Document Title	Specification of Secure Hardware Extensions
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	948

Document Status	published
Part of AUTOSAR Standard	Foundation
Part of Standard Release	R23-11

Document Change History

Date	Release	Changed by	Description
2023-11-23	R23-11	AUTOSAR Release Management	• No content changes
2022-11-24	R22-11	AUTOSAR Release Management	• No content changes
2021-11-25	R21-11	AUTOSAR Release Management	• No content changes
2020-11-30	R20-11	AUTOSAR Release Management	• No content changes
2019-11-28	R19-11	AUTOSAR Release Management	• Initial release

Contents

1 Introduction

This technical report represents a republication under AUTOSAR development partnership of HIS *SHE - Functional Specification v1.1, rev 439* specification. Errata and amendments to this specification are published in AUTOSAR_TR_ListOfKnownIssuesSecureHardwareExtensions.pdf document.

2 Definition of terms and acronyms

2.1 Acronyms and abbreviations

Abbreviation / Acronym:	**Description:**
HIS	Hersteller Initiative Software
SHE	Security Hardware Extension
AES	Advanced Encryption Standard
TPM	Trusted Platform Module
CBC	Cipher Block Chaining
ECB	Electronic Code Book
MAC	Message Authentication Code
CMAC	Cipher-based Message Authentication Code
IV	Initialization Vector
UID	Unique IDentification item
TRNG	True Random Number Generator
PRNG	Pseudo Random Number Generator

2.2 Definition of terms

Terms:	**Description:**
Term	

3 Related Documentation

3.1 Input documents & related standards and norms

[1] NIST:Announcing the Advanced Encryption Standard (AES)
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[2] NIST Special Publication 800-38A:Recommendation for Block Cipher Modes of Operation:Methods and Techniques
http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

[3] NIST Special Publication 800-38B:Recommendation for Block Cipher Modes of Operation:The CMAC Mode for Authentication
http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf

[4] NIST:Updated CMAC Examples
http://csrc.nist.gov/publications/nistpubs/800-38B/Updated_CMAC_Examples.pdf

[5] Handbook of Applied Cryptography
http://www.cacr.math.uwaterloo.ca/hac/

[6] Recommendation for Key Derivation Using Pseudorandom Functions (Revised)
https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf

[7] BSI:A proposal for:Functionality classes and evaluation methodology for true (physical)random number generators, Version 3.1
http://www.bsi.bund.de/zertifiz/zert/interpr/trngk31e.pdf

[8] BSI:Application Notes and Interpretation of the Scheme (AIS)
http://www.bsi.bund.de/zertifiz/zert/interpr/ais20e.pdf

[9] Trusted Computing Group
https://www.trustedcomputinggroup.org/

4 Functional specification

4.1 Introduction

The Secure Hardware Extension (SHE) is an on-chip extension to any given microcontroller. It is intended to move the control over cryptographic keys from the software domain into the hardware domain and therefore protect those keys from software attacks. However, it is not meant to replace highly secure solutions like TPM chips or smart cards, i.e. no tamper resistance is required by the specification.

The main goals for the design at hand are

- Protect cryptographic keys from software attacks
- Provide an authentic software environment
- Let the security only depend on the strength of the underlying algorithm and the confidentiality of the keys
- Allow for distributed key ownerships
- Keep the flexibility high and the costs low

Basically SHE consists of three building blocks, a storage area to keep the cryptographic keys and additional corresponding information, a implementation of a block cipher (AES) and a control logic connecting the parts to the CPU of the microcontroller, see Figure 4.1 for a simplified block diagram. SHE can be implemented in several ways, e.g. a finite state machine or a small, dedicated CPU core.

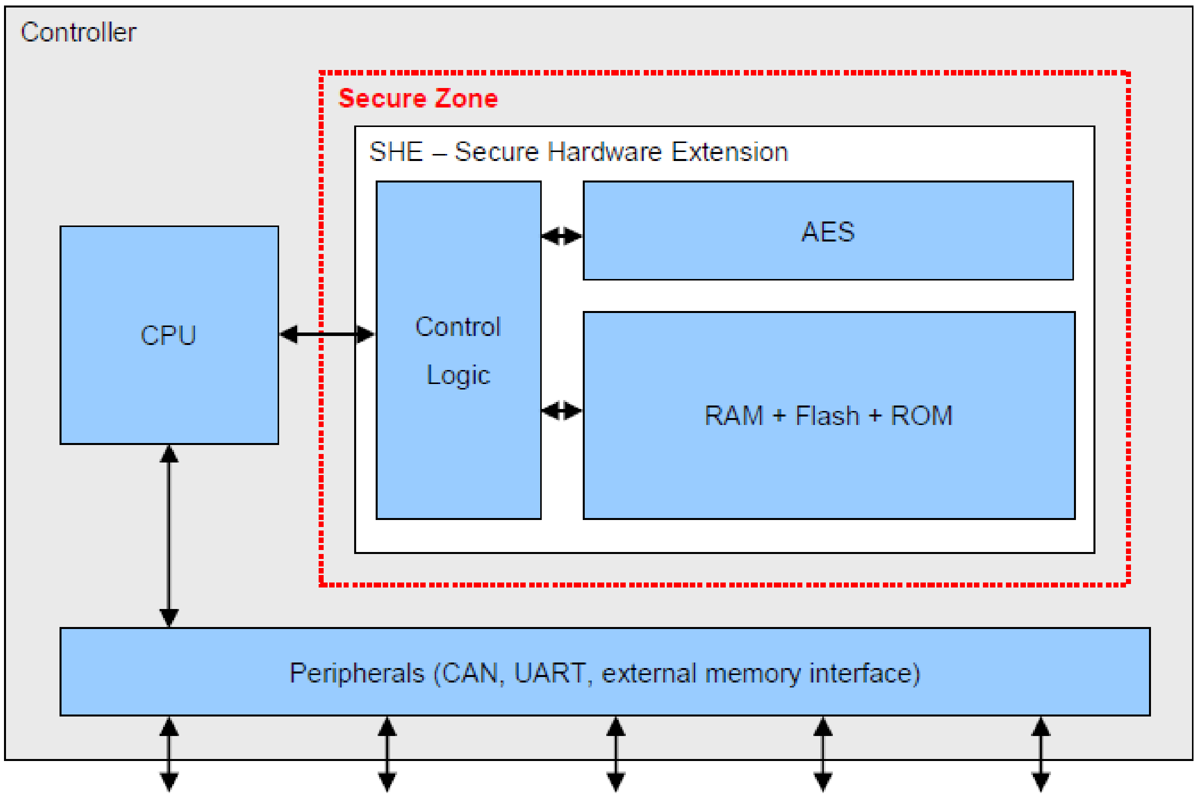
Controller
Secure Zone
SHE – Secure Hardware Extension
CPU
Control Logic
AES
RAM + Flash + ROM
Peripherals (CAN, UART, external memory interface)

