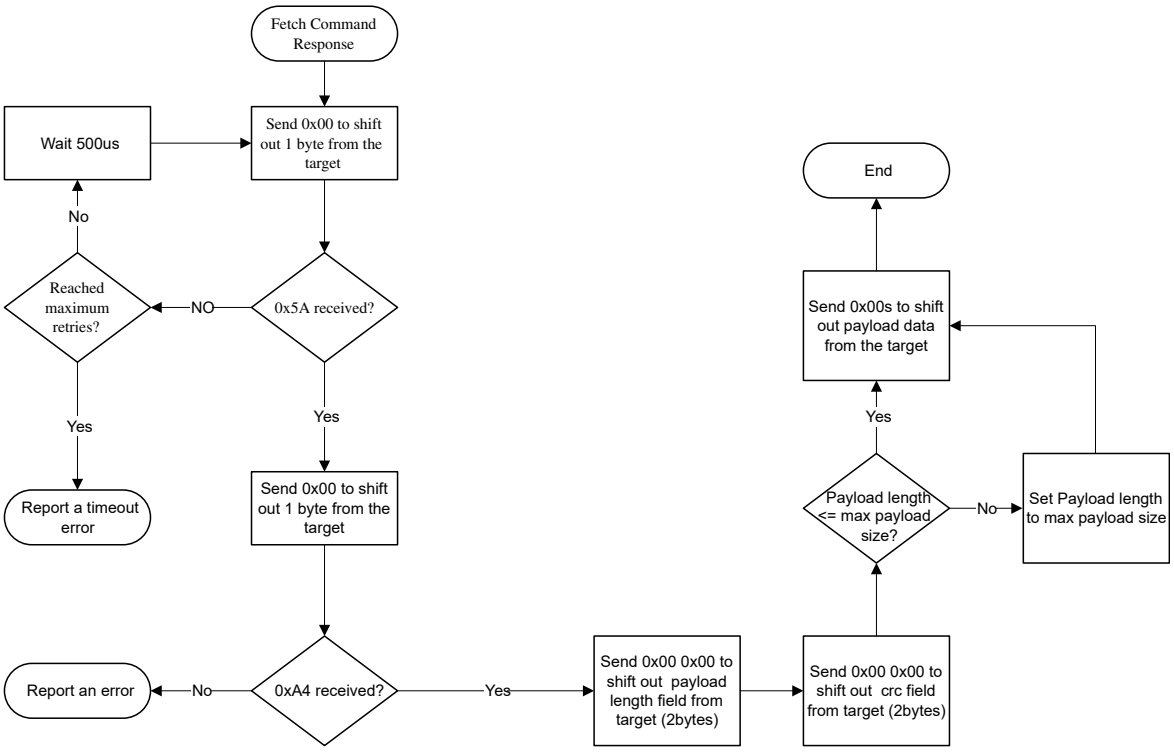


Figure 58. Host reads response from target via LPSPI

For information on commands and formats, see ISP packet type

10.10 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
- 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 FlexCAN 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.

