NXP Semiconductors
Application Note

Document Number: AN5401
Rev. 1, 03/2018

Getting Started with CSEc Security Module

CSEc – Cryptographic Service Engine Compressed

by: NXP Semiconductors

1 Introduction

This Application note explains features and functionalities offered by the new hardware security module: “CSEc – Cryptographic Service Engine compressed”. The CSEc security module has been implemented in NXP’s S32K1xx series of devices. This module is completely compatible with the functional specifications of Hersteller Initiative Software-Secure Hardware Extension (HIS-SHE) and GM-SHE+ standards. New CSEc features include:

- Secured Sequential, Strict Sequential and Parallel Boot mode support
- “VERIFY_ONLY” functionality, limiting key usage to CMAC verification only.

The application note lists several security use cases, provides an overview of the CSEc module, describes how to program the CSEc, how to protect 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 CSEc in the form of the S32 Design Studio examples. However it is not within the scope of this application note to discuss the details of the HIS-SHE or GM-SHE+ specification. This document mainly focuses on the hardware features provided by the CSEc module and it is implied that the user is acquainted with the content of the HIS-SHE and GM-SHE+ specification.

Why is Cryptography needed?

The automotive market is rapidly advancing towards the age of autonomous driving and connected vehicles. These technological advances require automotive electronic systems to be connected to the internet and other communication media, such as V2X infrastructure, to a greater extent than ever before. As a result, automotive electronic systems are becoming more susceptible to hacking efforts which attempt to gain unauthorized access to vehicle data or control systems. To prevent unauthorized access, every part of the automotive electronic system needs to be secured: from small microcontrollers managing small tasks to larger gateway processors controlling complex systems, and from application software to data stored in memory. All of these automotive systems are important to the safety and security of the driver and passenger(s) of a vehicle. Embedded security modules and cryptographic engines provide effective tools for ensuring automotive safety and security by enabling secure information exchange and data authenticity and integrity.

Some vital security use-cases include following:

- Immobilizer
- Component protection
- Secure flash updates/ Secure Over The Air(OTA) updates

Contents

- Dataset protection (e.g. mileage)
- Feature management via Digital-Rights- Management(DRM)
- Secure Communication
- IP Protection
- V-to-X Communication (V – Vehicle & X – Vehicle or Infrastructure) and counting…..

1.1 CSEc security module features

Major CSEc security module features are as below:

- Meets the requirements of HIS- SHE specification 1.1 rev 439 and GM-SHE+ security specification
- More Secure cryptographic key storage (ranging from 3 to 17 user keys)
- AES-128 encryption and decryption
- AES-128 CMAC (Cipher-based Message Authentication Code) generation and verification
- ECB (Electronic Cypher Book) Mode - encryption and decryption
- CBC (Cipher Block Chaining) Mode - encryption and decryption
- True & Pseudo random 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 CSEc is not intended to be used to encrypt the code flash content.

2 Security use cases

This section describes automotive 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 prior to execution

2.1 Secure Mileage

In the past, the mileage of cars was illegally reduced to increase the resale value of the vehicle. This essentially created a negative impact on the OEM's (Original Equipment Manufacturer) reputation and increased quality and warranty issues. For this reason the OEM has a strong interest in preventing any illegal manipulation of the mileage. The CSEc can help protect the mileage data. The principle idea is that the mileage is stored encrypted in a non-volatile flash area. Initially, the encoded mileage is read from the flash memory (for example, EEPROM emulation). Before the value is used, it has to be decrypted by the CSEc and whenever the mileage is stored periodically back into the non-volatile memory the CSEc has to encode the mileage value again. This encoding and decoding will only work if the CSEc has previously verified the application code without any failures.

2.2 Immobilizers

Today, immobilizers are standard equipment in every modern car. They prevent cars being stolen without the car key. Additionally, the reduction of the overall number of stolen vehicles has a positive effect on insurance premiums. A simple immobilizer implementation could look like this: The car key includes a transponder, a small cipher unit and a unique cryptographic key. The immobilizer unit sends a random value, generated by the CSEc, to the car key. The car key encrypts this value with the internal AES engine and sends the result back to the immobilizer. The immobilizer has the same secret key stored in the CSEc and is able to decrypt back the random value. Now, the immobilizer code is able to verify that the answer from the car key is correct and start the engine.

2.3 Component Protection

Component protection prevents dismantling a single ECU from a car and re-using it in other ones. Often cars are stolen specifically to re-sell the single components into the aftermarket. OEMs can now address several issues with a secure component protection scheme. Firstly, they can reduce the number of stolen cars; secondly, they can prevent any negative impact on reputation and quality and thirdly, they can protect their own aftermarket business. In a component protection system based on the CSEc, t he most valuable ECUs will include a controller which has a CSEc module. A master node which may be assigned by design or dynamically with a specific algorithm will poll all ECUs of the component protection system and request a specific answer (let say the unique ID) encoded by the CSEc. In this case only CSEc enabled ECUs with the right secret key will be able to send back a valid response. Additionally, the master node can crosscheck the unique ID with a database of all assembled modules in this specific car. This component check can be done periodically while the car is used. If the system detects an unauthorized ECU in the car network, it is able to react on it.

2.4 Flash programming/firmware updates

The CSEc supports secure flash programming by the means of cipher based message authentication code (CMAC) calculation. The application code will verify each block of the new flash image by re-calculating the CMAC value and comparing it with the offline pre-calculated value which is part of the flash image. This check will only be verified when the same secret key was used for the CMAC calculation. This use case is presented and discussed in NXP's application note AN4235 – Using CSE to protect your application via a circle of trust. The same approach can be applied with the CSEc module.

2.5 Secure communication

Car electronic systems are now open to communicate with external devices such as cell phones, other vehicles and other infrastructures. Hence, these systems are becoming 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 car system can be prevented from malfunctioning. The secure communication can be realized when the central ECU needs to communicate with any other ECU. The Central ECU sends the random number generated by its CSEc engine to the other ECU. The other ECU receives the random number, encrypts it along with the data and sends the encrypted message back to the central ECU. The central ECU decrypts the received message and retrieves the data and random number. central ECU also compares the received random number to the already generated random number to verify the correctness. Hence, random number protects against reply attacks and encryption protects against eavesdropping; the whole mechanism ensures data integrity and authenticity.

3 CSEc: An Overview

The main functionality of the CSEc is implemented in the core of the Flash Memory Module (FTFC). Features have been added to the FTFC module to support HIS-SHE functional specification version 1.1. By using an embedded processor, firmware and a hardware assisted AES-128 sub-block, the FTFC module enables encryption, decryption and CMAC generation-verification algorithms for secure messaging applications. Additional APIs are also available for Secure Boot configuration, True Random Number Generation (TRNG) and Miyaguchi-Preneel compression. Figure 1 shows the high level block diagram of the FTFC module. 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 is opaque to the user. 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 the figure 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).