Figure 4.1: Simplified logical structure of SHE

This document is intended to provide a detailed description of the technical realization of SHE which will be complemented by a reference implementation. This document does not contain the motivation for every single requirement and neither does it contain concepts how to use SHE in certain applications.

Beware that SHE will not solve all security flaws by simply adding it to a microcontroller. It has to be supported by the application software and processes.

4.1.1 Conventions

In the following chapters several paragraphs are printed italic. These paragraphs are not meant as a hard requirement but to provide additional explanation of the underlying mechanisms.

Throughout the whole document the term "CPU" denotes the actual microprocessor while "microcontroller" is used to describe the hardware complete chip, i.e. a CPU and all included peripherals. The term "control logic" refers to the system (e.g. a finite state machine or a small microprocessor) controlling the algorithms and memories inside of SHE.

Cryptographic operations are written as

$\mathtt{OUTPUT} = \mathtt{OPERATION}_{\mathtt{MODE,KEY[,IV]}}\ \mathtt{(INPUT\ [,\ INPUT2,\ ...])}$

Additionally the following symbols are used to describe the operations:

$\oplus$ bitwise exclusive or
|concatenation of two values

Whenever interfaces to the CPU are described the internal memory slots of SHE, which are not exposed to the CPU, are identified by their address. An actual key is generally written as $KEY_{\texttt{KEY\_NAME}}$ while the identifier of the key is written as $ID_{\texttt{KEY\_NAME}}$.

All values are given in the form MSB...LSB, i.e. the most significant bit/byte is on the left.

Bit sequences given in the form

$"0...0"_{128}$

shall mean a string of bits with the value '0' and length of 128.

4.2 Basic Requirements

SHE has to be realized as an on-chip peripheral of the microcontroller. SHE must not have any other connections except those explicitly specified within this document. If additional resources have to be included to assure proper function during the fabrication of the chip, all ports have to be physically and permanently deactivated if accessible on external pins.

SHE can be connected to the CPU in several ways, e.g. through a dedicated interface or an internal peripheral bus. The interconnection must be implemented in a way that no other peripheral or an external entity can modify the data transferred between the CPU and SHE.

SHE does not need to be fabricated in a special process to increase security nor need any actions to be taken to strengthen the system against physical attacks, e.g. etching the chip casing open, differential power analysis, glitching attacks.

Note: No sophisticated secure hardware mechanisms are required to meet the specification of SHE. However, a manufacturer may of course strengthen the design to provide a higher security level for higher security requirements.

SHE needs to be notified by a status signal whenever internal or external debuggers for debugging software and hardware are attached and active. Examples for debuggers are JTAG, BDM etc.

This document does only describe the technical parts of SHE. Processes and environmental conditions, e.g. for inserting keys, are not subject of this specification.

4.3 Algorithms

All cryptographic operations of SHE are processed by an AES-128 [1]. The latency of the AES must remain <2us per encryption/decryption of a single block, including the key schedule.

Additionally, the performance of the AES must be high enough to allow for a secure boot (see Chapter 4.10 for details) of 5% of the flash memory, but 32kByte at minimum and 128kByte at maximum, of the microcontroller in <10ms.

In case the flash memory is slower than requested, the flash memory must be the limiting factor for the secure boot and the limit has to be stated in the datasheet.

4.3.1 Encryption/decryption

For encryption and decryption of data, SHE has to support the electronic cipher book mode (ECB) for processing single blocks of data and the cipher block chaining mode (CBC) for processing larger amounts of data, see [2] for details.

The latency for both modes may not exceed the value given in the beginning of section 4.3

SHE can only process multiples of the block length of the AES, i.e. all necessary padding has to be done by an application.

The input, output and key input as well as any intermediate results may not be directly accessible by the CPU but access must be granted depending on the policies by the controller logic of SHE.

4.3.2 MAC generation/verification

The MAC generation and verification has to be implemented as a CMAC using the AES-128 as specified by [3]. See [4] for updated examples.

4.3.3 Compression function

The Miyaguchi-Preneel construction (see [5] Algorithm 9.43) with the AES as block cipher is used as compression function within SHE. Messages have to be preprocessed before feeding them to the compression algorithm, i.e. they have to be padded and parsed into 128 bit chunks.

Padding is done by appending one '1' bit to the message M of bit-length l, followed by k '0' bits, where k is the smallest, non-negative solution to the equation $l + 1 + k \equiv 88 \bmod 128$. Finally append a 40 bit block that is equal to the number l expressed using an unsigned binary representation.

