AN14729

Getting Started with EdgeLock Accelerator (CSEC) on MCX E24x

Rev. 1.0 — 22 July 2025 **Application note**

Document information

Information	Content
Keywords	AN14729, MCX E24x, CRA requirements, EdgeLock Accelerator (CSEC)
Abstract	This application note explains the features and functionalities offered by the EdgeLock Accelerator (CSEC) hardware security module on MCX E24x. The EdgeLock Accelerator (CSEC) has been implemented in NXP's MCX E2 series of devices.

1 Introduction

This application note explains the features and functionalities offered by the EdgeLock Accelerator (CSEC) hardware security module on MCX E24x. The EdgeLock Accelerator (CSEC) has been implemented in NXP's MCX E2 series of devices.

This application note lists several security use cases, provides an overview of the EdgeLock Accelerator (CSEC) module, describes how to program the EdgeLock Accelerator (CSEC), how to protect the application code, and how to establish secure communication. This document also provides a first guidance for the most typical functions to be used with the EdgeLock Accelerator (CSEC) in the form of the MCUXpresso IDE examples. This document mainly focuses on the hardware features provided by the EdgeLock Accelerator (CSEC) module.

Why is cryptography needed?

The industrial market is rapidly advancing toward the age of smart manufacturing and connected devices. These technological advances require industrial control systems to be connected to the Internet and other communication media, such as IIoT infrastructure, to a greater extent than ever before. As a result, industrial electronic systems are becoming more susceptible to hacking efforts, which attempt to gain unauthorized access to factory data or control systems. To prevent unauthorized access, every part of the industrial electronic system must be secured, from small microcontrollers managing small tasks to larger gateway processors controlling complex systems, and from the application software to the data stored in the memory. All of these industrial systems are important to the safety and security of the facility and its operators. Embedded security modules and cryptographic engines provide effective tools for ensuring industrial safety and security by enabling secure information exchange and data authenticity and integrity.

Some vital security use cases include the following:

- Component protection
- Secure communication

1.1 EdgeLock Accelerator (CSEC) security module features

The major EdgeLock Accelerator (CSEC) has the following security module features:

- Secure cryptographic key storage (ranging from 3 to 17 user keys)
- AES-128 ECB (Electronic Code Book) Mode - encryption and decryption
- AES-128 CBC (Cipher Block Chaining) Mode - encryption and decryption
- AES-128 CMAC (Cipher-based Message Authentication Code) generation and verification
- True and pseudorandom number generation
- Miyaguchi-Preneel compression function (available via API)
- Secure boot mode (user configurable)
  - Sequential boot mode
  - Parallel boot mode
  - Strict sequential boot mode (unchangeable once set)

Note: *The EdgeLock Accelerator (CSEC) is not intended to encrypt the code flash content.*

2 Security use cases

This section describes some use cases and how these cases can be supported by the CSEC. Many of these use cases assume that the application code was verified with the CSEC secure-boot function before the execution.

2.1 Component protection

Component protection prevents dismantling a single controller from an industrial system and reusing it in other systems. Manufacturers can now address several issues with a secure component-protection scheme. In a component-protection system based on the CSEC, the most valuable controllers include a processor with a CSEC module. A main node, which may be assigned by design or dynamically with a specific algorithm, polls all controllers of the component protection system and requests a specific answer (the unique ID) encrypted by the CSEC. In this case, only CSEC-enabled controllers with the right secret key can send back a valid response. Additionally, the main node can crosscheck the unique ID with a database of all assembled modules in this specific industrial system. This component check can be done periodically while the system is operating. If the system detects an unauthorized controller in the industrial network, it can react to it.

2.2 Secure communication

Industrial electronic systems are now open to communicate with external devices, such as mobile devices, other industrial equipment, and other infrastructures. These systems are prone to hacking attempts. If these hacking attempts are prevented right at the point of entry, that is at the point of the communication medium, then the industrial system is prevented from malfunctioning. The secure communication can be realized when the controller must communicate with any other controller. For example, the TLS protocol of Ethernet and other secure communication applications. CSEC's random number, encryption, and key management functions can be used to achieve encryption (hides the data being transferred from third parties), authentication (ensures that the parties exchanging information are who they claim to be), and integrity (verifies that the data has not been forged or tampered with) between devices.

3 EdgeLock Accelerator (CSEC) overview

The main functionality of the CSEC is implemented in the core of the Flash Memory Module (FTFC). By using an embedded processor, firmware, and a hardware-assisted AES-128 subblock, the FTFC module enables encryption, decryption, and CMAC generation-verification algorithms for secure messaging applications. Additional APIs are also available for the secure boot configuration, True Random Number Generation (TRNG), and Miyaguchi-Preneel compression. Figure 1 shows the high-level block diagram of the CSEC. The FTFC core takes care of the flash as well as CSEC functionalities. A RAM is also dedicated to the flash core to improve its performance and it is only accessible to the FTFC core. The host interface is a medium for the system core to talk to the FTFC module and a way to issue and get the control and status information, respectively. The block at the right-hand side of Figure 1 shows physical memories and the CSEC Parameter space Random Access Memory (CSEC PRAM). They are explained in detail throughout this section. The flash and PRAM controllers are responsible for efficient working of the FTFC system and are out of scope of this application note. These controllers connect physical memories and the CSEC PRAM to the crossbar switch via the Memory Protection Unit (MPU).

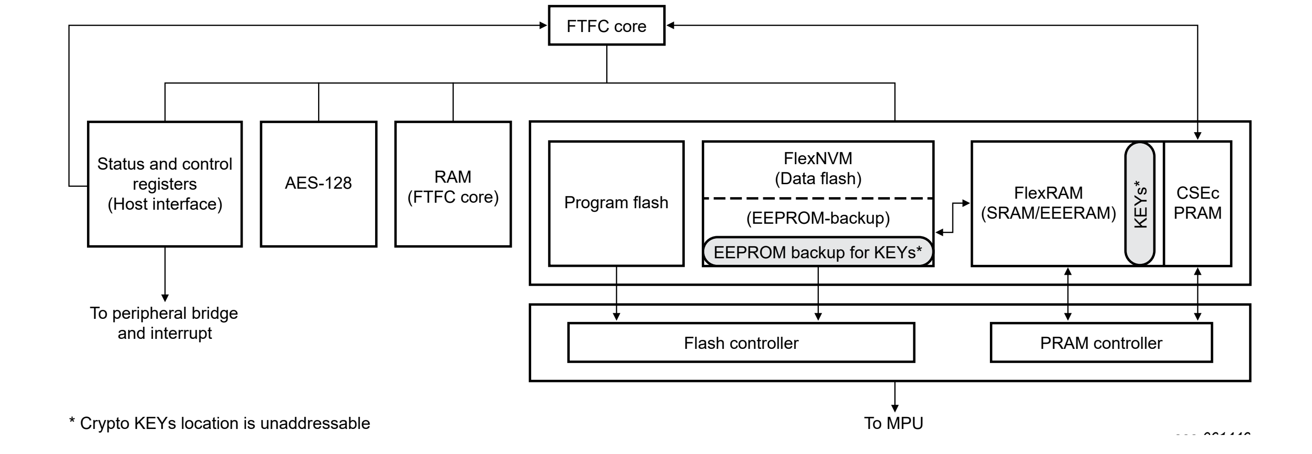


Figure 1. CSEC block diagram

Before going into the detail of the CSEC, we discuss the high-level idea of different physical memories. There are three types of flash memories:

1. Program Flash
2. FlexNVM
3. FlexRAM

Program flash is a nonvolatile flash memory used to store the application code.

FlexNVM is the flexible nonvolatile memory, which can be "flexed" between the "normal flash" operation and "emulated-EEPROM" operation. During "normal flash" operations, it can be used to store application code or data. During the "emulated-EEPROM" operations, it can be used as a nonvolatile backup memory for emulated-EEPROM data.

Similarly, **FlexRAM** is the flexible RAM (volatile) and can be "flexed" between the "normal SRAM" operation and "emulated-EEPROM" operation. During the "normal SRAM" operation, it can be used in addition to the main SRAM. During the "emulated-EEPROM" operation, it can be used as a high-endurance emulated EERAM (EEERAM).