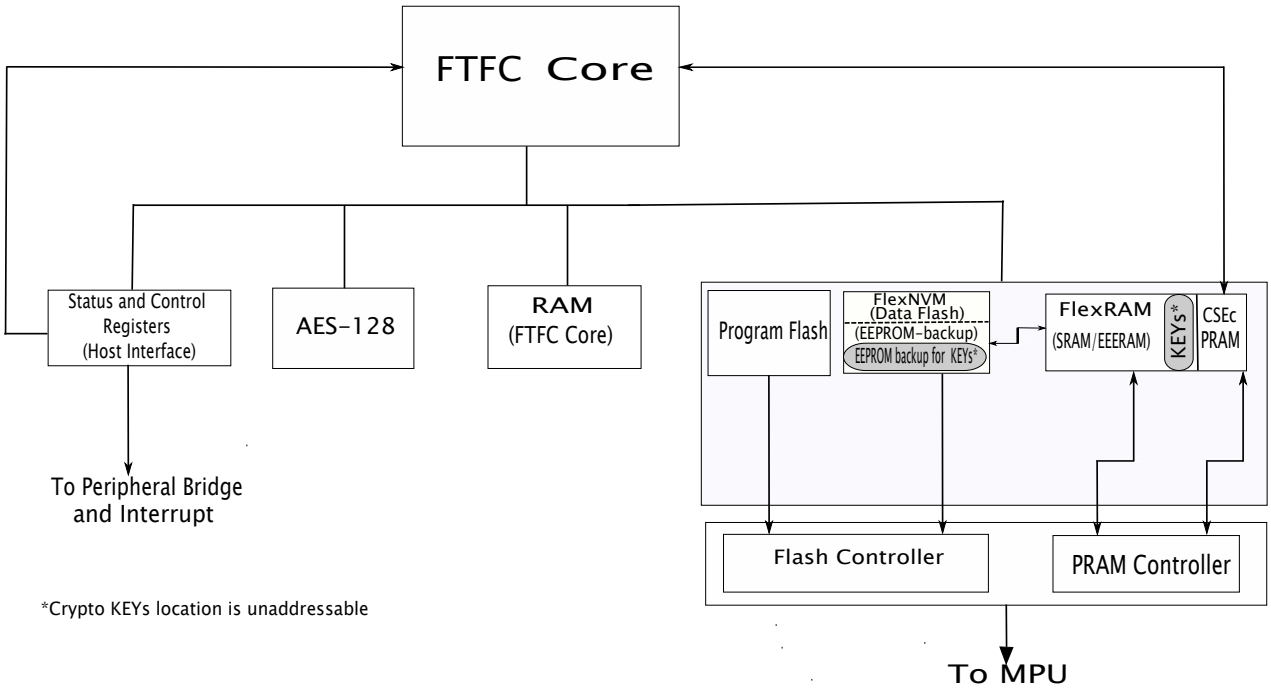

Figure 1. FTFC block diagram with the CSEc

Before going into the detail of the CSEc, we'll 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 non-volatile memory which can be ‘flexed’ between ‘normal flash’ operation and ‘emulated-EEPROM’ operation. During ‘normal flash’ operations it can be used to store application code or data. During ‘emulated-EEPROM’ operations it can be used as non-volatile backup memory for emulated-EEPROM data.

Similarly, **FlexRAM** is the flexible RAM(volatile) and can be ‘flexed’ between ‘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, FlexNVM and FlexRAM together emulate the EEPROM storage. In 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 the user wants to write to the emulated-EEPROM, the user writes to the EEERAM and internally, the flash system locks the interface and writes data back to the EEPROM-backup for non-volatile update. Similarly 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, the user can configure the FTFC module for emulated-EEPROM operation by issuing a Program Partition Command (PRGPART). The Flash Common Command Objects (CCOB) registers are used to issue the PRGPART command. The PRGPART command also provides flexibility to specify the partition size between emulated-EEPROM operation and normal operations as per user’s wish.

Now let’s get back to the CSEc operations.

To enable CSEc functionality, the device must be configured for emulated-EEPROM operation. The PRGPART command is used to enable CSEc and 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 un-addressable (so as corresponding EEPROM –backup). This storage is secured and utilized to store cryptographic keys. Further, this storage is not accessible by any other masters from the system. The secure storage areas are shown as grey sections in figure1.

As per the HIS-SHE specifications, some features are disabled upon any type of debugger connection to the device. (See the HIS-SHE document for detail).

Once the user configures the FTFC module for CSEc functionality and loads the user keys for the security operations, the device is ready for any security related operations as described in the HIS-SHE and GM-SHE+ Specifications.

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 cryptographic operation and once the operation is completed then transferring the results back to the system memory. All data block sizes are 128 bit and need to be padded by the application if the block size is less than 128 bit. Although, 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, EEERAM and CSEc PRAM are locked. Further, no other commands may be initiated until 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, it is also possible to generate an interrupt on completion of a CSEc command. Moreover, any error occurring during CSEc command execution is reported in the Error Bits position of CSEc PRAM.

NOTE

1. No CCOB or CSEc commands are available when the part is in VLPR (Very Low Power) and HSRUN (High Speed Run) modes.
   - As CCOB and CSEc command operations can only be performed when device is in RUN mode. If user desire to switch to any other mode (HSRUN mode to operate at higher frequency for example or VLPR mode to decrease the power consumption value) it is recommended that, prior to switching to any power mode, all command operations must be completed first (by polling for FSTAT[CCIF] flag). In case the device is running in HSRUN/VLPR mode and an CCOB/CSEc operation is needed, user must switch to RUN mode to perform the write and after it finishes, user can switch back to HSRUN/ VLPR mode.
2. It is not possible to execute CCOB commands (Program, Erase etc.) 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 continuation of an ongoing CSEc command, but this results in cancellation of the ongoing CSEc command.
4. Execution of a CSEc command while in Erase Suspend (ERSSUSP) state will result in the suspended erase operation being aborted (not able to be resumed).
5. Starting execution of CCOB or CSEc commands locks out the CCOB interface, EEERAM and CSEc PRAM. These are unlocked on completion of those commands.

3.1 Cryptographic Keys

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

Tables 1 and 2 summarize available key types and their properties such as ID, memory type, size, attributes etc. All keys and their properties are described throughout this section.

Table 1. User 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	Wildcard	Verify Only	
Secret_Key	X	0x0	ROM	16	-	-	√	√	-	-	-	Wtritten by NXP
UID	X	0x0	ROM	15	-	-	-	-	-	-	-	Wtritten by NXP
MASTER_ECE_KEY	X	0x1	Non-Volatile	16	28	√	√	√		√		Empty
BOOT_MAC_KEY	X	0x2	Non-Volatile	16	28	√		√		√		Empty
BOOT_MAC	X	0x3	Non-Volatile	16	28	√		√		√		Empty
KEY_01-KEY_10	1'b0	0x4–0xD	Non-Volatile	16	28	√	√	√	√	√	√	Empty
KEY_11-KEY_17	1'b1	0x4–0xA	Non-Volatile	16	28	√	√	√	√	√	√	Empty

Table continues on the next page...

Table 1. User 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'b1	0xE	-	16	-	-	-	-	-	-	-	-
Reserved	1'b0	0xE	-	-	-	-	-	-	-	-	-	-
RAM_KEY*	X	0xF	Volatile	16	-	-	-	-	-	-	-	Undefined after every reset

√ indicates available attributes to the key.

*RAM_KEY has only PLAIN_KEY flag and is only implemented for RAM_KEY. It is set by CSEc when key is loaded into RAM_KEY slot as a plain text.

Table 2. Other Volatile Keys

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

The FTFC module provides secure, non-volatile storage for cryptographic keys as described in the HIS-SHE functional specification. The first five slots have a dedicated use, the remaining slots are available for application specific keys.

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

NOTE

Both the SECRET_KEY and the UID are un-addressable and unalterable.

- MASTER_ECU_KEY – It is used to reset the CSEc to factory state or to change any other keys.
- BOOT_MAC_KEY – It is used by secure boot process to verify the authenticity of the software.
- BOOT_MAC – This 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 detailed description
- KEY_01 to KEY_17 – The user keys are stored in the EEERAM space, with a configurable amount of space for user keys. Anywhere from 3 to 17 user keys may be configured using the program partition command
- RAM_KEY – This is a volatile key and 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.