Before feeding the padded message to the compression function, it has to be parsed into n 128 bit chunks $x_1, x_2, ..., x_n$. The value OUT_0 is called initialization vector (IV).

$$\texttt{AES-MP}(x_i) : \mathrm{OUT}_i = \mathrm{ENC}_{\mathrm{ECB},\ \mathrm{OUT}_{i-1}}(x_i) \oplus x_i \oplus \mathrm{OUT}_{i-1};\ i > 0;\ \mathrm{OUT}_0 = 0;$$

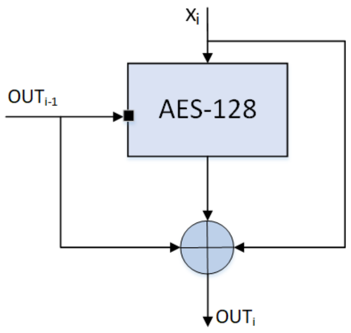

Figure 4.2: Miyaguchi-Preneel one-way compression function

4.3.3.1 Key derivations

Keys are derived using the Miyaguchi-Preneel compression algorithm based on [6]. Derived keys are calculated by compressing the correctly preprocessed concatenation of a secret `K` and a constant `C′` with

```
C′ = 0x01 | counter | "SHE" | 0x00
```

Note that the constants `C` given in section 4.12 have already been padded according to subsection 4.3.3.

```
KDF(K, C): AES-MP(K | C)
```

4.4 Data storage

SHE needs memory to store keys and MACs. A non-volatile memory is required to store information that needs to be available after power cycles and resets of the microcontroller. A volatile memory is required to store temporary information. The volatile memory may loose its contents on reset or power cycles.

The memory of SHE should only be accessible by the SHE control logic. The CPU or any other peripherals, including debugging and testing facilities being available without opening the chip package, should not be able to access the memory.

If data from and to the non-volatile memory of SHE has to be transferred over a bus shared with other peripherals and it could not be guaranteed that these data will not be read or modified by any other instance, the following precautions have to be taken transparently to all functionality of SHE:

- An additional, individual key has to be stored inside of SHE. The key has to meet the same requirements as SECRET_KEY described in Chapter 4.4.4.1
- All data has to be encrypted with the AES in ECB mode using the key described above before being put on the bus or decrypted respectively upon reading
- Write access to the memory must only be possible by SHE, write access to the connection between SHE and its memory must be prevented for other peripherals during read/write access by SHE

The memory must be readable and writeable independently by SHE while the CPU operates on other public/non-public memory blocks of the microcontroller, i.e., time sharing of memory interfaces is allowed but blocking the CPU for a complete SHE operation is not allowed.

The persistent memory of SHE is separated into logical blocks called memory slots. Each has a width of 128 bits plus up to five security bits (see Chapter 4.4.1.1 to Chapter 4.4.1.5 and a saturating, unsigned counter with 2^{28} states.. The segments must be writable/erasable separately, i.e. when changing a memory slot the other memory slots may not be affected. See Table 4.3 in Chapter 4.14 for an overview of which information has to be stored with every key. The information does not necessarily have to be physically stored in this order.

Note: The slot is separated into the actual key, the protection bits and a counter being used to protect the memory slot against replay attacks during update.

Precautions have to be taken to keep the current value of a non-volatile memory slot if a write operation fails due to interruption, e.g. power loss. A write operation must be treated as an atomic operation, in any case the memory must contain either the old value or the new value but no corrupted areas.
The write operation may not be interrupted either.

The initial value of all non-volatile memory slots has to be given in the data sheet and it is referred to as "empty" in the following.

Note: The functions of SHE rely on a detection of empty memory cells. If the underlying technology does not distinguish between erased cells and cells written with the same logical value, the implementation has to introduce another status bit for every memory slot to allow for the detection. The additional status bit has to be handled transparently to the user.

Note: The initial value of the memory slots has to be used in conjunction with the memory update protocol described in Chapter 4.9 to initialize SHE.

The value of the counter for every non-volatile memory slot has to be 0 after production.

At least 100 successful write-cycles to the non-volatile memory must be guaranteed per memory slot by the implementation, more write cycles must be possible.

Table4.4 gives a matrix to show which memory slot is used by which function while Table 4.5 shows which keys can serve as a secret to update another key. Figure 4.3 gives an overview of all keys implemented in SHE.

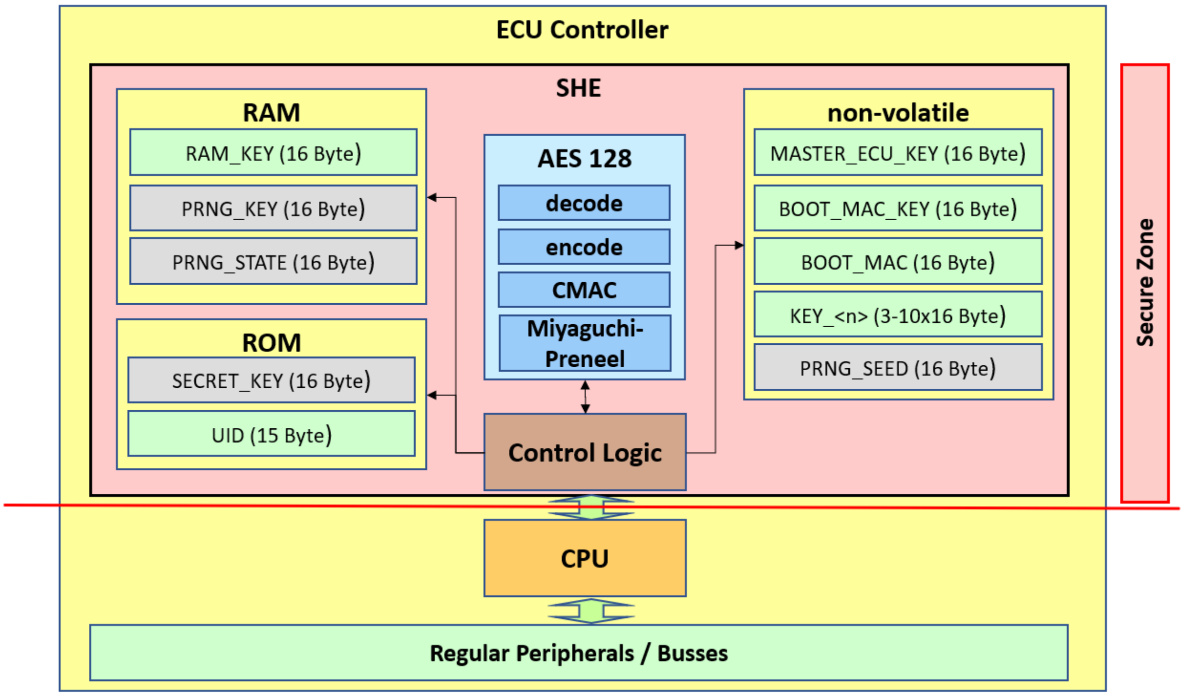

Figure 4.3: Detailed logical structure of SHE

4.4.1 Security flags for memory slots

When flags are transmitted in protocols the value “0" shall mean the flag is not set and “1" shall mean the flag is set. See Table 4.3 for details which key is protected by which security bits.