As shown in Figure 1, the FlexNVM and FlexRAM together emulate the EEPROM storage. In the emulated form, FlexNVM is called EEPROM-backup and FlexRAM is called EEERAM. The user/application program directly interfaces with EEERAM for emulated-EEPROM operations. For example, if you want to write to the emulated-EEPROM, write to the EEERAM and internally, the flash system locks the interface and writes data back to the EEPROM-backup for nonvolatile update. On every power-up, data is retrieved from EEPROM-backup and copied to the EEERAM for use/application use. In simple language, EEERAM is the point of contact for user/application to talk to emulated-EEPROM storage.

Through the host interface, you can configure the FTFC module for emulated-EEPROM operation by issuing a Program Partition Command (PRGPART). The Flash Common Command Objects (FCCOB) registers are used to issue the PRGPART command. The PRGPART command also provides flexibility to specify the partition size between the emulated-EEPROM operation and normal operations, according to your wish.

Now, let’s get back to the CSEC operations.

To enable the CSEC functionality, configure the device for the emulated-EEPROM operation. The PRGPART command is used to enable the CSEC and it also provides a mechanism to specify the key size. Depending on the key size, the last 128/256/512 bytes of EEERAM are reduced from the emulated EEPROM and become unaddressable (as the corresponding EEPROM backup). This storage is secured and utilized to store cryptographic keys. Further, this storage is not accessible by any other bus initiators from the system. The secure storage areas are shown as grey sections in Figure 1.

Once you configure the FTFC module for the CSEC functionality and load the user keys for the security operations, the device is ready for any security-related operations.

The CSEC PRAM interface is used to supply data and command header for security operation. This process involves transferring blocks of system memory contents into the CSEC PRAM space for a cryptographic operation once the operation is completed. Then it transfers the results back to the system memory. All data block sizes are 128-bit and must be padded by the application if the block size is less than 128 bits. The CMAC operations do not require padding because padding is taken care of internally.

Once the CSEC command header is written, the command execution starts and the CCOB interface, and the EEERAM and CSEC PRAM are locked. No other commands may be initiated until the completion of the ongoing one.

The status of the CSEC-related operations is reported in the Flash Status Register (FSTAT) and CSEC Status Register (FCSESTAT). Using this status information, you can generate an interrupt on completion of a CSEC command. Any error occurring during the CSEC command execution is reported in the error bits position of the CSEC PRAM.

Note:

1. *No CCOB or CSEC commands are available when the part is in the VLPR (Very Low Power) or HSRUN (High Speed Run) modes.*
   - *The CCOB and CSEC command operations can only be performed when the device is in the RUN mode. If you want to switch to any other mode (HSRUN mode to operate at a higher frequency or VLPR mode to decrease the power consumption value) before switching to any power mode, all command operations must be completed first (by polling for the FSTAT[CCIF] flag). If the device runs in the HSRUN/VLPR mode and the CCOB/CSEC operation is needed, switch to the RUN mode to perform the write. After it finishes, you can switch back to the HSRUN/VLPR mode.*
2. *It is not possible to execute CCOB commands (flash program, erase, and so on) and CSEC commands concurrently.*
3. *It is also not possible to execute a different CSEC command during execution of an ongoing CSEC command. However, it is possible to issue a CCOB command in the middle of a CSEC command, but this cancels the ongoing CSEC command.*
4. *The execution of a CSEC command while in the Erase Suspend (ERSSUSP) state results in the suspended erase operation being aborted (not able to be resumed).*
5. *Starting the execution of the CCOB or CSEC commands locks out the CCOB interface, EEERAM, and CSEC PRAM. These are unlocked when those commands complete.*

3.1 Cryptographic keys

To encrypt and decrypt the data, use cryptographic keys. This section provides insight into the CSEC's key management policy.

Table 1 and Table 2 summarize the available key types and their properties such as ID, memory type, size, attributes, and so on. This section describes all keys and their properties.

Table 1. Nonvolatile keys and RAM_KEY

Key name	Key ID		Memory type	Key size (bytes)	Key counter size (bits)	Key attributes (bits)						Factory default state
	KBS	KEY IDs				Write prot	Boot prot	Debugger prot	Key usage	Wild card	Verify only	
Secret_Key	X	0x0	Nonerasable	16	-	-	√	√	-	-	-	Written by NXP
UID	X	0x0	Nonerasable	15	-	-	-	-	-	-	-	Written by NXP
MASTER_ECU_KEY	X	0x1	Nonvolatile	16	28	√	√	√		√		Empty
BOOT_MAC_KEY	X	0x2	Nonvolatile	16	28	√		√		√		Empty
BOOT_MAC	X	0x3	Nonvolatile	16	28	√		√		√		Empty
KEY_01-KEY_10	1'b0	0x4-0xD	Nonvolatile	16	28	√	√	√	√	√	√	Empty
KEY_11-KEY_17	1'b1	0x4-0xA	Nonvolatile	16	28	√	√	√	√	√	√	Empty
Reserved	1'b1	0xB-0xD	-	16	-	-	-	-	-	-	-	-

Table 1. Nonvolatile keys and RAM_KEY*...continued*

Key name	Key ID		Memory type	Key size (bytes)	Key counter size (bits)	Key attributes (bits)						Factory default state
	KBS	KEY IDs				Write prot	Boot prot	Debugger prot	Key usage	Wild card	Verify only	
Reserved	1'b0	0xE	-	-	-	-	-	-	-	-	-	-
RAM_KEY[1]	X	0xF	Volatile	16	-	-	-	-	-	-	-	Undefined after every reset

[1] RAM_KEY has only the PLAIN_KEY flag and is only implemented for the RAM_KEY. It is set by CSEC when the key is loaded into the RAM_KEY slot as a plain text.

√ indicates the attributes available to the key.

Table 2. Other volatile keys

Key name	Address	Type	Size (bytes)
PRNG_KEY	N/A	RAM	16
PRNG_STATE	N/A	RAM	16

The CSEC module provides a secure and nonvolatile storage for cryptographic keys. The first five slots have a dedicated use. The remaining slots are available for application-specific keys.

- The SECRET_KEY is programmed with a random value during device fabrication, whose value is never disclosed and it is used internally to generate derived keys; for example, the Pseudo Random Number Generator key (PRNG_KEY).
- The UID unique identifier number is unique for every part and it is programmed into the secure flash when it is tested in the wafer form.

Note: *Both the SECRET_KEY and the UID are unaddressable and unalterable.*

- The MASTER_ECU_KEY is used to reset the CSEC to factory state or to change any other keys.
- The BOOT_MAC_KEY is used by the secure boot process to verify the authenticity of the software.
- The BOOT_MAC slot is loaded with a MAC value used by the secure boot process. It can be loaded automatically by the CSEC under specific circumstances or manually by user software. See Section 4.4.3 for a detailed description.
- The KEY_01 to KEY_17 user keys are stored in the EEERAM space with a configurable amount of space for user keys. Anywhere from 3 to 17, the user keys can be configured using the program partition command.
- The RAM_KEY volatile key can be used for any arbitrary operations.

Note: *Since the key loaded into the RAM_KEY is stored externally and not under control of the CSEC, it is vulnerable to attacks.*

- The PRNG_KEY and PRNG_STATE are not directly accessible by any user functions and are used internally by a pseudorandom number generator.

Note: *The MASTER_ECU_KEY, BOOT_MAC_KEY, BOOT_MAC, and user keys can be populated, as described in Section 4.2. Each key has its associated properties. This includes the mechanisms to index the key, to count the number of key updates, and to implement different security attributes.*

3.1.1 KeyID: {KBS, Key IDx}

Each key has an identification number associated, which is called KeyID. This number is used to identify the key being used, updated, or authorizing the update. The Key Block Select (KBS) is used to select the bank of the key. Using the KBS and KeyID together, you can index any key from 1 to 17.

The keys 1-10 are in bank 0 and keys 11-17 are in bank 1. For example, KEY_11: {KBS, Key IDx} = {1,0x4}

3.1.2 Key counter

Each user key has a counter to keep track of updates. This counter must be increased on every update. The counter is 28 bits long. The new counter value is used in the derivation of M2 (described in the following sections and Section 8) when a key is updated.