- PRNG_KEY and PRNG_STATE – These are not directly accessible by any user functions and are internally used by pseudo random number generator.

NOTE

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

3.1.1 Key ID: {KBS, Key IDx}

Each key has an identification number associated with it called KeyID. This number is used to identify the key being used, updated or authorizing the update. Since the CSEc has more keys than specified by the HIS-SHE, Key Block Select (KBS) is used to select the bank of the key. Using KBS and KeyID together one can index any key from 1 to 17.

The keys_1-10 are in bank-0 and keys_11-17 are in bank-1.

e.g. 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 later chapters and appendix A.) when a key is being updated.

3.1.3 Key Attributes

Each key has 6 flags associated with it. These flags determine how and under what conditions that key can be used. These value are included while deriving M2. (See Appendix A)

Write Protection Flag (WRITE_PROT)

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

NOTE

This flag should be set with caution. Setting this flag is an irreversible step and will prevent the device from being reset to factory state.

Boot Protection Flag (BOOT_PROT)

If set, the key cannot be used if the MAC value calculated in the SECURE_BOOT step did not match the BOOT_MAC value stored in secure Flash. In other words, if secure boot fails, the keys marked BOOT_PROT remain locked and they cannot be used during 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.

Wildcard Protection Flag (WILDCARD)

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

Verify Only Flag (VERIFY_ONLY)

This flag is introduced to comply with the GM-SHE+ specification. This functionality can be used if enabled. It can be enabled during PGMPART configuration settings (SFE = 0x01).

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 affect.

3.2 Generic 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 is used to issue the CSEc commands and pass data to the CSEc interface for security operations. The CSEc PRAM is organized into eight 128-bit wide 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																
0	FuncI D	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 message control length (Page 0, Word 3). The rest of the pages are utilized for input/output data information.

Writing to the command header triggers the macro to lock the CSEc PRAM interface and start the CSEc operation. Hence, to setup a CSEc command, the user should first enter data information followed by message length information and must write command header last.

As soon as the command header is written, the CSEc starts execution by resetting 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, on completion (i.e. FSTAT[CCIF] == 1) the user may read required data back from the CSEc PRAM location.

There can be a case where data cannot fit into the CSEc PRAM. In this case, the CSEc requires repeating the same command with new data. To continue the same command, the user must write the remaining data in CSEc PRAM as applicable, followed by updated Command Header information. In the Command Header, only the "CallSeq" field needs to be changed and other information must remain the same. Changing the FuncID field, while continuing the previous command will result in a sequence error. Hence, the FuncID field must stay consistent throughout the continuation of a previous command.

Sections 3.2.1 and 3.2.2 describe how to enter data and command information into the CSEc PRAM interface.

3.2.1 Writing Data and message length information in PRAM interface

The data and message length information can be written by the usual means at the CSEc PRAM locations specified by 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 6 bytes as shown in figure 2.

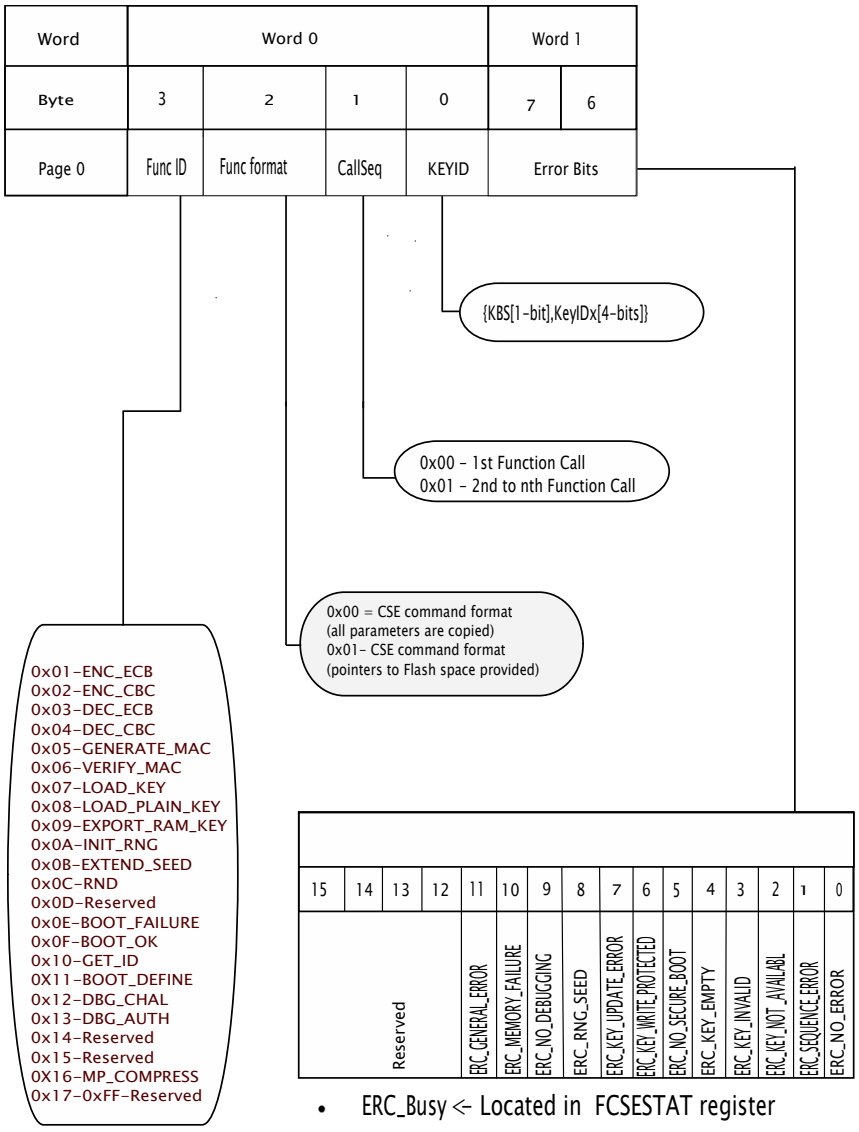

Figure 2. CSEc Command Header

FuncID: Function Identification (ID) field is 8-bits long and specifies the security command to be executed according to the HIS-SHE command definition. Valid values: 0x0 to 0x16.

FuncFormat: 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 and the command function. Here, the main core or DMA will first copy the data and then issue the function call.

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

Valid Values:

Copy method: 0x00

Pointer method: 0x01

CallSeq: Call Sequence specifies whether the information is the first or a following function call. When the amount of data used for a command exceeds the size of the PRAM, CallSeq can be used to specify the continuation of the command. For example if the user has more than seven 128-bit blocks of data for CMAC verification, this field can be used to indicate continuation of data during subsequent call.

Valid Values:

1st Function Call: 0x00

2nd through 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. And the KBS (Key Block Select – not used in all commands) is a 1-bit value allowing the user to switch between key banks. It is described in Cryptographic Keys on page 6.

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 will result in addressing incorrect key. If KBS is not used for particular key then user must write 0 to bit[4].

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

3.2.3 CSEc Status, Error and Interrupt Reporting

The CSEc clears the FSTAT[CCIF] flag once the operation is started and sets the FSTAT[CCIF] flag as soon as the operation completes. This flag can be used to generate an interrupt by setting 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. IDB and EDB bits indicates whether internal and external debugging features are enabled. RIN bit is set upon initialization of random number generator. While BSY is set when CSEc specific command processing is ongoing. BOK, BFN, BIN and SB bits are secure boot specific bits and is described in Automatically using CSEc on page 21. For more information refer to the device Reference Manual.