4.4.1.1 Write-protection of memory slots

Non-volatile keys, see Chapter 4.4.2, can be write-protected, i.e. it is not possible to change the key anymore, even if the corresponding secret is known.

The write-protection must be irreversible.

The write-protection is stored in a non-volatile memory only accessible by SHE and evaluated by the state machine controlling SHE upon write access.

Whenever the flag is transmitted in a protocol and it is not applicable to that particular key, it has to be transmitted as “0" and ignored by SHE.

The flag must not be set for any key after production.

4.4.1.2 Disabling keys on boot failure

Non-volatile keys, see Chapter 4.4.2, can be disabled separately when the secure boot mechanism (see Chapter 4.10) detects a manipulation of the software or the secure boot process is bypassed by other boot mechanisms, e.g., by boot strapping over external interfaces. The memory slots may only be reactivated on next successful boot.

Disabling a key shall mean that SHE refuses to use the memory slot in any operation except those explicitly stated.

The status of this protection is stored in a non-volatile memory area only accessible by SHE and evaluated by the state machine controlling SHE upon read and write access.

Whenever the flag is transmitted in a protocol and it is not applicable to that particular key, it has to be transmitted as “0" and ignored by SHE.

The flag must not be set for any key after production.

4.4.1.3 Disabling keys on debugger activation

Non-volatile keys, see Chapter 4.4.2, can be disabled separately when a debugger is attached (e.g., JTAG, BDM) or any other mechanism is activated to boot without measuring the boot process by SHE while secure booting is activated, e.g. bootstrap over external interfaces. The memory slots may only be reactivated after a reset.

Disabling a key shall mean that SHE refuses to use the memory slot in any operation except those explicitly stated.

The status of the debugging protection is stored in a non-volatile memory area only accessible by SHE and evaluated by the state machine controlling SHE upon read and write access.

Whenever the flag is transmitted in a protocol and it is not applicable to that particular key, it has to be transmitted as “0" and ignored by SHE.

The flag must not be set for any key after production.

4.4.1.4 Disable wildcard usage for key updates

The flag determines if a key may be updated without supplying a valid UID, i.e. by supplying a special wildcard. For details on updating keys see Chapter 4.9 for details on the UID see Chapter 4.4.4.2.

If the flag is set, wildcards are not allowed.

The flag must not be set for any key after production.

4.4.1.5 Key usage determination

The flag determines if a key can be used for encryption/decryption or for MAC generation/verification.

If the flag is set, the key is used for MAC generation/verification.

The flag has only to be implemented for the keys KEY_<n>, see Chapter 4.4.2.4.

Whenever the flag is transmitted in a protocol and it is not applicable to that particular key, it has to be transmitted as “0" and ignored by SHE.

4.4.1.6 Plain key flag

The flag has only to be implemented for RAM_KEY, see Chapter 4.4.3.1. The flag has to be set by SHE whenever a key is loaded into KEY_{RAM_KEY} in plaintext. The flag is evaluated before exporting a RAM_KEY. The flag has to be reset whenever a key is loaded by the secure protocol, see Chapter 4.9.

4.4.2 Non-volatile memory slots

The keys and the corresponding policies are described in detail within the following sections.

Non-volatile memory slots must implement a mechanism to detect empty memory slots, i.e. memory slots that have not been populated with a key after erasing or after production. Empty memory slots may not be used by any function but for inserting a key via the key update protocol described in Chapter 4.9.

4.4.2.1 MASTER_ECU_KEY

Note: The MASTER_ECU_KEY is intended to be populated by the “owner" of the component using SHE and it can be used to reset SHE or change any of the other keys.

The MASTER_ECU_KEY is only used for updating other memory slots inside of SHE, see Chapter 4.9 for details on updating memory slots.

A new MASTER_ECU_KEY can be written with the knowledge of the current MASTER_ECU_KEY and is protected by the common lock mechanisms described in Chapter 4.4.1.1, Chapter 4.4.1.2, Chapter 4.4.1.3 and Chapter 4.4.1.4.

The MASTER_ECU_KEY must be empty after production.

4.4.2.2 BOOT_MAC_KEY

The BOOT_MAC_KEY is used by the secure booting mechanism to verify the authenticity of the software.

The BOOT_MAC_KEY may also be used to verify a MAC.

See Chapter 4.10 for details on the secure booting.

The BOOT_MAC_KEY can be written with the knowledge of the MASTER_ECU_KEY or BOOT_MAC_KEY and is protected by the common lock mechanisms described in Chapter 4.4.1.1, Chapter 4.4.1.2, Chapter 4.4.1.3 and Chapter 4.4.1.4.

Note: When changing the BOOT_MAC_KEY the BOOT_MAC usually should be changed, too, except when first activating secure booting for autonomous MAC learning, see Chapter 4.10.3.

The BOOT_MAC_KEY must be empty after production.

4.4.2.3 BOOT_MAC

Note: The BOOT_MAC is required for the secure boot mechanism and is therefore stored inside of SHE. It is not considered to be a secret information, however, it is treated like any other key inside of SHE for the ease of use.

The BOOT_MAC is used to store the MAC of the Bootloader of the secure booting mechanism and may only be accessible to the booting mechanism of SHE.

See Chapter 4.10 for details on the secure booting.