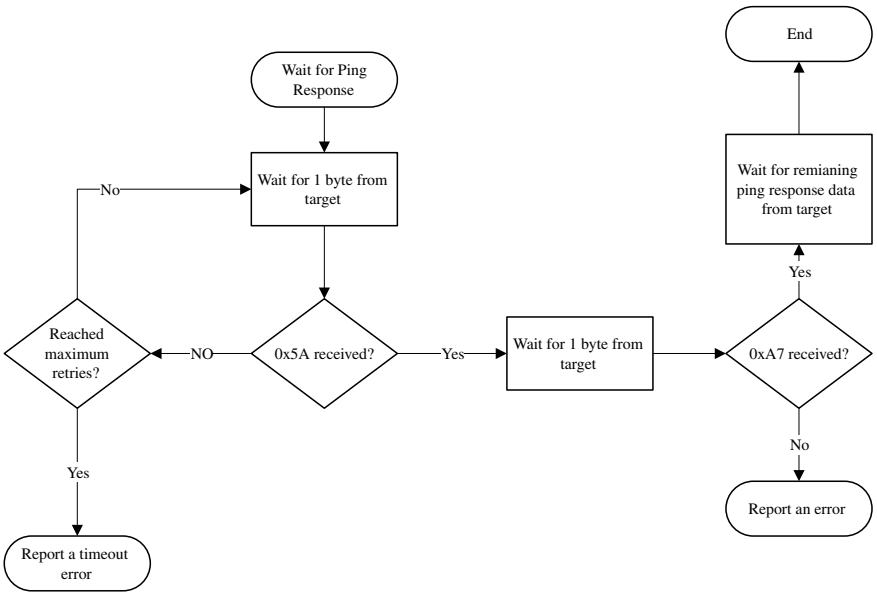


Figure 59. Host reads ping response from target via FlexCAN

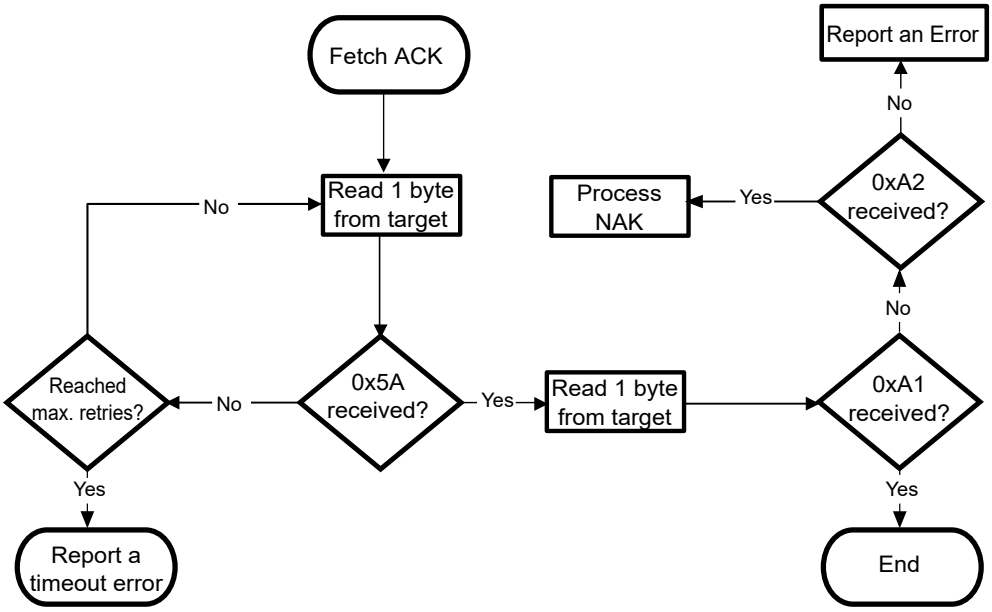


Figure 60. Host reads ACK packet from target via FlexCAN

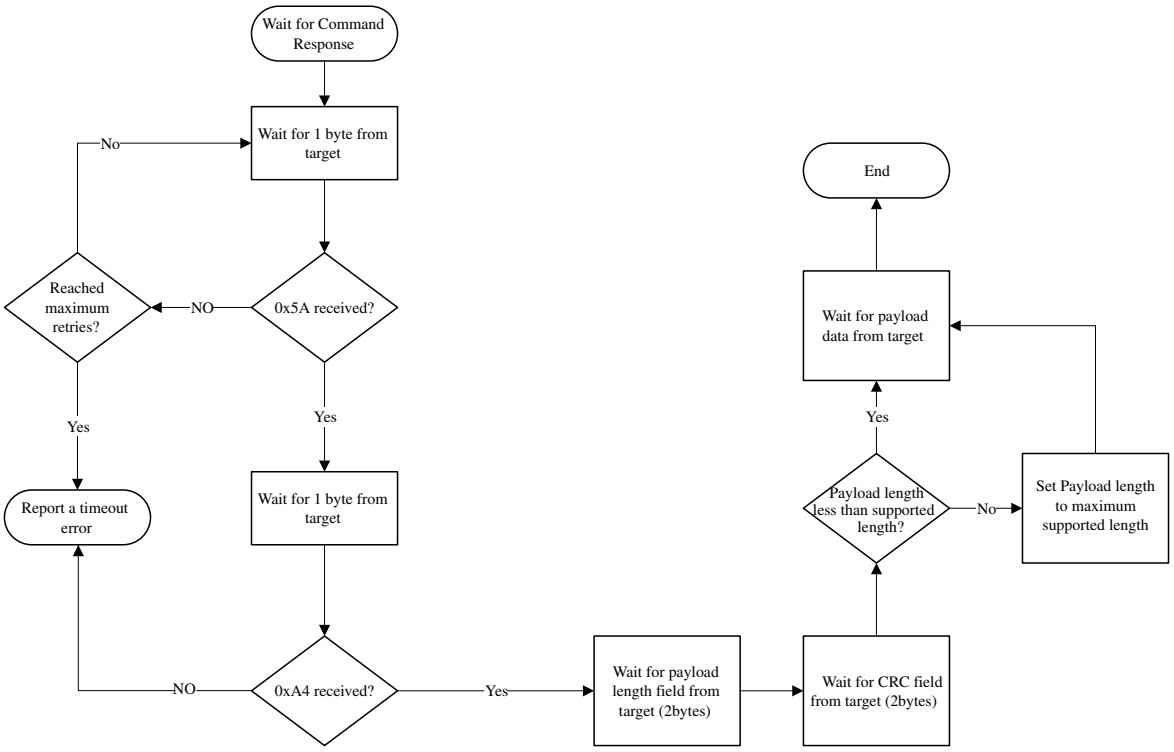


Figure 61. Host reads command response from target via FlexCAN

Chapter 11
ROM API

11.1 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. Please note that 5 KB of SRAM (0x30002000 - 0x30003400) needs to be reserved when calling flash api, SPI NOR API and kb APIs in user application.

The ROM API table locates at address 0x14816fe0. See Figure below for the ROM API layout.