Table 5. FCSESTAT Register

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

4 Programming the CSEc Security Module

To program the device for security operations, the following the steps shown in the figure 3 need to be followed:

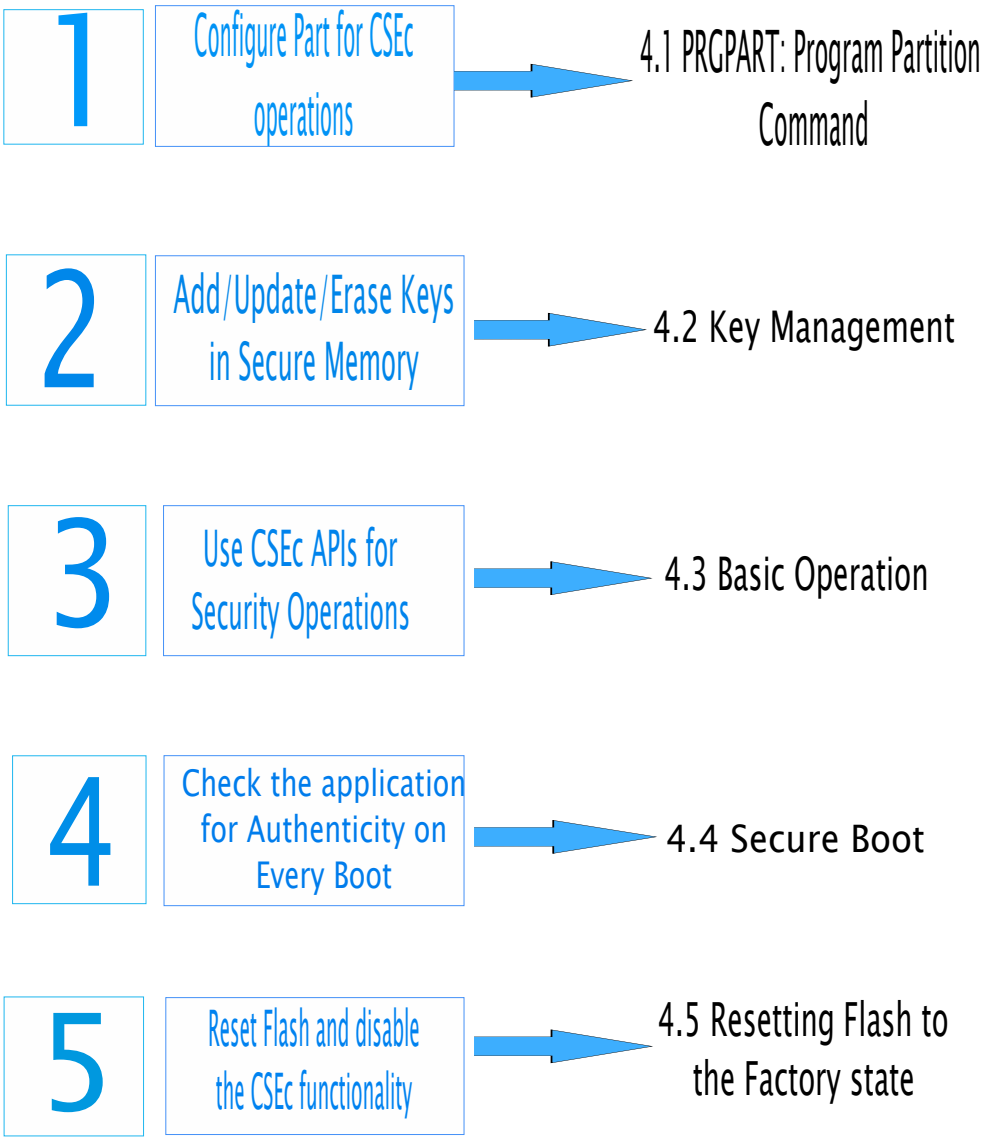

Figure 3. The CSEc programming directions

4.1 PRGPART: Program Partition Command

To start using the CSEc command set, the user must configure the device into emulated-EEPROM mode. The PGMPART command defines the CSEc Key Size and FlexNVM/FlexRAM partition between normal and EEPROM operation. It also defines whether VERIFY_ONLY functionality is enabled and whether 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 refer to the Program Partition Command Section of the device reference manual.

NOTE

- The Emulated EEPROM-backup size should be at least 16 times the EEERAM.
- Prior to 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 Erase All Blocks command in Device Reference manual).
- It is highly recommended that the user reviews the “Program Partition command” section in the device reference manual.

Example Code:

Example-1: Configure Part and Load Keys, demonstrates “how to issue PGMPART command”. This example is developed in the S32 Design Studio IDE and runs on an NXP EVB (S32K144EVB-Q100) , in RUN mode, at 48MHz FIRC. This example is developed to run from RAM as it updates the system or secure flash. It is available as a separate download with this application note. The “Load Keys” part of this example is explained in section 4.2.1: Adding keys to secure memory slots. This example configures EEPROM to the following settings: (See table along with the code below)

- Key size = 1 to 20
- VERIFY_ONLY functionality = Disabled
- FlexRAM(EEERAM) status = it will be loaded with valid data on reset
- EEPROM Partition: EEERAM = 4kB and EEPROM-backup = 16 x 4 = 64kB

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

Key specific size will be subtracted from total emulated-EEPROM available for application use. e.g. in case of 20 keys, 3.5kB (4k-512) of emulated-EEPROM will be 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

Once the device is configured successfully for CSEc operation, the FCNFG register fields are set to the following.

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

NOTE

- EEERAM = 4 kB is the only option available with S32K1xx devices. It can vary with different devices from the S32K1xx family.
- Refer application note AN11983: Using the S32K1xx EEPROM Functionality, for detailed understanding of emulated-EEPROM feature.

```
while((FTFE-> FSTAT & FTFE_FSTAT_CCIF_MASK) != FTFE_FSTAT_CCIF_MASK); /* Wait until any
ongoing flash operation is completed */ FTFC-> FSTAT = (FTFC_FSTAT_FPVIOL_MASK |
FTFC_FSTAT_ACCERR_MASK); /* Write 1 to clear error flags
```

```
FSTAT = (FTFC_FSTAT_FPVIOL_MASK | FTFC_FSTAT_ACCERR_MASK); /* Write 1 to clear error flags
*/ FTFE-> FCCOB [3] = 0x80; /* FCCOB0 = 0x80, program partition command */
FTFE-> FCCOB [2] = 0x03; /* FCCOB1 = 2b11, 20 keys *
/ FTFE-> FCCOB [1] = 0x00; /* FCCOB2 = 0x00, SFE = 0, VERIFY_ONLY attribute functionality
disable*
/ FTFE-> FCCOB [0] = 0x00; /* FCCOB3 = 0x00, FlexRAM will be loaded with valid EEPROM data
during reset sequence */
FTFE-> FCCOB [7] = 0x02; /* FCCOB4 = 0x02, 4k EEPROM Data Set Size*/
FTFE-> FCCOB [6] = 0x04; /* FCCOB5 = 0x04, no data flash, 64k(all , 16*4k = 64k ) EEPROM
backup */

FTFE-> FSTAT = FTFE_FSTAT_CCIF_MASK; /* Start command execution by writing 1 to clear CCIF
bit*/

while((FTFE-> FSTAT & FTFE_FSTAT_CCIF_MASK) != FTFE_FSTAT_CCIF_MASK); /* Wait until ongoing
flash operation is completed */

flash_error_status = FTFE-> FSTAT ; /* Read the flash status register for any Execution
Error */
```