The BOOT_MAC can be written with the knowledge of the MASTER_ECU_KEY or BOOT_MAC_KEY and is protected by the common lock mechanisms described in Chapter 4.4.1.1, Chapter 4.4.1.2, Chapter 4.4.1.3 and Chapter 4.4.1.4.

The BOOT_MAC must be empty after production.

4.4.2.4 KEY_<n>

Note: The KEY_<n> are actually intended to be used to process bulk data in any given application.

KEY_<n> can be used for arbitrary functions. n is a number 3..10, i.e. SHE must at least implement three and at maximum ten keys for arbitrary use.

The usage of the keys has to be selected between encryption/decryption or MAC generation/verification on programming time by setting the key usage flag accordingly, see Chapter 4.4.1.5.

KEY_<n> can be written with the knowledge of the MASTER_ECU_KEY or the current KEY_<n> and is protected by the common lock mechanisms described in Chapter 4.4.1.1, Chapter 4.4.1.2, Chapter 4.4.1.3 and Chapter 4.4.1.4.

The KEY_<n> must be empty after production.

4.4.2.5 PRNG_SEED

PRNG_SEED is used to store the seed for pseudo random number generator as described in Chapter 4.5.1.1. If the random number generator is implemented as described in Chapter 4.5.1.2 the memory slot is not required.

PRNG_SEED may only be accessed by CMD_INIT_RNG as described in Chapter 4.5.1.1.

PRNG_SEED must be initialized during fabrication of the chip. Its value must be generated by a certified physical random number generator, e.g. a High Security Module (HSM), and should meet at least the requirements of class P2 from [7].

Note: PRNG_SEED needs to be recalculated (see Chapter 4.5.1.1) before random numbers can be requested, i.e. in worst-case scenarios it is written on every power cycle/reset.

4.4.3 Volatile memory slots

The volatile memory slots should be cleared to “0" on reset or power-on-cycles.

The volatile memory slots are not protected by the lock mechanisms specified in Chapter 4.4.1.1, Chapter 4.4.1.2 and Chapter 4.4.1.3

4.4.3.1 RAM_KEY

The RAM_KEY can be used for arbitrary operations.

The RAM_KEY can be written with the knowledge of the KEY_<n> or in plain text.

The RAM_KEY can be exported if it was loaded as plaintext, see CMD_EXPORT_RAM_KEY in Chapter 4.7.9. A key loaded by the secure protocol may not be exported. The origin of the key has to be stored in a flag, see Chapter 4.4.1.6.

Note: Since the keys loaded into RAM_KEY are stored externally they are not under full control of SHE, hence they are vulnerable to several attacks, e.g. replay attacks and denial of service attacks. It is strongly advised to consider this fact when designing applications.

4.4.3.2 PRNG_KEY

The PRNG_KEY is not directly accessible by any user function but is used by the pseudo random number generator. See Chapter 4.5 for details on usage and data population.

The PRNG_KEY may also be implemented in non-volatile memory. In this case the key has to be generated, populated and protected by the same means as SECRET_KEY (see Chapter 4.4.4.1).

4.4.3.3 PRNG_STATE

This memory slot holds the state of the pseudo random number generator.

The PRNG_STATE is not directly accessible by any user function but is used by the pseudo random number generator. See Chapter 4.5 for details on usage and data population.

4.4.4 Read-Only memory slots

Two memory slots are defined read-only during the life-cycle of the controller. Read-only shall mean that they are at least protected by the controller logic, i.e. they may be writable during production but not after leaving the fabrication.

4.4.4.1 SECRET_KEY

SHE must contain a unique secret key SECRET_KEY that shall not only be derived from the serial number or any other publicly available information.

The SECRET_KEY has to be inserted during chip fabrication by the semiconductor manufacturer and should not be stored outside of SHE.

The SECRET_KEY must at least meet the requirements of class P2 from [7]. It can be generated by a certified physical random number generator, e.g. a Hardware Security Module (HSM).

The SECRET_KEY may only be used to import/export keys.

4.4.4.2 Unique identification item UID

SHE or the microcontroller must contain a unique identification item, i.e. a serial number, of at most 120 bits. The identification item must be directly accessible by the controller logic of SHE.

Note: The UID is specified to 120 bit because it is always used in conjunction with two key ids or the status register to form a 128 bit block.

If the identification item is smaller than 120 bits it has to be padded with zero bits on the MSB side before feeding it into SHE.

The UID has to be inserted during chip fabrication by the semiconductor manufacturer.

The UID may not be 0, i.e. at least one bit has to be set. The UID with the value 0 is reserved as a wildcard UID for updating keys.

Note: The UID does not have to follow a special format, i.e. already inserted serial numbers can be reused. It should at least be unique for all chips of a certain manufacturer.

The UID may be exported through additional ports from SHE to be used with other components on the microcontroller. It must not be possible to affect the operation or status of SHE through this port.

4.4.5 Identification of memory slots

All memory slots being accessible by user-functions must be addressable by a four-bit value. The internal, physical addressing may differ. Table 4.1 shows the address of every key.

Key name	Address (hexadecimal)	Memory area
SECRET_KEY	0x0	ROM
MASTER_ECU_KEY	0x1	non-volatile
BOOT_MAC_KEY	0x2	
BOOT_MAC	0x3	
KEY_1	0x4	
KEY_2	0x5	
KEY_3	0x6	
KEY_4	0x7	
KEY_5	0x8	
KEY_6	0x9	
KEY_7	0xa	
KEY_8	0xb	
KEY_9	0xc	
KEY_10	0xd	
RAM_KEY	0xe	volatile

Table 4.1: Key addresses

4.5 Random number generation

SHE must include pseudo random number generator. The seed can be generated in two ways as described in Chapter 4.5.1

Note: Only one of the two described methods has to be implemented

The random numbers may not be generated directly by a true random number generator.

Note: A "P1 medium" [7] true random number generator (TRNG) may not be directly used for the purposes of SHE due to cryptographic reasons. Even smart cards or other advanced security solutions which possess high-quality physical sources of randomness usually rely at least on a compression of the TRNG output.