3.1.3 Key attributes

Each key has six flags associated with it. These flags determine how and under what conditions that key can be used. These values are included while deriving M2 (see Section 8).

Write Protection Flag (WRITE_PROT)

If set, the key cannot ever be updated, even if an authorizing key is known.

Note: *Set this flag with caution. Setting this flag is an irreversible step and it prevents the device from being reset to the factory state.*

Boot Protection Flag (BOOT_PROT)

If set, the key cannot be used if the MAC value calculated in the SECURE_BOOT step does not match the BOOT_MAC value stored in the secure flash. If the secure boot fails, the keys marked BOOT_PROT remain locked and cannot be used during the application execution.

Debugger Usage Protection Flag (DEBUG_PROT)

If set, the key cannot be used if a debugger is (or has ever been) connected to the MCU since it was last reset.

Key Usage Flag (KEY_USAGE)

This flag determines if a key is used for encryption/decryption or for CMAC generation/verification. If the flag is set, the key is used for CMAC generation/verification. If the flag is clear, the key is used for encryption/decryption.

Wildcard Protection Flag (WILDCARD)

If set, the key cannot be updated by supplying a special wildcard UID (UID = 0).

Verify Only Flag (VERIFY_ONLY)

This functionality can only be used if SFE = 0x01 during the PGMPART configuration settings.

If set, the key cannot be used by the GENERATE_MAC command and can only be used by the VERIFY_MAC command. If KEY_USAGE==0, then this attribute has no effect.

3.2 Generic EdgeLock Accelerator (CSEC) PRAM interface

After getting familiar with the keys, their types, and their properties, let's get familiar with the programming interface that the CSEC uses for security-related operations. The PRAM interface issues the CSEC commands and passes data to the CSEC interface for security operations. The CSEC PRAM is organized into eight 128-bit RAM pages. They can be accessed as a word or a byte.

Table 3. Generic CSEC PRAM interface

Bits	[127:0]															
Bits	31:24	23:17	15:8	7:0	31:24	23:17	15:8	7:0	31:24	23:17	15:8	7:0	31:24	23:17	15:8	7:0
WD	Word 0				Word 1				Word 2				Word 3			
Byte	3	2	1	0	7	6	5	4	B	A	9	8	F	E	D	C
PAGE																

Table 3. Generic CSEC PRAM interface*...continued*

0	FuncID	Func format	CallSeq	KEYID	Error bits	Command-specific
1	Data input to CSEC or data output from CSEC					
2						
3						
4						
5						
6						
7						

As shown in Table 3, the first page (Page 0) includes the command header (Page 0, Word 0) and the message control length (Page 0, Word 3). The rest of the pages are used for the input/output data information.

Writing to the command header triggers the macro to lock the CSEC PRAM interface and start the CSEC operation. To set up a CSEC command, enter the data information followed by the message length information and write the command header last.

When you write the command header, the CSEC starts the execution by resetting the Command Complete Interrupt Flag (CCIF) field in the Flash Status Register (FSTAT[CCIF] == 0). The completion of the command is indicated by FSTAT[CCIF] == 1.

Depending on the command and upon completion (FSTAT[CCIF] == 1), you can read the required data back from the CSEC PRAM location.

When the data cannot fit into the CSEC PRAM, the CSEC requires repeating the same command with new data. To continue the same command, write the remaining data in the CSEC PRAM as applicable, followed by the updated command header information. In the command header, change only the "CallSeq" field and leave the other information the same. Changing the FuncID field while continuing the previous command results in a sequence error. The FuncID field must stay consistent when the previous command continues.

Section 3.2.1 and Section 3.2.2 describe how to enter the data and command information into the CSEC PRAM interface.

3.2.1 Writing data and message length information in PRAM interface

You can write the data and message length information by the usual means at the CSEC PRAM locations specified by the command being used.

3.2.2 Command header

The structure of the command header is standard for all commands. The command header is divided into six bytes, as shown in Figure 2.

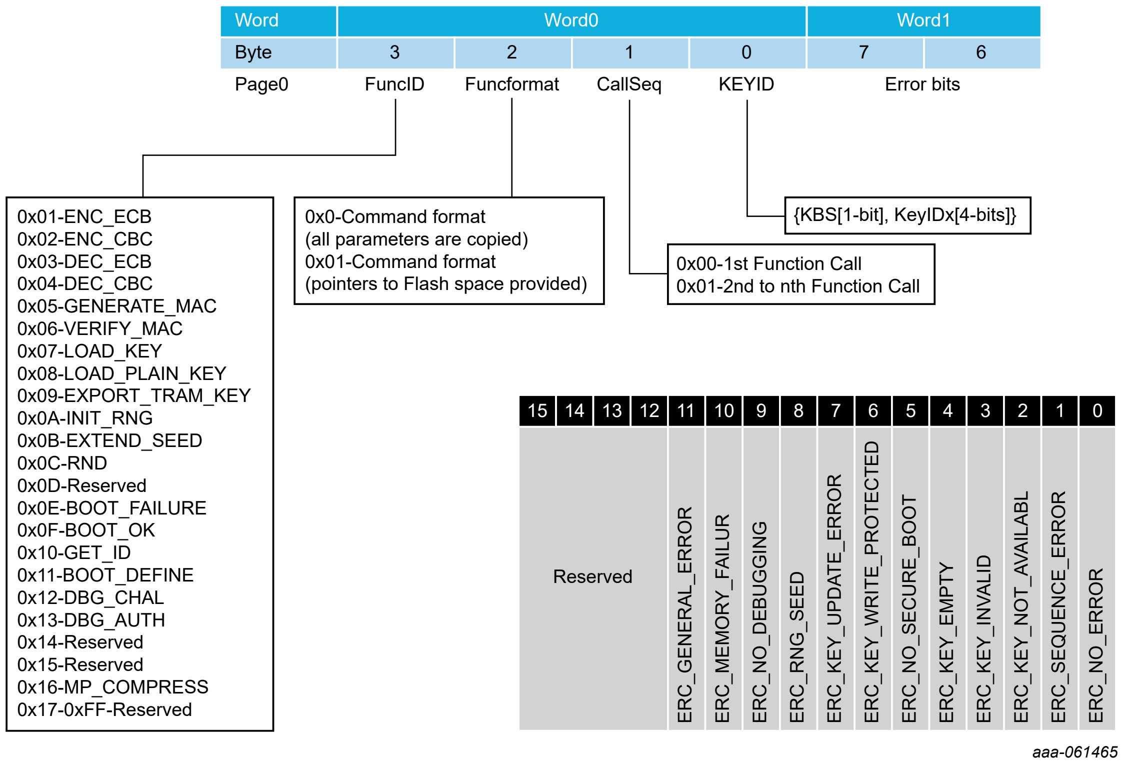


Figure 2. CSEC command header

FuncID: The Function Identification (ID) field has eight bits and specifies the security command to execute. Its valid values range from 0x0 to 0x16.

FuncFormat: The function format specifies how the data is transferred to/from the CSEC. There are two use cases.

The first and most common method is to copy all data to the PRAM. Here, the main core or DMA copies the data and issues the function call.

The second use case is the pointer and function call method. For the pointer case (only available for two CMAC commands, GENERATE_MAC and VERIFY_MAC), the main core or DMA provide the pointer information.

The valid values are as follows:

- Copy method: 0x00
- Pointer method: 0x01

CallSeq: The call sequence specifies whether the information goes first or follows a function call. When the amount of data used for a command exceeds the size of the PRAM, CallSeq specifies the continuation of the command. For example, if you have more than seven 128-bit blocks of data for CMAC verification, this field indicates the continuation of data during the subsequent call.

The valid values are as follows:

- First function call: 0x00
- Second through to nth function call: 0x01

KeyID: {KBS, KeyIDx}: It is divided into two parts. The KeyIDx is a 4-bit value, pointing to the key to be used. The KBS (Key Block Select - not used in all commands) is a 1-bit value, allowing you to switch between the key banks. It is described in Section 3.1.

Table 4. KeyID format

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	0	0	0	KBS	KeyIDx			

Note: *Bits [7-5] must always be 0. Writing 1 results in the addressing of an incorrect key. If the KBS is not used for a particular key, write 0 to bit [4].*

Error bits: A 16-bit error bit field returns the error information after a command execution. See the device reference manual for more details.