The figure below shows Flash memory map before and after partitioning on S32K144 with:

- CSEc enabled with 20 keys
- Highest endurance i.e. all data Flash is used as EEPROM backup

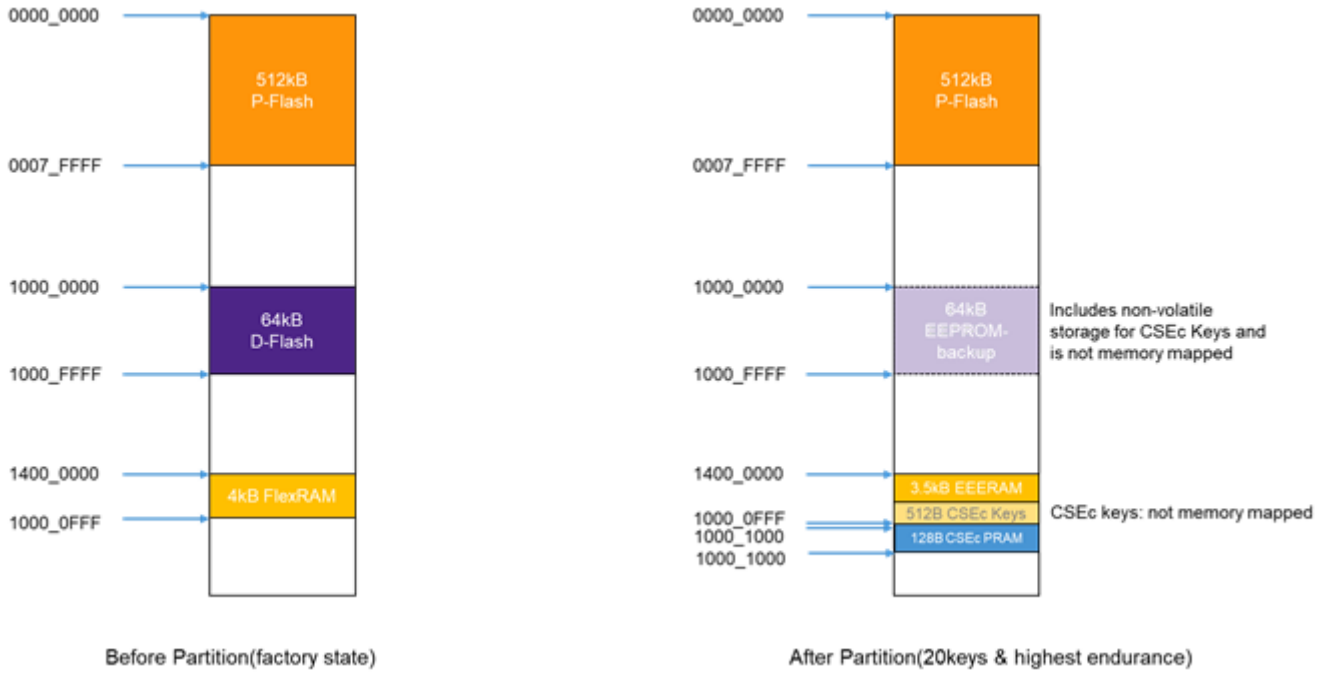

Figure 4. Flash memory map before and after partitioning on S32K144

4.2 Key Management

Key management includes 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, the protocol defined in the HIS-SHE specification must be used (HIS-SHE Functional Specification, v1.1, Section 9.1 Description of memory update protocol). This ensures confidentiality, integrity, authenticity and protects against replay attacks. HIS-SHE requires that in order to update the memory containing the keys the following must be calculated and passed to the CSEc:

- M1 = UID'|ID|AuthID – 128 bits
- M2 = $ENC_{CBC, K1, IV=0}$ (CID'|FID'|"0...0"95|KID') – 256 bits : SFE==0x00

  M2 = $ENC_{CBC, K1, IV=0}$(CID'|FID'|"0...0"94|KID') – 256 bits : SFE==0x01
- M3 = $CMAC_{K2}$ (M1|M2) – 128 bits

These values will typically be derived on an offline computer and created as arrays in a header file. The values can be also derived using the target part (S32K1xx) and CSEc. CSEc PRAM pages 1 to 4 are loaded with M1, M2 and M3. Then, CSE_LOAD_KEY command header shall be written (on page 1) to start command execution. After the execution is completed, the CSEc PRAM pages 5 to 7 will return M4 and M5.

- M4 = UID|ID|AuthID|M4* - 256 bits
- M5 = $CMAC_{K4}$ (M4) – 128 bits

These values can be verified against the pre-calculated offline values for flawless addition of the key.

NOTE

See Appendix A for detailed explanation of M1 to M5.

NOTE

- If the key to be updated is not Wildcard protected (WILDCARD == 0), UID=0 may be used in the generation of M1 and M3. Otherwise the device UID will need to be established and used in the generation of M1 and M3. The UID can be established as described in section 4.3.2, UID Retrieval.
- 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 will be required in all other cases.

Example Code:

Example-1 Configure Part and Load Keys. This example is developed in the S32 Design Studio IDE and runs on an NXP EVB (S32K144EVB-Q100) , in RUN mode, at 48MHz FIRC. This example is developed to run from RAM as it updates the system or secure flash. It is available as a separate download with this application note.

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 M5 during run-time 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, it is no longer possible to update that key.

After a device's keys are programmed into the secure flash and the device is no longer in its factory state, it may be necessary to update one or more keys. HIS-SHE describes a mechanism for doing this and this has been implemented in the CSEc module via the CMD_LOAD_KEY command.

4.2.2.1 Authorization:

In order to keep keys secure, HIS-SHE requires that an authorizing key be known before an update to a specific key can be attempted. In general, knowledge of a specific key is needed in order to update that specific key. MASTER_ECU_KEY is a key with special meaning and can be used to authorize update of all keys (BOOT_MAC_KEY, BOOT_MAC and all KEY_1 to KEY_17) without knowledge of those keys. See Table 6. Flash Common Command Objects registers (FCCOB) requirements for Program Partition command on page 13 of the HIS-SHE specification below.

Table 7. Keys must be known to update key

Key to Update	Keys which must be known to update Key				
	MASTER_ECU_KEY	**BOOT_MAC_KEY**	**BOOT_MAC**	**KEY_<N>**	**RAM_KEY**
MASTER_ECU_KEY	√				
BOOT_MAC_KEY	√	√			
BOOT_MAC	√	√			
KEY_<N>	√			√	
RAM_KEY[1]				√	

1. RAM_KEY can also be updated by simply supplying new key in plain text format.

- To update a particular key knowledge of any one key is sufficient. For example, to update BOOT_MAC_KEY, knowledge of either MASTER_ECU_KEY or BOOT_MAC_KEY is sufficient.

- 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, the user needs to increase the counter value associated with that key. The process for updating a given key is the same as that described in Adding keys to secure memory slots on page 14.

Example Code:

Example-2 Update User Keys. This Example updates the already programmed user key. First run Example-1, that programs the user keys and then run this program to update them. This example is available as a separate download with this application note. This program uses the MASTER_ECU_KEY as authorizing key and updates KEY_1 and KEY_11 to the new value. This code calculates M1 to M5 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. But it is possible to erase all keys together. The procedure for erasing all keys is described in Resetting Flash to the Factory State on page 23 .