Note: The pseudo random number generator is designed to provide random numbers at constant quality for generating challenges for authentication at rates only limited by the performance of the underlying algorithm (AES).

4.5.1 Seed generation

The seed for the pseudo random number generator can be generated in two different ways. Either by implementing another pseudo random number generator as described in Chapter 4.5.1.1 or by compressing the output of a true random number generator as described in Chapter 4.5.1.2.

4.5.1.1 Seed generation through a pseudo random number generator (PRNG)

The seed is a non-volatile 128 bit value used as the input to the random number generator.

The seed is updated by invoking the command CMD_INIT_RNG, see Chapter 4.7.10

To update the seed, a key is derived with PRNG_SEED_KEY_C from SECRET_KEY (see Chapter 4.12 for the values of the single constants). The derived key is used to encrypt PRNG_SEED. The output has to be stored to PRNG_SEED first and must only be transferred to PRNG_STATE after completing the write transaction to PRNG_SEED.

$\texttt{PRNG\_SEED\_KEY = KDF(SECRET\_KEY, PRNG\_SEED\_KEY\_C)}$
$\texttt{PRNG\_SEED}_{\texttt{i}} = \texttt{ENC}_{\texttt{ECB,PRNG\_SEED\_KEY}}(\texttt{PRNG\_SEED}_{\texttt{i-1}})$

If PRNG_KEY is implemented as a volatile memory slot, a key for running the PRNG has to be derived from SECRET_KEY and PRNG_KEY_C (see Chapter 4.12 for the values of the single constants).

```
PRNG_KEY = KDF(SECRET_KEY, PRNG_KEY_C)
```

4.5.1.2 Seed generation trough a true random number generator (TRNG)

The TRNG must at least fulfill the requirement "P1 medium" as described in [7] in all operating conditions according to the rating of the chip.

The seed is a 128 bit value used as the input to the random number generator. After generation it has to be stored as PRNG_STATE.

The seed is generated by invoking the command CMD_INIT_RNG, see Chapter 4.7.10.

The entropy of the seed needs to be at least 80 bits, i.e. if the level of entropy delivered by the TRNG is lower, enough entropy has to be collected and compressed. The semiconductor manufacturer has to provide evidence of reaching this entropy level upon request or in the data sheet.

The compression function has to be called at least once to generate the seed.

To compress the output of the TRNG into the seed the Miyaguchi-Preneel compression function can be used as defined in Chapter 4.3.3

If PRNG_KEY (see Chapter 4.4.3.2) is implemented as a volatile memory slot, a key for running the PRNG has to be derived according to Chapter 4.5.1.1.

4.5.2 Random generation

The random number generator must not output any random values before the seed is updated as described above.

To generate a new random value according to E.4 [8], the content of PRNG_STATE is encrypted with PRNG_KEY. The output of the encryption is used as the input for the next random generation (encryption), i.e. it replaces PRNG_STATE, and is also output to the user, see Figure 4.4 for details on the workflow:

$\mathtt{PRNG\_STATE}_{\mathtt{i}} = \mathtt{ENC}_{\mathtt{ECB,PRNG\_KEY}}(\mathtt{PRNG\_STATE}_{\mathtt{i-1}})$
$\mathtt{RND} = \mathtt{PRNG\_STATE}_{\mathtt{i}}$

That is, the implementation of PRNG must fulfill K3 with strength of mechanisms "high" according to [8].