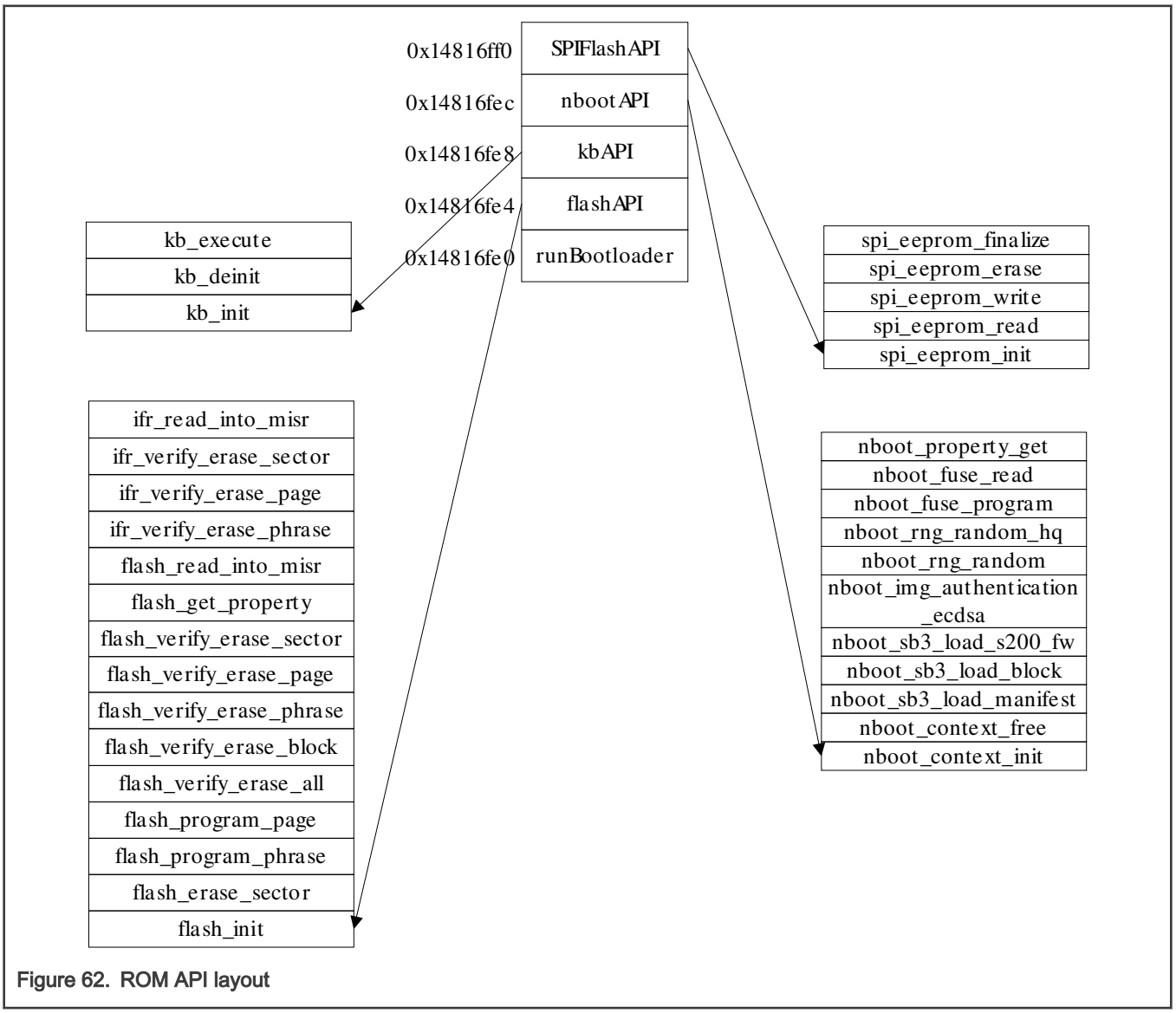


Figure 62. ROM API layout

11.2 Flash API

Flash APIs implement all the flash operation supported by CM33 FMU and RF-FMU1. Please note that operations to RF_FMU1 are only allowed when lifecycle is at OEM Open and Token indicates OEM1 ownership. Details of each API is described in this section.

11.2.1 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 142. 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                        (0x14816fe0u)
/** 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);
```

11.2.2 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 143. 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);
```

11.2.3 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 144. 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

Table continues on the next page...

Table 144. flash_program_phrase parameters (continued)

Parameter	Description
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).

11.2.4 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 145. 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.
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).

11.2.5 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 146. flash_verify_erase_all parameters

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

11.2.6 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 147. 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

11.2.7 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 148. 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
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).

11.2.8 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 149. 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).

11.2.9 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 150. 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).

11.2.10 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 151. 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

11.2.11 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 152. 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).

11.2.12 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 153. 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).

11.2.13 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 154. 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).

Table continues on the next page...

Table 154. ifr_verify_erase_sector parameters (continued)

Parameter	Description
lengthInBytes	The length, given in bytes (not words or long words) to be checked. The lengthInBytes must be sector-aligned(8192-byte).

11.2.14 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 155. 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).
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

11.2.15 flash_get_property

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

Prototype

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

Parameters

Table 156. 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 157. 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

11.2.16 Flash API status code

Table 158. 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

11.3 nboot API

The ROM API table is located at address 0x14816fe0 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.

11.3.1 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:

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

Parameters

Table 159. nboot_context_init parameters

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

11.3.2 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.

11.3.3 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 160. 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.

11.3.4 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 161. 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

11.3.5 nboot_sb3_load_s200_fw

This API is used to verify and decrypt sb3.1 file with the ELE firmware. Decryption is performed to the ELE RAM and firmware automatically started after successful load operation. NBOOT context must be initialized by the API nboot_context_init before calling this API.