3.2.3 EdgeLock Accelerator (CSEC) status, error, and interrupt reporting

The CSEC clears the FSTAT[CCIF] flag when the operation starts and sets the FSTAT[CCIF] flag when the operation completes. This flag can generate an interrupt by setting the Command Complete Interrupt Enable (CCIE) field in the Flash Configuration Register (FCNFG[CCIE] = 1).

Any error occurring during the CSEC command execution is reported in the "Error Bits" field of the command header.

The CSEC Status Register (FCSESTAT) of the FTFC module reports the status of the CSEC. The IDB and EDB bits indicate whether internal and external debugging features are enabled. The RIN bit is set upon the initialization of the random number generator. The BSY is set when CSEC-specific command processing is ongoing. The BOK, BFN, BIN, and SB bits are secure-boot specific bits and they are described in Section 4.4.3.2. For more information, see the device reference manual.

Table 5. FCSESTAT register

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	IDB	EDB	RIN	BOK	BFN	BIN	SB	BSY

4 Programming the EdgeLock Accelerator (CSEC) security module

To program the device for security operations, perform the following steps:

1. Configure the part for CSEC operations: PRGPART: Section 4.1 (mandatory)
2. Add/update/erase the keys in the secure memory: Section 4.2 (mandatory)
3. Use the CSEC APIs for the security operations: Section 4.3 (optional)
4. Check the application for the authenticity on every boot: Section 4.4 (optional)
5. Reset the flash and disable the CSEC functionality: Section 4.5 (optional)

4.1 PGMPART program partition command

To start using the CSEC command set, configure the device into the emulated-EEPROM mode. The PGMPART command defines the CSEC key size and FlexNVM/FlexRAM partition between the normal and EEPROM operations. It also defines whether the VERIFY_ONLY functionality is enabled and whether the FlexRAM (EEERAM) is loaded with valid EEPROM data during the reset sequence. Any error that occurs during command execution can be observed in the FSTAT register. For more information, see the Program Partition Command section of the device reference manual.

Note:

- *The emulated EEPROM back-up size must be at least 16 times the EEERAM size.*
- *Before launching the Program Partition command, the data flash IFR must be in an erased state, which can be accomplished by executing the Erase All Blocks command or by an external request (see the Erase All Blocks command in the device reference manual).*

- *It is highly recommended to see the Program Partition Command section of the device reference manual.*

Example code:

The Configure Part and Load Keys example demonstrates how to issue the PGMPART command. This example is developed in the MCUXpresso IDE and runs on the NXP FRDM-MCXE248 board. The "Load Keys" part of this example is explained in Section 4.2.1. This example configures the EEPROM to the following settings (see Table 6):

- Key size = MASTER_ECU_KEY, BOOT_MAC_KEY, and BOOT_MAC + 17 user keys
- VERIFY_ONLY functionality = disabled
- FlexRAM (EEERAM) status = it is loaded with valid data on reset
- EEPROM partition: EEERAM = 4 kB and EEPROM backup = 16 x 4 = 64 kB

Since EEERAM == 4 kB, the available emulated-EEPROM data set is also 4 kB.

The key specific size is subtracted from the total emulated EEPROM available for application use. For example, with 20 keys, 3.5 kB (4k-512) of emulated EEPROM is available for application use.

Table 6. Flash Common Command Objects registers (FCCOB) requirements for Program Partition command

FCCOB number	FCCOB contents [7.0]
0	0x80 (PGMPART)
1	CSEC key size
2	SFE
3	FlexRAM load during reset option (only bit 0 used): 0 - FlexRAM loaded with valid EEPROM data during reset sequence 1 - FlexRAM not loaded during reset sequence
4	EEPROM data set size code
5	FlexNVM partition code

When the device is configured for the CSEC operation, the FCNFG register fields are set as follows:

```
FCNFG[RAMRDY] == 0 and FCNFG[EEERDY] == 1
```

Figure 3 shows the flash memory map before and after partitioning on MCXE248 with:

- CSEC enabled with 20 keys
- Highest endurance (the whole data flash is used as the EEPROM backup)

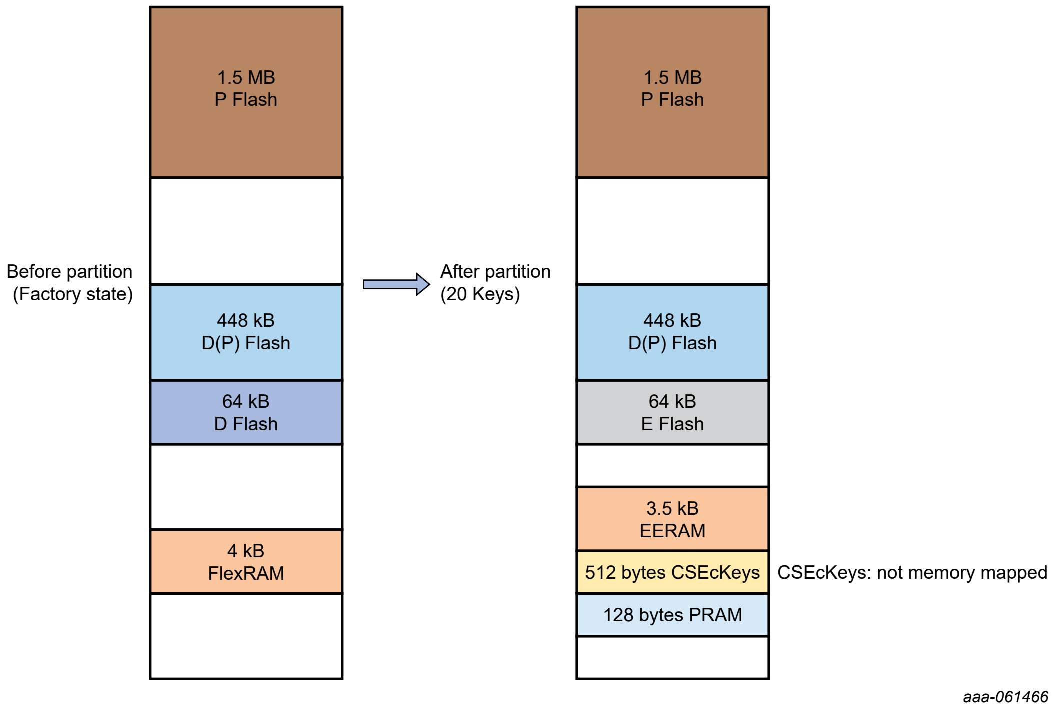


Figure 3. Flash memory map before and after partitioning on MCX E248

4.2 Key management

The key management includes the procedures to add keys to the secure memory locations and to update existing keys with new keys.

4.2.1 Adding keys to secure memory slots

Note: *This section is applicable to MASTER_ECU_KEY, BOOT_MAC_KEY, BOOT_MAC, and all user keys.*

To add keys, CSEC uses the protocol defined in the HIS-SHE specification (HIS-SHE Functional Specification, v1.1, Section 9.1, "Description of memory update protocol"). This ensures confidentiality, integrity, authenticity, and protects against replay attacks. To update the memory containing the keys, HIS-SHE requires to calculate the following and pass it to the CSEC:

- $M1 = UID^{'}|ID|AuthID$ - 128 bits
- $M2 = ENC_{CBC,\ K1,\ IV=0}\left(CID'|FID'|\text{“}0\ldots0\text{”}95|KID'\right)$ - 256 bits: SFE == 0x00
- $M3 = CMAC_{K2}\left(M1 \mid M2\right)$ - 128 bits

M1-M3 is derived on an offline computer and created as arrays in a header file. The values are also derived using the target part (MCX E24x) and the CSEC. The CSEC PRAM pages from 1 to 4 are loaded with M1, M2, and M3. The CMD_LOAD_KEY command header must be written (to page 0) to start the command execution. After the execution is completed, the CSEC PRAM pages from 5 to 7 return M4 and M5.

- $M4 = UID|ID|AuthID|M4^{*}$ - 256 bits
- $M5 = CMAC_{K4}\left(M4\right)$ - 128 bits

These values can be verified against the precalculated offline values for a flawless addition of the key.