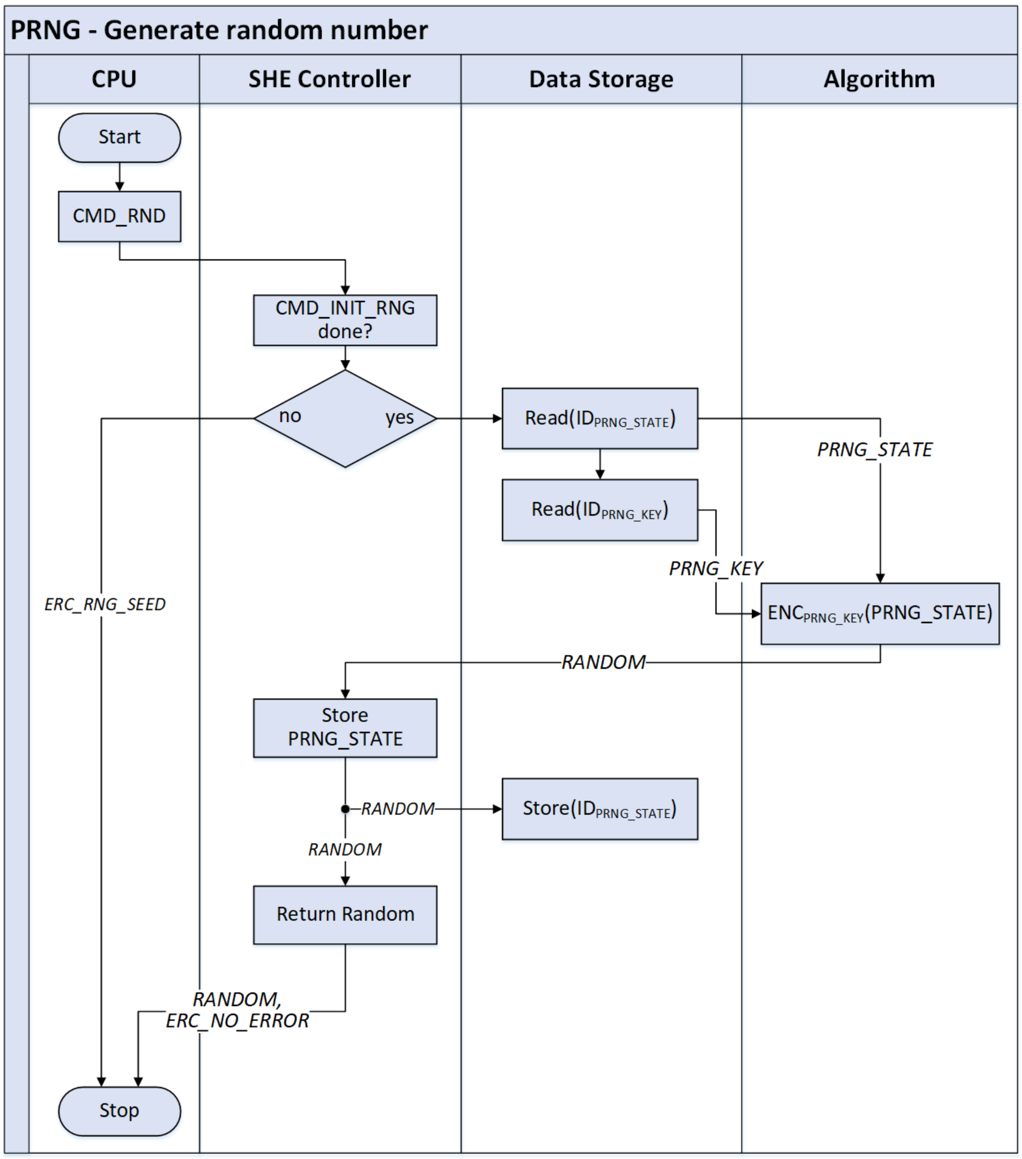

Figure 4.4: Random number generation

4.5.3 Extending the seed

The seed and the current PRNG_STATE can be extended by any user by calling the function CMD_EXTEND_SEED and supplying 128 bit of entropy. The seed and the state are extended by compressing the concatenation of the old seed/state and the entropy input with the Miyaguchi-Preneel compression function described above. The input to the compression function has to be preprocessed according to Chapter 4.3.3. Since the compression is a fixed length operation for 2x128 bit = 256 bit the padding becomes a constant operation. The padding is given as constant PRNG_EXTENSION_C in Chapter 4.12.

```
PRNG_STATE = AES-MP(PRNG_STATE | ENTROPY)
PRNG_SEED = AES-MP(PRNG_SEED | ENTROPY)
```

4.6 Status Register

A status register can be read by the CPU to check the internal state of SHE. The status register has a total width of 8 bits. If a bit is set its value is ’1’, if it is cleared its value is ’0’.

Bit	Name	Description
2^0	BUSY	The bit is set whenever SHE is processing a command.
2^1	SECURE_BOOT	The bit is set if the secure booting is activated.
2^2	BOOT_INIT	The bit is set if the secure booting has been personalized during the boot sequence.
2^3	BOOT_FINISHED	The bit is set when the secure booting has been finished by calling either CMD_BOOT_FAILURE or CMD_BOOT_OK or if CMD_SECURE_BOOT failed in verifying BOOT_MAC.
2^4	BOOT_OK	The bit is set if the secure booting (CMD_SECURE_BOOT) succeeded. If CMD_BOOT_FAILURE is called the bit is erased.
2^5	RND_INIT	The bit is set if the random number generator has been initialized.
2^6	EXT_DEBUGGER	The bit is set if an external debugger is connected to the chip, i.e. it reflects the input for debugger activation.
2^7	INT_DEBUGGER	The bit is set if the internal debugging mechanisms of SHE are activated

4.7 User-accessible Functions

SHE provides several functions to the CPU. In general, only a single function can be executed at a given time. Only the commands CMD_GET_STATUS (Chapter 4.7.16) and CMD_CANCEL (Chapter 4.7.18) may be called while another function is processed.

In the following subchapters all available functions are listed together with the necessary parameters, the direction of the parameters and the width of the single parameters in bits.

The function interface has to be asynchronous, i.e. all functions have to be non-blocking for the CPU and therefore return immediately.

Despite of returning data, every function must be able to return an error code to communicate the status of the processing to the calling application, see Chapter 4.8 for details on error codes.

If a function returns an error code the output data has to be set to '0'.

Note: If output has been generated and transmitted, e.g., on CBC operations or via DMA transfer, this output does not have to be deleted but the output of the round containing an error has to be '0'. Especially the CMAC functions may not output anything on errors.

If not stated differently, all functions can only process complete blocks of data, i.e. 128 bits of data, or multiples of the block size.

Note: The functions are described on a high level and should be segmented into several sub functions to comply with the requirements above. Handling of data can also be implemented in a more sophisticated way, e.g. DMA transfers for processing larger blocks of data.

Note: Implementation of a DMA interface is strongly suggested if supported by the CPU. This will allow for a larger set of supported use-cases and more efficient software design.

An interrupt may optionally be implemented to provide information about the status of the commands, e.g. send an interrupt when the execution of a command is finished.

4.7.1 Encryption: CMD_ENC_ECB

Parameter	Direction	Width
KEY_ID	IN	4
PLAINTEXT	IN	128
CIPHERTEXT	OUT	128
The function encrypts a given PLAINTEXT with the key identified by KEY_ID and returns the CIPHERTEXT. See Table 4.4 for an overview of the allowed keys for the operation.		

▽

△

Parameter	Direction	Width
`CIPHERTEXT` = $\mathrm{ENC}_{\mathrm{ECB},\mathrm{KEY\_KEY\_ID}}$ `(PLAINTEXT)`		
Error codes: `ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_KEY_NOT_AVAILABLE, ERC_KEY_INVALID, ERC_KEY_EMPTY, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR`		

4.7.2 Encryption: CMD_ENC_CBC

Parameter	Direction	Width
`KEY_ID`	`IN`	`4`
`IV`	`IN`	`128`
`PLAINTEXT`	`IN`	`n * 128`
`CIPHERTEXT`	`OUT`	`n * 128`
The function encrypts a given PLAINTEXT with the key identified by KEY_ID and returns the CIPHERTEXT. See Table 4.4 for an overview of the allowed keys for the operation. The plaintext can have a multiple length of the block width of 128 bit, n = 1, 2, 3, ...		
`CIPHERTEXT` = $\mathrm{ENC}_{\mathrm{CBC},\mathrm{KEY\_KEY\_ID},\mathrm{IV}}$ `(PLAINTEXT)`		
Error codes: `ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_KEY_NOT_AVAILABLE, ERC_KEY_INVALID, ERC_KEY_EMPTY, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR`		