NOTE

All keys must be erased to be able to issue flash mass erase.

4.3 Basic Operations

Now after getting familiar with the CSEc architecture and feature set and learning how to add and update keys, this section describes a few basic operations, such as UID retrieval, AES-128 encoding and decoding, CMAC generation and verification and random number generation.

4.3.1 Random number generation:

The PRNG has a 128-bit state variable and uses AES in output feedback mode to generate pseudo random values. A key derived from the SECRET_KEY is used for the PRNG. The RND command updates the state of the PRNG and returns the 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. If the seed is run through compression before the seed is used in the PRNG, then FCSESTAT[RIN] will be set to indicate RNG is initialized. The PRNG uses the PRNG_STATE/KEY and seed per HIS-SHE specification and the AIS20 standard. Run CMD_RND to generate a random number.

NOTE

PRNG must be initialized before issuing the following commands: CMD_EXTEND_SEED, CMD_RND and CMD_DBG_CHAL

Example Code for generating Random number:

See Example-3 Basic Operations to learn how to initialize PRNG and generate the random number. It is available as a separate download with this application note.

4.3.2 UID Retrieval:

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

The UID can be obtained by issuing the CSE_GET_ID command.

NOTE

If MASTER_ECU_KEY is empty then the UID_MAC return value will be set to ‘0’.

Example Code for retrieving UID from secure flash:

See Example-3 Basic Operations. It is available as a separate download with this application note.

4.3.3 AES-128 Encoding and Decoding:

The CSEc supports AES-128 encryption and decryption in ECB (Electronic Codebook) and CBC (Cipher Block Chaining) modes of operation. The key is selected from one of the memory slots which must be enabled for the encryption (KEY_USAGE = 0, Section 3.1.2 Key Attributes).

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

Since the command takes length in terms of PAGE_LENGTH, 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 Encoding and Decoding:

See Example-3 Basic Operations, to learn how to encode and decode plain text using Cypher Block Chaining (CBC). It is available as a separate download with this application note. Run this lab from flash.

4.3.4 CMAC generation and verification:

The CSEc uses the AES-128 CMAC algorithm for message authentication. The key for the CMAC operation is selected from one of the memory slots. The key must be enabled for authentication (KEY_USAGE =1; Section 3.1.3 Key Attributes).

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

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

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

Example Code for CMAC generation and verification:

See Example-3 Basic Operations, to learn how to generate the CMAC for the given data and then how to verify it with existing CMAC and data. It is available as a separate download with this application note.

4.4 Secure Boot

The CSEc has a mechanism which allows users to authenticate boot code in flash. The MCU can be configured 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

S32K1xx devices support three secure boot modes.

NOTE

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

A. Sequential Boot Mode: In this mode, after RESET, the flash system comes out of RESET and the core stays in RESET or can execute from ROM code. The secure boot process verifies the application firmware block. If secure boot is successful then the keys will be made available for security tasks, otherwise keys marked as boot protected will be blocked for all tasks. Lastly, the core starts executing application firmware.

B. Strict Sequential Boot Mode: In this mode, after RESET, the flash system comes out of RESET and the core stays in RESET or can execute from ROM code. The secure boot process verifies the application firmware block. If secure boot is successful then the keys will be made available for security tasks. Otherwise, if CMAC comparison fails then main core stays in RESET (hence no application firmware is executed) or may execute ROM code.

NOTE

This boot mode is irreversible; that means once set, boot mode cannot be changed to the other boot modes.

Before setting this mode BOOT_MAC must be calculated and stored, otherwise the device will remain in RESET. (Automatic BOOT_MAC calculation does not run in this mode.)

C. Parallel Boot Mode: In this mode, after RESET, the flash system and the main core come out of RESET and main core starts executing application firmware. In parallel to the main core execution, CSEc verifies the application firmware block using the secure boot process. If secure boot is successful then the keys will be made available for security tasks, otherwise keys marked as boot protected will be blocked for all tasks. 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 that needs to be authenticated.

Figure and the description below illustrate the secure boot flow in S32K1xx devices.

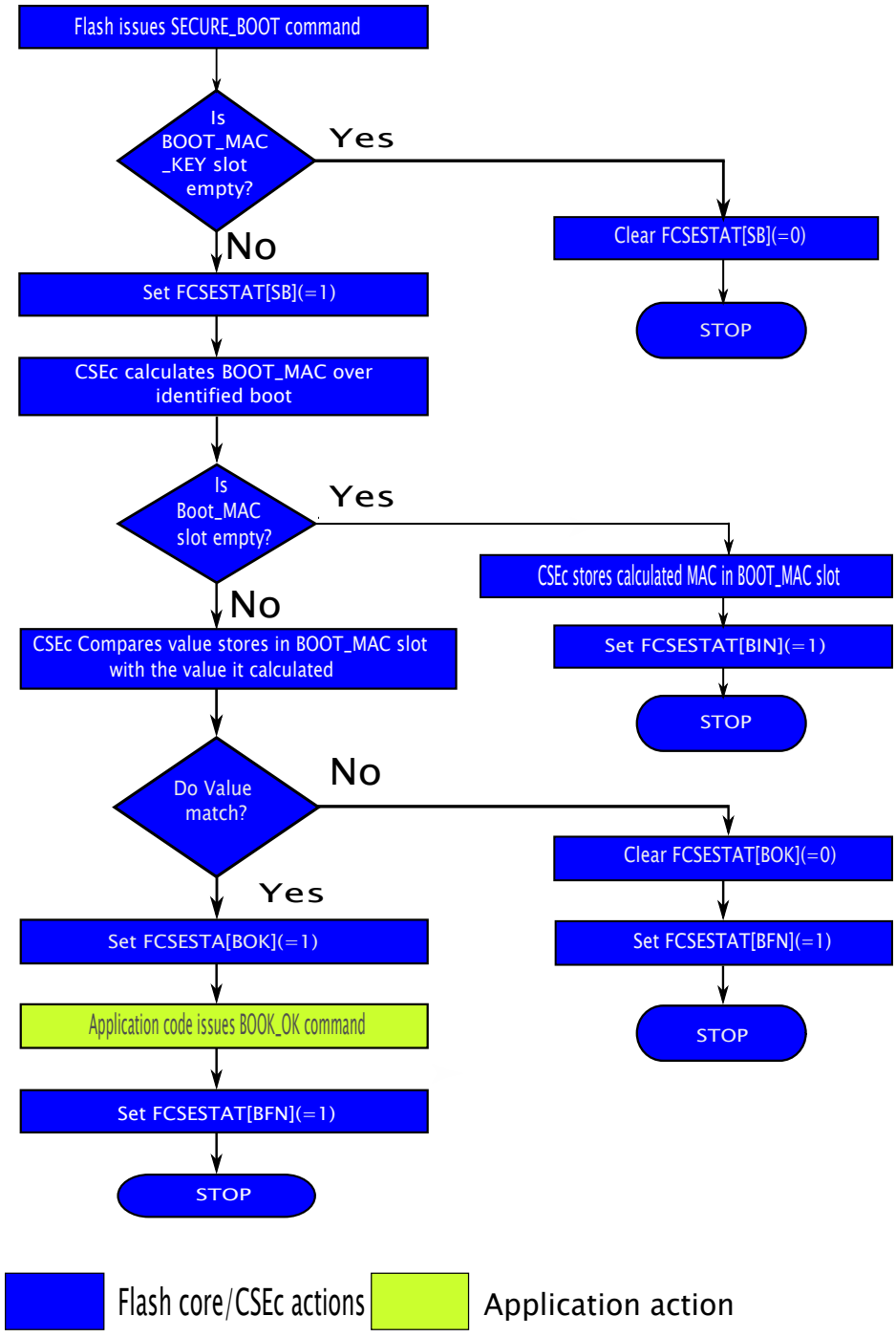

Figure 5. CSEc Boot Process on S32K1xx devices

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

Once secure boot is configured, on every reset the autonomous secure boot runs on 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 the code and compared against the BOOT_MAC stored in secure flash. If BOOT_MAC slot is empty then the CSEc stores the calculated BOOT_MAC automatically (Section 4.4.3.2) and aborts the secure boot process, setting FCSESTAT[BIN] (==1) bit

If the comparison of the calculated MAC value and the BOOT_MAC_KEY is successful then FCSESTAT[BOK] bit is set (==1). The end of the secure boot process is followed by executing CMD_BOOT_OK command by the application. This sets 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, keys marked as Boot Protected (BOOT_PROT) can be used by the application code. Otherwise boot protected keys remain locked for 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 as described in an application note, AN4235: Using CSE to protect your application via a circle of trust, can be followed to authenticate the entire application code (>512 kB).