Note: *See Section 8 for a detailed explanation of M1 to M5.*

Note:

- *If the key to be updated is not wildcard-protected (WILDCARD == 0), you can use UID = 0 to generate M1 and M3. Otherwise, the device UID must be read and used in the generation of M1 and M3. The UID can be established as described in Section 4.3.2.*
- *For a blank key, the key being updated has an initial value of "all 1s". This is a very specific case for a device in its factory state and the authorizing key is the key itself or MASTER_ECU_KEY (if programmed). Substitution of the authorizing key value is required in all other cases.*

Example code:

Example-1: The Configure Part and Load Keys example is developed in the MCUXpresso IDE and runs on the NXP FRDM-MCXE248 board.

This program configures the part for the CSEC operations, initializes the random number generator, and loads the MASTER_ECU_KEY, KEY_1, and KEY_11. This code calculates M1 to M3 during runtime using the device resources.

Observe the error status after each operation to verify the intended operation outcome.

4.2.2 Updating key

Note:

- *This section is applicable to MASTER_ECU_KEY, BOOT_MAC_KEY, BOOT_MAC, and all user keys.*
- *If a key has its WRITE_PROT attribute set, you can no longer update that key.*

After programming a device's keys into the secure flash and when the device is no longer in its factory state, it may be necessary to update one or more keys. This is done using the same CMD_LOAD_KEY command and procedure used for adding keys described in Section 4.2.1.

4.2.2.1 Authorization

To keep keys secure, the CSEC requires knowing an authorizing key before updating a specific key. In general, the knowledge of a specific key is needed to update that specific key. MASTER_ECU_KEY is a key with a special meaning and can be used to authorize the update of all keys (BOOT_MAC_KEY, BOOT_MAC, and all KEY_1 to KEY_17) without knowledge of those keys. See Table 7.

Table 7. Keys to update other keys

Key to update	Keys which must be known to update other keys				
	MASTER_ECU_KEY	BOOT_MAC_KEY	BOOT_MAC	KEY_<N>	RAM_KEY
MASTER_ECU_KEY	√				
BOOT_MAC_KEY	√	√			
BOOT_MAC	√	√			
KEY_<N>	√			√	
RAM_KEY				√	

- To update a particular key, the knowledge of any other key is sufficient. For example, to update BOOT_MAC_KEY, the knowledge of either MASTER_ECU_KEY or BOOT_MAC_KEY is sufficient.
- The knowledge of MASTER_ECU_KEY enables updating of all user keys except RAM_KEY.

4.2.2.2 Update process

For a successful key update, increase the counter value associated with that key. The process for updating a given key is the same as that described in Section 4.2.1.

Example code:

Example-2: The Update User Keys example updates the already programmed user key. Run example 1, which programs the user keys, and then run this program to update them.

This program uses MASTER_ECU_KEY as the authorizing key and updates KEY_1 and KEY_11 to the new value. This code calculates M1 to M3 during run-time using the device resources. Observe the error status after each operation to verify the intended operation outcome.

4.2.2.3 Erasing keys

No individual key can be erased. It is possible to erase all keys together. The procedure for erasing all keys is described in Section 4.5.

Note: *All keys must be erased to issue the flash mass erase.*

4.3 Basic operations

After getting familiar with the CSEC architecture and feature set and learning how to add and update keys, this section describes basic operations, such as UID retrieval, AES-128 ncrypting and decrypting, CMAC generation and verification, and random number generation.

4.3.1 Random number generation

The PRNG has a 128-bit state variable and uses the AES in the output feedback mode to generate pseudorandom values. A key derived from SECRET_KEY is used for the PRNG. The RND command updates the state of the PRNG and returns a 128-bit random value. The CMD_EXTEND_SEED command can be used to add entropy to the PRNG state. The PRNG state must be initialized after each reset with the CMD_INIT_RNG command, which uses the internal TRNG to generate a 128-bit seed value for the PRNG. PRNG uses PRNG_STATE/KEY and seed per the HIS-SHE specification and the AIS20 standard. Run CMD_RND to generate a random number.

Note: *Initialize PRNG before issuing the CMD_EXTEND_SEED, CMD_RND, and CMD_DBG_CHAL commands.*

Example code to generate a random number:

Example-3: See the Basic Operations example to learn how to initialize PRNG and generate a random number.

4.3.2 UID retrieval

The Unique Identifier Number (UID) is unique for every part and it is programmed into the secure memory location when the part is tested in the wafer form. The UID is 120 bits long. The UID can be used during communications between components within an IIOT device to confirm that external controllers are not substituted. The UID is also used in the process of resetting parts to their factory state.

You can obtain the UID by issuing the GET_ID command.

Note: *The GET_ID command returns the UID and MAC. If MASTER_ECU_KEY is empty, then the MAC return value is set to 0.*

Example code for retrieving UID from secure flash:

Example-3: See the Basic Operations example.

4.3.3 AES-128 encryption and decryption

The CSEC supports the AES-128 encryption and decryption in the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes of operation. The key is selected from a memory slot, which must be enabled for the encryption (KEY_USAGE = 0, see Section 3.1.3).

For a key that is not stored in a nonvolatile memory slot, load a plain text key into the RAM_KEY slot using the CMD_LOAD_PLAIN_KEY command. However, as this method implies a potential security risk, this is only useful for development or debug purposes.

Since the command takes length in terms of PAGE_LENGTH, the data must be presented in 128-bit blocks. Any required padding must be done by the application.

CMD_ENC_ECB, CMD_ENC_CBC, CMD_DEC_ECB, and CMD_DEC_CBC are used for these operations.

Example code for AES-128 encryption and decryption:

Example-3: See the Basic Operations example to learn how to encrypt and decrypt using the Cipher Block Chaining (CBC).

4.3.4 CMAC generation and verification

CSEC uses the AES-128 CMAC algorithm for message authentication. The key for the CMAC operation is selected from a memory slot. The key must be enabled for verification (KEY_USAGE = 1; see Section 3.1.3).

For a key that is not stored in a nonvolatile memory slot, you can load a plain text key into the RAM_KEY slot using the CMD_LOAD_PLAIN_KEY command. However, as this method implies a potential security risk, this can only be useful for development or debug purposes.

The CMD_GENERATE_MAC command uses a key (KEY_ID) to encode a MAC value (128 bits) for the full message body.

The CMD_VERIFY_MAC command supports comparison of a calculated MAC with an input MAC value.

Since the command takes length in terms of number of bits, all required padding is taken care of internally.

Example code for CMAC generation and verification:

Example-3: See the Basic Operations example to learn how to generate the CMAC for the given data and how to verify it with the existing CMAC and data.

4.4 Secure boot

The CSEC has a mechanism which allows you to authenticate the boot code in the flash. You can configure the MCU so that on every boot, a section of code is authenticated and the generated MAC is compared with a value previously stored in a secure memory slot.

4.4.1 Secure boot modes

The MCX E24x devices support three secure boot modes.

These modes are supported only for the flash boot. They are not supported for other boot types (serial download, wakeup to RAM) as this may present a potential security issue.

1. Sequential Boot Mode: In this mode, after RESET, the flash system comes out of RESET and the core stays in RESET or it can execute. The secure boot process verifies the application firmware block. If the secure boot is successful, the keys become available for security tasks. Otherwise, the keys marked as boot-protected are blocked for all tasks. Lastly, the core starts executing the application firmware.
2. Strict Sequential Boot Mode: In this mode, after RESET, the flash system comes out of RESET and the core stays in RESET or it can execute from the ROM code. The secure boot process verifies the application

firmware block. If the secure boot is successful, the keys become available for security tasks. Otherwise, if the CMAC comparison fails, the main core stays in RESET (no application firmware is executed) or it may execute the ROM code.
Note: *This boot mode is irreversible. That means that once set, the boot mode cannot be changed to the other boot modes. Before setting this mode, calculate and store BOOT_MAC. Otherwise, the device remains in RESET. The automatic BOOT_MAC calculation does not run in this mode.*

3. Parallel Boot Mode: In this mode, after RESET, the flash system and the main core come out of RESET and the main core starts executing the application firmware. In parallel to the main core execution, CSEC verifies the application firmware block using the secure boot process. If the secure boot is successful, the keys become available for security tasks. Otherwise, the keys marked as boot-protected are blocked for all tasks. The main core can still execute the firmware.