Prototype:

```
nboot_status_t nboot_sb3_load_s200_fw(nboot_context_t *context, uint32_t *sb3Data);
```

Parameters

Table 162. nboot_sb3_load_s200_fw parameters

Parameter	Description
context	Pointer to nboot_context_t data structure in memory to store runtime state.
block	Pointer to the sb3.1 block with the ELE firmware.

11.3.6 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 163. 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,

Table continues on the next page...

Table 163. nboot_img_authenticate_ecdsa (continued)

Parameter	Description
	kNBOOT_OperationDisallowed = 0x5aa5cc33U, } nboot_bool_t;

11.3.7 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 164. 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.

11.3.8 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 165. 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.

11.3.9 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 166. 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

11.3.10 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 167. 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

11.3.11 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 168. 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

Table continues on the next page...

Table 168. nboot_property_get parameters (continued)

Parameter	Description
destData	Pointer to data buffer for storing returned contents
dataLen	Data buffer length

Property IDs supported are as follows:

Table 169. 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.
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
RADIO_IMG_OWNER	0x80	Returns 4 byte to indicate radio ownership as follows: 0x0 = NXP 0xAABBCCDD = OEM1 (Can be a radio firmware developer outside NXP based on token). 0xDDCCBBAA = OEM2
UUID	0x90	Returns UUID to serve as device unique identifier.

11.3.12 nboot API status code

The following table lists the nboot API status code.

Table 170. 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

11.4 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.

11.4.1 kb_init

This API is used to initialize bootloader and nboot context necessary to process sb3 file format.

Prototype:

```
status_t kb_init(void);
```

This API is expected to return kStatus_Success (0) on success and kStatus_Fail (1) on failure.

11.4.2 kb_deinit

This API is used to release nboot context and finalize sb3 file processing.

Prototype:

```
status_t kb_deinit(void)
```

11.4.3 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 sblaoder command “programFuses” then voltage must be regulated for over-drive and normalize voltage once operation is completed.

Prototype:

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

Parameters

Table 171. kb_execute parameters

Parameter	Description
data	Pointer to start of sb file data in memory
dataLength	sb file data length in bytes
isUpdateExt	Indicator for update(sb) file start address is in internal or external flash. 0x74784578u = External flash Other values = Internal flash

11.5 SPI Flash API

11.5.1 spi_eeprom_init

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

Prototype:

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

Parameters

Table 172. spi_eeprom_init parameters

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

11.5.2 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 173. 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.

11.5.3 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 174. 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.

11.5.4 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 175. spi_eeprom_erase parameters

Parameter	Description
address	Start address of the SPI Flash memory to be erased.

option	Following are the options supported for erase operation:
	`typedef enum`
	`{`
	`  kSize_ErasePage = 0x1,`
	`  kSize_Erase4K = 0x2,`
	`  kSize_Erase32K = 0x3,`
	`  kSize_Erase64K = 0x4,`
	`  kSize_EraseAll = 0x5,`
	`} eraseOptions_t;`
	NOTE
	kSize_ErasePage option is not supported for all memory models. Example: Adesto flash can support page erase commands but not Micron.

11.5.5 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);
```

11.5.6 SPI Flash API status code

The following table lists the SPI Flash API status code.

Table 176. SPI Flash API status code

Status	Code
kStatus_Success	0
kStatus_Fail	1

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 ELE changes

No substantial content changes.

A.4 S3MU changes

No substantial content changes.

A.5 Lifecycle changes

No substantial content changes.

A.6 Key Management changes

- Added Bluetooth LE 5.x secure connections key generation function f5.

A.7 ELE Software Architecture and API changes

- Added DERIVE_KEY command.

A.8 Flash Memory Controller (FMC-NPX) module changes

No substantial content changes.

A.9 ROM Bootloader changes

No substantial content changes.

A.10 ISP Path changes

No substantial content changes.

A.11 ROM API 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.

EdgeLock — is a trademark 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: 9 December 2025
Document identifier: KW45SRM