4.4.3 Adding BOOT_MAC to secure flash (first time)

The BOOT_MAC can be programmed in two ways:

1. Manually
2. Automatically using CSEc

Both are explained in this section below.

NOTE

This assumes that the CSEc is already enabled with necessary keys programmed.

4.4.3.1 Manually

1. Program the code flash with the code to be protected.
2. Program BOOT_MAC_KEY into secure flash (other user keys may be programmed at this time too)
3. Define the secure boot mode/flavor and the 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. It can be done offline or using the RAM_KEY feature of the CSEc.
   - In this method an external program must be used. A binary image of the application should be input to a program which can calculate a new BOOT_MAC value using the BOOT_MAC_KEY. *Using RAM_KEY feature of the CSEc:*
   - Load the BOOT_MAC_KEY into the RAM key slot by issuing the CMD_LOAD_PLAIN_KEY command. Then use the CMD_GENERATE_MAC command to derive the new BOOT_MAC.

     NOTE

     During BOOT_MAC calculation the additional 128 bit data is appended before the binary of the code that needs to be protected. The new Message_Length = BOOT_SIZE + 128 and format for DATA for CMAC calculation is as follows:

     DATA = “0…0”96 | BOOT_SIZE_value | PFLASH_DATA

     - BOOT_SIZE_value should be equal to the BOOT_SIZE as defined in CMD_BOOT_DEFINE command. It occupies the 32 bits.
     - PFLASH_DATA is the application code that needs to be protected starting from address 0x00000000
5. Load calculated MAC at the BOOT_MAC location using the procedure of section 4.2.1, Adding Keys to secure memory slots.

6. Reset the device. The CSEc confirms previously stored BOOT_MAC and sets FCSESTAT[BOK]=1 (Secure Boot OK bit).

4.4.3.2 Automatically using CSEc

Devices from the factory have no user keys stored in the secure flash. The CSEc will calculate and store BOOT_MAC in secure flash if the following sequence is followed:

1. Program the code flash with code to be protected
2. Program BOOT_MAC_KEY into secure flash (other user keys may be programmed at this time too)
3. Define the secure boot flavor and the BOOT_SIZE using CMD_BOOT_DEFINE command.
4. Reset the device. The CSEc calculates BOOT_MAC and stores it in secure memory slot.
5. Reset the device again; The CSEc confirms previously calculated BOOT_MAC and sets FCSESTAT[BOK]=1 (Secure Boot OK bit)

Example Code:

See Example-4 Secure_boot_add_BOOT_MAC, to learn how to set up secure boot mode(flavor/type) and how to add BOOT_MAC manually and automatically. It is available as a separate download with this application note.

Run example 1 first to configure CSEc and load necessary keys.

For manual setup, execute all steps in the program.

For automatic setup, execute only till step 3.

The secure boot status can be seen in the Flash CSEc Status Register (FCSESTAT) and can be interpreted as shown in the table below.

Table 8. Boot/MAC map to CSEc Status Flags of FCSESTAT register 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	Secure boot never executed
BOOT_MAC is Empty	1	1	1	0	NO_ERR	Secure boot process calculates BOOT_MAC and loads it inside the BOOT_MAC slot and exits without success
BOOT_MAC Mismatched	1	0	1	0	NO_ERR	Secure boot process completes with failure.

Table continues on the next page...

Table 8. Boot/MAC map to CSEc Status Flags of FCSESTAT register Boot/MAC map to CSEc Status Flags of FCSESTAT register (continued)

Scenario	SB	BIN	BFN	BOK	Error Code	Description
BOOT_MAC Matched	1	0	0	1	NO_ERR	Secure boot process executed successfully. User application needs to run BOOT_OK command to set BFN bit and enable access to BOOT_PROT keys.
BOOT_MAC_KEY is Empty	0	0	1	0	NO_SECURE_BOOT	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 ECU's life cycle, it may be necessary to change the program code flash which is authenticated by the secure boot process. When this occurs, the BOOT_MAC calculated by the CSEc will not match the BOOT_MAC stored in the secure flash. In this scenario, cryptographic services which used keys marked as Boot Protected will be unavailable. The BOOT_MAC stored in secure flash must be updated to avoid this situation. There are 2 scenarios which 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 re-programmed

In this case the CMD_DBG_CHAL and CMD_DBG_AUTH commands can be used to set the secure flash back to its factory state. Resetting Flash to the Factory State on page 23. The CMD_DBG_CHAL and CMD_DBG_AUTH commands will only work on a device which has no keys marked as Write Protected. All keys will be erased by this process, therefore the keys must be known in order 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 will be erased and the device will be restored to the factory state. New keys can be programmed into the secure flash following steps in Section 4.1, and 4.2. Then, the procedure to generate BOOT_MAC should be followed as described in Adding BOOT_MAC to secure flash (first time) on page 20.

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, the CMD_DBG_CHAL and CMD_DBG_AUTH commands cannot be used to set the secure flash back to its factory state. In order to update the BOOT_MAC, a new value must be derived and updated as described in Updating key: on page 15.There are 2 methods which can be used to derive the new BOOT_MAC. These 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**
  - Both of these procedures are the same as described in Adding BOOT_MAC to secure flash (first time) on page 20-**Manually** on page 20

4.5 Resetting Flash to the Factory State

NOTE: Device cannot be reset to factory state if any of the keys are write protected.

HIS-SHE describes 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 have been write protected. The process is described in section 11 of the HIS-SHE spec “failure analysis of SHE/Resetting of SHE”.

The CSEc has implemented this mechanism by way of 2 commands. These are CMD_DEBUG_CHAL and CMD_DEBUG_AUTH.

1. The CMD_DBG_CHAL command is issued to request a random number (let say CHALLENGE – 128-bits).
2. Now, CMD_DBG_AUTH command is issued to return the authorization parameter (let say AUTHORIZATION – 128bits). It can be calculated as below

   K = KDF($KEY_{MASTER\_ECU\_KEY}$, DEBUG_KEY_C)

   - See Appendix A for KDF.
   - $KEY_{MASTER\_ECU\_KEY}$ – MASTER_ECU_KEY value
   - KEY_UPDATE_MAC_C – Constant value defined by HIS-SHE as :

     0x01035348 45008000 00000000 000000B0

   AUTHORIZATION = $CMAC_K$ (CHALLENGE | UID)

   - CMAC is performed over CHALLENGE concatenated with UID using key-K
3. Reset the device.