4.4.2 Enabling secure boot

The CMD_BOOT_DEFINE command configures the boot mode (flavor) and boot code size (in bits) to authenticate.

Figure 4 and the description below illustrate the secure boot flow in the MCX E24x devices.

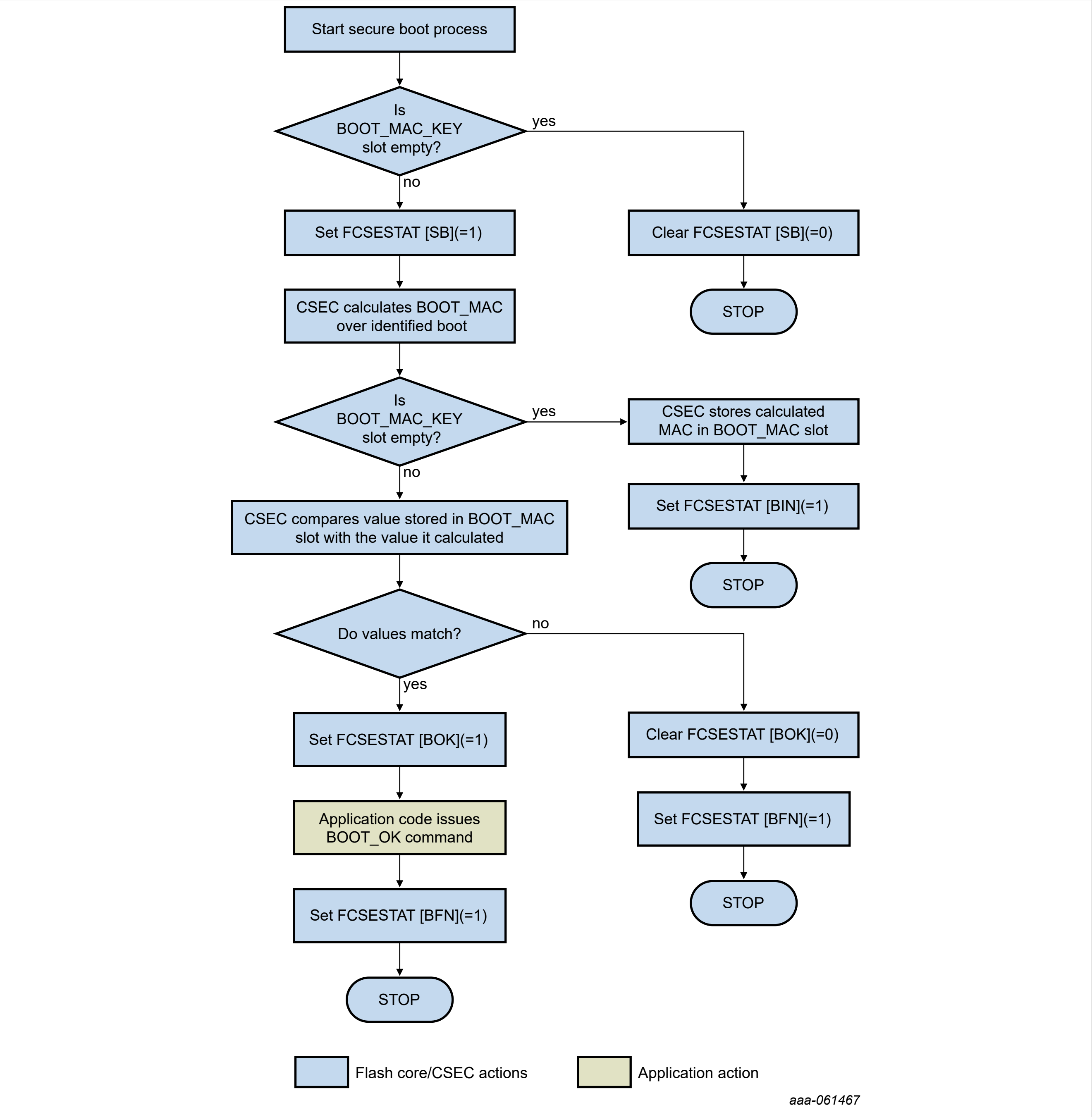

Start secure boot process
Is BOOT_MAC_KEY slot empty?
yes
no
Set FCSESTAT [SB](=1)
Clear FCSESTAT [SB](=0)
STOP
CSEC calculates BOOT_MAC over identified boot
Is BOOT_MAC_KEY slot empty?
yes
CSEC stores calculated MAC in BOOT_MAC slot
no
Set FCSESTAT [BIN](=1)
CSEC compares value stored in BOOT_MAC slot with the value it calculated
STOP
Do values match?
no
yes
Set FCSESTAT [BOK](=1)
Clear FCSESTAT [BOK](=0)
Application code issues BOOT_OK command
Set FCSESTAT [BFN](=1)
STOP
Set FCSESTAT [BFN](=1)
STOP
Flash core/CSEC actions
Application action
aaa-061467

Figure 4. Boot process on MCX E24x devices

The key used to authenticate the boot code is BOOT_MAC_KEY. It is assumed that BOOT_MAC_KEY is already programmed. If not, the device aborts the secure boot process and clears the FCSESTAT[SB](==0) bit.

Once the secure boot is configured, on every reset, the autonomous secure boot runs in the program flash block starting at address 0 and finishes at BOOT_SIZE number of bits. During this process, the MAC is calculated on this portion of code and it is compared against the BOOT_MAC stored in the secure flash. If the BOOT_MAC slot is empty, the CSEC stores the calculated BOOT_MAC automatically (Section 4.4.3.2) and aborts the secure boot process, setting the FCSESTAT[BIN] (==1) bit.

If the comparison of the calculated MAC value and the BOOT_MAC_KEY is successful, the FCSESTAT[BOK] bit is set (==1). The end of the secure boot process is followed by executing the CMD_BOOT_OK command by the application. This sets the FCSESTAT[BFN](==1) bit to mark the end of the secure boot process.

If the secure boot process is successful and CMD_BOOT_OK is executed, the keys marked as boot protected (BOOT_PROT) can be used by the application code. Otherwise, the boot-protected keys remain locked for the application use.

Note: *A maximum of 512 kB of code can be authenticated using the automated secure boot process. However, the circle of trust method can be followed to authenticate the entire application code (>512 kB) and use a smaller firmware as the bootloader. This bootloader completes the secure boot through the CSEC and the bootloader performs a secure verification of the remaining firmware.*

4.4.3 Adding BOOT_MAC to secure flash (first time)

You can program the BOOT_MAC in two ways:

1. Manually
2. Automatically using CSEC

Both are explained in the sections below.

Note: *This assumes that the CSEC is already enabled with the necessary keys programmed.*

4.4.3.1 Manually

1. Program the code flash with the code to be authenticated.
2. Program BOOT_MAC_KEY into the secure flash using the procedure shown in Section 4.2.1. You can program other user keys at this time too.
3. Define the secure boot mode/flavor and BOOT_SIZE using the CMD_BOOT_DEFINE command.
4. Calculate the MAC for the binary records of the initial BOOT_SIZE portion of the code to be protected using BOOT_MAC_KEY. You can do it offline or using the RAM_KEY feature of the CSEC.
   - In this method, you must use an external program. Put a binary image of the application to a program which can calculate a new BOOT_MAC value using BOOT_MAC_KEY.
   - Load BOOT_MAC_KEY into the RAM key slot by issuing the CMD_LOAD_PLAIN_KEY command. Use the CMD_GENERATE_MAC command to derive the new BOOT_MAC.
     Note: *During BOOT_MAC calculation, an additional 128 bits of data is appended before the binary of the code that must be protected. The new Message_Length = BOOT_SIZE + 128 and the format for DATA for CMAC calculation is as follows:*
     - $DATA = \text{“}0\ldots0\text{”}96 \,|\, BOOT\_SIZE\_value \,|\, PFLASH\_DATA$
     - *BOOT_SIZE_value must be equal to BOOT_SIZE, as defined in the CMD_BOOT_DEFINE command. It occupies 32 bits.*
     - *PFLASH_DATA is the application code that must be protected, starting from address 0x00000000.*
5. Load the calculated MAC at the BOOT_MAC location using the procedure shown in Section 4.2.1.
6. Reset the device. The CSEC confirms the previously stored BOOT_MAC and sets FCSESTAT[BOK]=1 (secure boot OK bit).