The CMD_DBG_CHAL command must be followed by the CMD_DBG_AUTH command, otherwise the CMD_DBG_CHAL command would be required to be reissued before continuing.

Successfully issuing these commands will result in:

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

NOTE

Above changes can be reflected on reset only.

The PRNG must be initialized prior to the CMD_DEBUG_CHAL command being issued. The PRNG is initialized by executing the CMD_INIT_RNG command and used in deriving a challenge value.

Example Code:

See Example-5 Resetting flash to the factory state. It is available as a separate download with this application note.

5 Performance Numbers

The table below shows nominal execution time for different commands on CSEc engine.

Setup: M4 Core running at 80 MHz and Flash running at 26.67 MHz. All execution times are measured to process 112bytes 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

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

Table 10. Secure boot execution time

BOOT FLAVOR	BOOT SIZE (in KB)	CLOCK	BOOT TIME (in ms)
Sequential	128	25Mhz IPG CLOCK	5.09
Sequential	128	12.5Mhz IPG CLOCK	5.09
Parallel	32	LPBOOT=1 i.e. 48MHz	2.06
Parallel	128	LPBOOT=1 i.e. 48MHz	8.33
Parallel	32	LPBOOT=0 i.e. 24MHz	4.17
Parallel	128	LPBOOT=0 i.e. 24MHz	16.60

6 Examples

To get started with the CSEc module, this application note covers the example codes developed in NXP's S32 Design Studio IDE for ARM Architecture products and tested on the **S32K144** MCU. The example code is available as a separate download with this application note. Moreover, the user should be aware that these are example codes only and not intended to use for production.

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

The explanations in this application note walk through the CSEc security module, which exemplifies the low cost solution to the security needs with realization of high security standards specified by the HIS-HIS-SHE and GM-SHE+ specifications. Furthermore, this application note illustrated the CSEc operations via example software enabling users to jumpstart their security designs.

Please refer to the NXP Application Note AN4235 – Using CSE to protect your application via a circle of trust, for theoretical explanation on how to protect entire application code in the flash.

8 Glossary

- CSEc – Cryptographic Service Engine – Compressed
- HIS – Hersteller Initiative Software
- SHE – Secure Hardware Extension
- AES – Advanced Encryption Standard
- CMAC – Cipher based message authentication code
- ECB – Electronic Code Book
- CBC – Cipher Block Chaining
- OEM – Original Equipment Manufacturer
- FTFC – Flash Memory Module of S32K1xx devices
- CCOB/ FCCOB – Flash Common Command Object
- PRAM – Parameter space Random Access Memory
- SRAM – System Random Access Memory
- PGMPART – Program Partition Command
- EEEPROM/emulated-EEPROM – Emulated Electrically Erasable Programmable Read-Only Memory
- RNG – Random Number Generator
- PRNG – Pseudo Random Number Generator
- TRNG – True Random Number Generator

9 References

1. HIS-SHE - Secure Hardware Extension functional specification Version1.1 (rev 439) available on https://www.automotive-his.de
2. S32K144 Reference Manual available on http://www.nxp.com/
3. AN4234 – Using the Cryptographic Service Engine (CSE)
4. AN4235 – Using CSE to protect your application via a circle of trust
5. AN11983 - Using the S32K1xx EEPROM Functionality
6. [FIPS197] NIST/FIPS: Announcing the Advanced Encryption Standard (AES); November 26, 2001; http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

10 Revision history

Revision number	Date	Substantive Changes
0	12/2016	Initial release
1	03/2018	• Editorial update throught the document. • Added section Performance Numbers on page 23. • Updated section for better understanding.

A Generating M1 to M5

- **Generate K1 & K2**

  K1 = KDF(KEY_{AuthID}, KEY_UPDATE_ENC_C)

  K2 = KDF(KEY_{AuthID}, KEY_UPDATE_MAC_C)

  — KDF is key derivation function which derives a secret key (K1) from a secret value.

  — KEY_{AuthID} – Authorizing key value (See Table 7 for valid authorizing keys). In the case where a part has that key not yet programmed, the initial value of key is all 1's.

  — KEY_UPDATE_ENC_C – Constant value defined by HIS-SHE as:

  0x01015348_45008000_00000000_000000B0

  — KEY_UPDATE_MAC_C – Constant value defined by HIS-SHE as :

  0x01025348_45008000_00000000_000000B0

- **Generate KDF**

  KDF (K, constant) = AES-MP(K | constant)

  — AES-MP – Miyaguchi-Preneel compression function.

  — | – this symbol indicates "Concatenation of values".

- **Generate M1**

  M1 = UID' | ID | AuthID

  — UID' – UID of the part. It is 120 bit long. It can be 0 (Wildcard value) for parts from the factory (because WILDCARD == 0) or for the keys that are not wildcard protected (WILDCARD == 0).

  — ID – KeyID of the key being updated. ID is 4 bits long. Do not consider KBS field value in ID.

  — AuthID – It can be either KeyID (ID of key being updated) or MASTER_ECU_KEY KeyID (=0x1). It is 4 bits long. Do not consider KBS value in AuthID.

  — Total length of M1 = 128 bits.

- **Generate M2**

  If Verify_Only flag is disabled: SFE==0x00

  $M2 = ENC_{CBC, K1, IV=0}(CID'|FID'|"0...0"95|KEY_{ID}')$

If Verify_Only flag is enabled: SFE==0x01

M2 = $ENC_{CBC, K1, IV=0}$(CID'|FID'|"0...0"94|KEY_{ID}')

- Run a AES-128 CBC encryption using key K1 (as defined previously) with Initial Value (IV) = 0
- CID' – the new counter value (28 bits).Starts from 0x0000001
- FID' – 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 first 128 bit block with zeros
- KEY_{ID}' – The new key value (128 bits)
- Total Length of M2 = 256 bits

- **Generate M3**

  M3 = $CMAC_{K2}$(M1|M2)
  - A CMAC is performed over M1 concatenated with M2 using key K2
  - Total Length of M3 = 128 bits

When the CSE_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.

- **Generating K3 & K4**

  K3 = KDF(KEY_{ID}, KEY_UPDATE_ENC_C)

  K4 = KDF(KEY_{ID}, KEY_UPDATE_MAC_C)
  - KEYID – Value of the key being updated.
  - KEY_UPDATE_ENC_C – Constant value defined by HIS-SHE as:

    0x01015348_45008000_00000000_000000B0
  - KEY_UPDATE_MAC_C – Constant value defined by HIS-SHE as :

    0x01025348_45008000_00000000_000000B0
- **Generate M4**

  M4 = UID|ID|AuthID|M4*
  - UID – Unique ID of part (120 bits)
  - ID – KeyID of key being updated. Do not consider KBS field. (4 bits)
  - AuthID – KeyID of key authorizing the update. Do not consider KBS field. (4 bits)
  - M4* - the encrypted counter value M4* = $ENC_{ECB, K3}$(CID'(28 bits) | "1"(1bit) | "0…0"(99 bits)) - Run a AES-128 ECB encryption using key K3
  - Total Length of M4 = 256 bits
- **Generate M5**

  M5 = $CMAC_{K4}$(M4)
  - A CMAC is performed over M4 using key K4
  - Total Length of M5 = 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. Write Protection should only be used when the user is absolutely certain that the key never needs to be changed or erased. Setting Write Protection on any single key will mean that the part cannot be reset to its factory state using the DEBUG CHALLENGE / AUTHORIZATION sequence. See section 4.5, Resetting Flash to the Factory State.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Document Number: AN5401
Rev. 1, 03/2018