4.4.3.2 Automatically using EdgeLock Accelerator (CSEC)

Devices from the factory have no user keys stored in the secure flash. The CSEC calculates and stores BOOT_MAC in the secure flash if the following sequence is followed:

1. Program the code flash with the code to be authenticated.

2. Program BOOT_MAC_KEY into the secure flash using the procedure shown in Section 4.2.1. You can program other user keys at this time too.
3. Define the secure boot flavor and BOOT_SIZE using the CMD_BOOT_DEFINE command.
4. Reset the device. The CSEC calculates BOOT_MAC and stores it in the secure memory slot.
5. Reset the device again. The CSEC confirms the previously calculated BOOT_MAC and sets FCSESTAT[BOK]=1 (secure boot OK bit).

Example code:

See Example-4: Secure Boot example to learn how to set up the secure boot mode (flavor/type) and how to add BOOT_MAC automatically.

Run example 1 to configure the CSEC and load the necessary keys.

The secure boot status is in the Flash CSEC Status Register (FCSESTAT) and it can be interpreted as shown in Table 8.

Table 8. Boot/MAC map to CSEC status flags of FCSESTAT register

Scenario	SB	BIN	BFN	BOK	Error code	Description
No secure boot	0	0	0	0	0	The secure boot never executed.
BOOT_MAC is empty	1	1	1	0	NO_ERR	The secure boot process calculates BOOT_MAC, loads it inside the BOOT_MAC slot, and exits.
BOOT_MAC mismatched	1	0	1	0	NO_ERR	The secure boot process completes with failure.
BOOT_MAC matched	1	0	0	1	NO_ERR	The secure boot process executed successfully. The user application must run the BOOT_OK command to set the BFN bit and enable access to the BOOT_PROT keys.
BOOT_MAC_KEY is empty	0	0	1	0	NO_SECURE_BOOT	The secure boot process exits without success and with an error code.

4.4.4 Updating code and resulting BOOT_MAC

During software development and at other times during an MCU's life cycle, you may have to change the program code flash, which is authenticated by the secure boot process. When this occurs, the BOOT_MAC calculated by the CSEC does not match the BOOT_MAC stored in the secure flash. In this scenario, the cryptographic services that use the keys marked as boot-protected are unavailable. The BOOT_MAC stored in the secure flash must be updated to avoid this situation. There are two scenarios that lead to different methods for updating the stored BOOT_MAC.

4.4.4.1 Scenario 1: no key is write protected and all user keys can be erased and reprogrammed

In this case, you can use the CMD_DBG_CHAL and CMD_DBG_AUTH commands to set the secure flash back to its factory state.

See Section 4.5. The CMD_DBG_CHAL and CMD_DBG_AUTH commands work only on a device that has no keys marked as write protected. All keys are erased by this process. Therefore, the keys must be known to restore them to their previous values. After successfully running the CMD_DBG_CHAL and CMD_DBG_AUTH commands, the user keys section of the secure flash is erased and the device is restored to the factory state.

New keys can be programmed into the secure flash according to Section 4.1 and Section 4.2. Follow the procedure to generate BOOT_MAC, as described in Section 4.4.3.

4.4.4.2 Scenario 2: one or more keys is write protected and all user keys cannot be erased (or not all user keys are known)

In this case, you cannot use the CMD_DBG_CHAL and CMD_DBG_AUTH commands to set the secure flash back to its factory state. To update BOOT_MAC, derive and update a new value, as described in Section 4.2.2. There are two methods that can be used to derive the new BOOT_MAC. They are described in the following sections.

- **Method 1: Use the RAM key and CSEC to generate the new BOOT_MAC**
- **Method 2: Generate the new BOOT_MAC offline**
  – These procedures are the same as those described in Section 4.4.3.

4.5 Resetting flash to the factory state

Note: *The device cannot be reset to the factory state if the keys are write protected.*

CSEC supports a mechanism for resetting the secure flash to the state it was in when it left the factory. The mechanism is only applicable if no user keys are write protected.

CSEC has implemented this mechanism by two commands (CMD_DEBUG_CHAL and CMD_DEBUG_AUTH).

1. PRNG must be initialized before issuing the CMD_DEBUG_CHAL command. PRNG is initialized by executing the CMD_INIT_RNG command and used in deriving a challenge value.
2. Issue the CMD_DBG_CHAL command to request a random number (CHALLENGE - 128 bits).
3. Issue the CMD_DBG_AUTH command to return the authorization parameter (AUTHORIZATION - 128 bits). It can be calculated as follows:

   $$K = KDF\left(KEY_{MASTER\_ECU\_KEY},\ DEBUG\_KEY\_C\right).$$

   - See Section 8 for the KDF.
   - $KEY_{MASTER\_ECU\_KEY}$ is the MASTER_ECU_KEY value.
   - KEY_UPDATE_MAC_C is a constant value defined by HIS-SHE as

   ```
   0x01035348 45008000 00000000 000000B0
   ```

   $$\mathrm{AUTHORIZATION} = \mathrm{CMAC}_K\left(\mathrm{CHALLENGE} \mid \mathrm{UID}\right).$$

   - CMAC is performed over the CHALLENGE, concatenated with the UID using key-K.
4. Reset the device.
   The CMD_DBG_CHAL command must be followed by the CMD_DBG_AUTH command. Otherwise, the CMD_DBG_CHAL command is required to be reissued before continuing.

Successfully issuing these commands results in the following:

1. The device has no user keys (MASTER_ECU_KEY, BOOT_MAC, BOOT_MAC_KEY, KEY1..KEY10 are all erased).
2. The FlexRAM is reset to the traditional RAM functionality (FCNFG[RAMRDY] == 1).
3. The FlexNVM is reset to all Data Flash (FCNFG[EEERDY] == 0).

Note: *The above changes can be reflected on reset only.*

Example code:

See Example-5: Resetting Flash to the Factory State example.

5 Performance numbers

Table 9 shows the nominal execution time for different commands on CSEC.

The setup is as follows: The M4 core runs at 80 MHz and the flash runs at 26.67 MHz. All execution times are measured to process 112 bytes of data.

Table 9. CSEC command execution time

Command	Execution time (to process 112 bytes of data) (in ms)
GENERATE_MAC	0.022925
VERIFY_MAC	0.036301
GENERATE_MAC (pointer method)	0.023488
VERIFY_MAC (pointer method)	0.023488
ENC_ECB	0.018423
DEC_ECB	0.018800
ENC_CBC	0.028718
DEC_CBC	0.029153

Table 10 shows the secure boot execution time for both sequential and parallel secure boot. The sequential and strict sequential boot numbers are the same. The sequential secure boot performance is independent of the IPG clock because the FTFC block has its own asynchronous clock source of 50 MHz, which drives FTFC in case of the sequential boot. While in the parallel boot, FTFC gets its clock from the MCU clock source.

Table 10. CSEC command execution time

Boot flavor	Boot size (in kB)	Clock	Boot time (in ms)
Sequential	128	25-MHz IPG CLOCK	5.09
Sequential	128	12.5-MHz IPG CLOCK	5.09
Parallel	32	LPBOOT=1 i.e. 48-MHz	2.06
Parallel	128	LPBOOT=1 i.e. 48-MHz	8.33
Parallel	32	LPBOOT=0 i.e. 24-MHz	4.17
Parallel	128	LPBOOT=0 i.e. 24-MHz	16.60

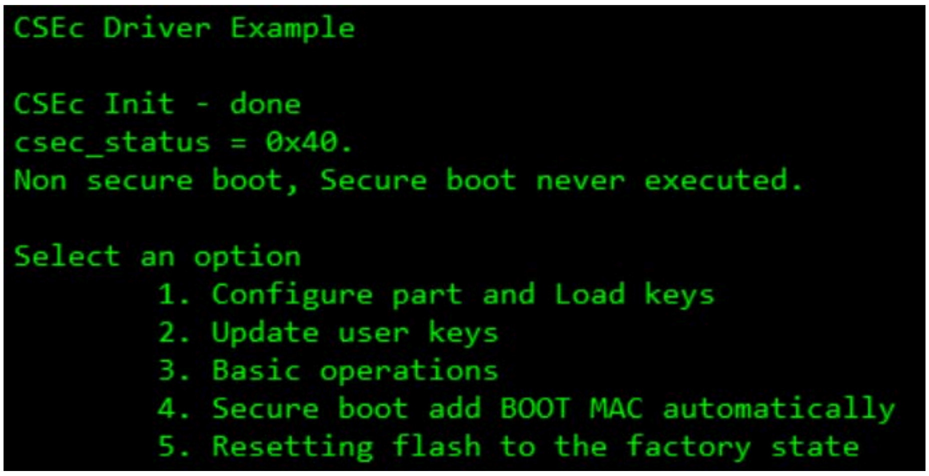
6 Examples

To get started with CSEC, this application note provides the example code developed in the MCUXpresso IDE and tested on the MCXE247 MCU. The example code is available as a separate download with this application note. These are only code examples and they are not intended to use for production.

Warning: *The keys used in the example are for demonstration purposes only. Do not use them in any other scenario.*

You can download the source code from the NXP application code hub: https://github.com/nxp-appcodehub/an-mcxe24x-csec-getting-started.

After downloading the source code, import the project to the MCUXpresso IDE v24.12 or later. Compile and download the code to the FRDM-MCXE247 board. This demo requires interaction through a serial terminal. The default settings of the terminal are as follows: 115200 baud rate, 8 data bits, 1 stop bit, no parity, no flow control.



```
CSEc Driver Example

CSEc Init - done
csec_status = 0x40.
Non secure boot, Secure boot never executed.

Select an option
        1. Configure part and Load keys
        2. Update user keys
        3. Basic operations
        4. Secure boot add BOOT MAC automatically
        5. Resetting flash to the factory state
```

Figure 5. Example

The following is the list of examples:

1. Configure Part and Load Keys
2. Update User Keys
3. Basic Operations
4. Secure Boot Add BOOT MAC Manual
5. Resetting Flash to the Factory State

7 Conclusion

This application note describes the CSEC security module, which exemplifies the low-cost solution to the security needs with a realization of high security standards specified by the HIS-SHE and GM-SHE+ specifications.

This application note illustrates the CSEC operations using software examples, enabling you to jumpstart your security designs.

8 Appendix A Generating M1 to M5

- Generate K1 and K2 as follows:

  $$\mathrm{K1} = \mathrm{KDF}\left(\mathrm{KEY}_{\mathrm{AuthID}},\ \mathrm{KEY\_UPDATE\_ENC\_C}\right)$$

  $$\mathrm{K2} = \mathrm{KDF}\left(\mathrm{KEY}_{\mathrm{AuthID}},\ \mathrm{KEY\_UPDATE\_MAC\_C}\right)$$

  – KDF is a key derivation function, which derives a secret key (K1) from a secret value. $\mathrm{KEY}_{\mathrm{AuthID}}$ is the authorizing key value. See Table 7 for the valid authorizing keys. When a part has a key that is not yet programmed, the initial values of the key are all 1s. The constant value of KEY_UPDATE_ENC_C is defined by HIS-SHE as follows:

  ```
  0x01015348_45008000_00000000_000000B0
  ```

  – The constant value of KEY_UPDATE_MAC_C is defined by HIS-SHE as follows:

  ```
  0x01025348_45008000_00000000_000000B0
  ```

- Generate KDF as follows:

  $$KDF\ (K,\ constant) = AES\text{-}MP(K \mid constant)$$

  – AES-MP is the Miyaguchi-Preneel compression function.

  - The | symbol indicates the concatenation of values.
- Generate M1 as follows:

  $M1 = UID' | ID | AuthID$
  - UID' - UID of the part. It is 120 bits long. It can be 0 (wildcard value) for parts from the factory (because WILDCARD == 0) or (WILDCARD == 0) for the keys that are not wildcard protected.
  - ID - KeyID of the key being updated. ID is four bits long. Do not consider the KBS field value in the ID.
  - AuthID - It can be either the KeyID (the ID of the key being updated) or the MASTER_ECU_KEY KeyID (=0x1). It is four bits long. Do not consider the KBS value in AuthID.
  - The total length of M1 is 128 bits.
- Generate M2 as follows:

  If the Verify_Only flag is disabled (SFE==0x00):

  $M2 = ENC_{CBC,\ K1,\ IV=0}\left(CID' | FID' | \text{“}0...0\text{”}95 | KEY_{ID}'\right)$

  If the Verify_Only flag is enabled (SFE==0x01):

  $M2 = ENC_{CBC,\ K1,\ IV=0}\left(CID' | FID' | \text{“}0...0\text{”}94 | KEY_{ID}'\right)$
  - Run an AES-128 CBC encryption using key K1 (as defined previously) with Initial Value (IV) = 0.
  - CID’ is the new counter value (28 bits). It starts from 0x0000001.
  - FID’ are the new protection flags.

    For SFE == 0x00:

    ```
    WRITE_PROT | BOOT_PROT | DEBUG_PROT | KEY_USAGE | WILD_CARD (5 bits)
    ```

    For SFE == 0x01:

    ```
    WRITE_PROT | BOOT_PROT | DEBUG_PROT | KEY_USAGE | WILD_CARD | VERIFY_ONLY (6
     bits)
    ```
  - 95 (SFE == 0x00) or 94 (SFE == 0x01) zeros to fill the first 128-bit block with zeros.
  - KEY_{ID} is the new key value (128 bits).
  - The total length of M2 is 256 bits.
- Generate M3 as follows:

  $M3 = CMAC_{K2}\left(M1 | M2\right)$
  - A CMAC is performed over M1 concatenated with M2 using key K2.
  - The total length of M3 is 128 bits.

When the CMD_LOAD_KEY command is issued, CSEC derives M4 and M5. These values can be independently generated offline or using CSEC resources and compared against those generated by the CSE.

- Generate K3 and K4 as follows:

  $K3 = KDF\left(KEY_{ID},\ KEY\_UPDATE\_ENC\_C\right)$

  $K4 = KDF\left(KEY_{ID},\ KEY\_UPDATE\_MAC\_C\right)$
  - KEY_{ID} - the value of the key being updated.
  - KEY_UPDATE_ENC_C - the constant value defined by HIS-SHE as follows:

    ```
    0x01015348_45008000_00000000_000000B0
    ```
  - KEY_UPDATE_MAC_C - the constant value defined by HIS-SHE as follows:

    ```
    0x01025348_45008000_00000000_000000B0
    ```
- Generate M4 as follows:

  $M4 = UID | ID | AuthID | M4^{*}$
  - UID - the unique ID of a part (120 bits).

  - ID - the $\mathrm{KEY_{ID}}$ of the key being updated. Do not consider the KBS field (four bits).
  - AuthID - the KeyID of the key authorizing the update. Do not consider the KBS field (four bits).
  - $\mathrm{M4}^{*}$ - the encrypted counter value.
    $\mathrm{M4}^{*} = \mathrm{ENC_{ECB,\ K3}}\left(\mathrm{CID'}\left(28\ \mathrm{bits}\right)\middle|\text{“1”}\left(1\mathrm{bit}\right)\middle|\text{“0}\ldots0\text{”}\left(99\ \mathrm{bits}\right)\right)$ - Run an AES-128 ECB encryption using key K3.
  - The total length of M4 is 256 bits.
- Generate M5 as follows:
  $M5\ =\ CMAC_{K4}\left(M4\right)$
  - A CMAC is performed over M4 using key K4.
  - The total length of M5 is 128 bits.

If M4 and M5 match to what was calculated offline and CSEC returns NO_ERROR in the CSE_ECR (error code register), then the CMD_LOAD_KEY command was successful.

Note: *If a key has its write protection (WRITE_PROTECT) attribute set, the key cannot ever be updated or erased. Use the write protection only when you are absolutely certain that the key never must be changed or erased. Setting the write protection on any single key means that the part cannot be reset to its factory state using the DEBUG CHALLENGE/AUTHORIZATION sequence. See Section 4.5.*

9 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

10 Revision history

Table 11. Revision history

Document ID	Release date	Description
AN14729 v.1.0	22 July 2025	• Initial version

Legal information

Contents

Date of release: 22 July 2025
Document identifier: AN14729