4.7.3 Decryption: CMD_DEC_ECB

Parameter	Direction	Width
`KEY_ID`	`IN`	`4`
`CIPHERTEXT`	`IN`	`128`
`PLAINTEXT`	`OUT`	`128`
The function decrypts a given CIPHERTEXT with the key identified by KEY_ID and returns the PLAINTEXT. See Table 4.4 for an overview of the allowed keys for the operation.		
`PLAINTEXT` = $\mathrm{DEC}_{\mathrm{ECB},\mathrm{KEY\_KEY\_ID}}$ `(CIPHERTEXT)`		
Error codes: `ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_KEY_NOT_AVAILABLE, ERC_KEY_INVALID, ERC_KEY_EMPTY, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR`		

▽

4.7.4 Decryption: CMD_DEC_CBC

Parameter	Direction	Width
KEY_ID	IN	4
IV	IN	128
CIPHERTEXT	IN	n * 128
PLAINTEXT	OUT	n * 128
The function decrypts a given CIPHERTEXT with the key identified by KEY_ID and returns the PLAINTEXT. See Table 4.4 for an overview of the allowed keys for the operation. The plaintext can have a multiple length of the block width of 128 bit, n = 1, 2, 3, ...		
PLAINTEXT = $DEC_{CBC,KEY\_KEY\_ID,IV}$ (CIPHERTEXT)		
Error codes: ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_KEY_NOT_AVAILABLE, ERC_KEY_INVALID, ERC_KEY_EMPTY, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR		

4.7.5 MAC generation: CMD_GENERATE_MAC

Parameter	Direction	Width
KEY_ID	IN	4
MESSAGE_LENGTH	IN	64
MESSAGE	IN	n * 128
MAC	OUT	128
The function generates a MAC of a given MESSAGE with the help of a key identified by KEY_ID. See Table 4.4 for an overview of the allowed keys for the operation. The function has to discard the calculated MAC and return an error if the provided message has another length than stated in MESSAGE_LENGTH (bitlength of the message). All padding is done by SHE according to the length provided by MESSAGE_LENGTH. n = CEIL[1](MESSAGE_LENGTH / 128)		
MAC = $CMAC_{KEY\_KEY\_ID}$ (MESSAGE, MESSAGE_LENGTH)		
Error codes: ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_KEY_NOT_AVAILABLE, ERC_KEY_INVALID, ERC_KEY_EMPTY, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR		

[1]Let CEIL be the ceiling function such that *CEIL*(x) = min { n $\in \aleph_1$ |n $\geq$ x } ,$\aleph_1$ = {1,2,3,...}

4.7.6 MAC verification: CMD_VERIFY_MAC

Parameter	Direction	Width
KEY_ID	IN	4
MESSAGE_LENGTH	IN	64
MESSAGE	IN	n * 128
MAC	IN	128
MAC_LENGTH	IN	7
VERIFICATION_STATUS	OUT	1
The function verifies the MAC of a given MESSAGE with the help of a key identified by KEY_ID against a provided MAC. See Table 4.4 for an overview of the allowed keys for the operation. The number of bits to compare, starting from the leftmost bit of the MAC, are given in the parameter MAC_LENGTH, the value 0 is not allowed and is interpreted by SHE as to compare all bits of the MAC. The function has to return an error if the provided message has another length than stated in MESSAGE_LENGTH (bitlength of the message). All padding is done by SHE according to the length provided by MESSAGE_LENGTH. n = CEIL[2](MESSAGE_LENGTH / 128)		
MAC_{calc} = TRUNCATE(CMAC$_{KEY\_KEY\_ID}$(MESSAGE, MESSAGE_LENGTH), MAC_LENGTH) MAC_{ref} = TRUNCATE(MAC, MAC_LENGTH) VERIFICATION_STATUS = (0 != (MAC_{calc} - MAC_{ref}))		
Error codes: ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_KEY_NOT_AVAILABLE, ERC_KEY_INVALID, ERC_KEY_EMPTY, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR		

[2]Let CEIL be the ceiling function such that $CEIL(x) = \min \{ n \in \aleph_1 | n \geq x \}, \aleph_1 = \{1,2,3,...\}$

4.7.7 Secure key update: CMD_LOAD_KEY

Parameter	Direction	Width
M_1	IN	128
M_2	IN	256
M_3	IN	128
M_4	OUT	256
M_5	OUT	128
The function updates an internal key of SHE with the protocol described in Chapter 4.9 If a protected key is loaded into RAM_KEY, the function has to disable the plain key status bit.		
See Chapter 9 for operation details.		
Error codes: ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_KEY_NOT_AVAILABLE, ERC_KEY_INVALID, ERC_KEY_WRITE_PROTECTED, ERC_KEY_UPDATE_ERROR, ERC_MEMORY_FAILURE, ERC_KEY_EMPTY, ERC_BUSY, ERC_GENERAL_ERROR		

4.7.8 Plain key update: CMD_LOAD_PLAIN_KEY

Parameter	Direction	Width
KEY	IN	128
A key KEY is loaded without encryption and verification of the key, i.e. the key is handed over in plaintext. A plain key can only be loaded into the RAM_KEY slot. The command sets the plain key status bit for the RAM_KEY.		
$KEY_{RAM\_KEY} = KEY$ RAM_KEY_PLAIN = 1		
Error codes: ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_BUSY, ERC_GENERAL_ERROR		

4.7.9 Export key: CMD_EXPORT_RAM_KEY

Parameter	Direction	Width
M_1	OUT	128
M_2	OUT	256
M_3	OUT	128
M_4	OUT	256
M_5	OUT	128
The function exports the RAM_KEY into a format protected by SECRET_KEY. The key can be imported again by using CMD_LOAD_KEY. A RAM_KEY can only be exported if it was written into SHE in plaintext, i.e. by CMD_LOAD_PLAIN_KEY. For loading a RAM_KEY and therefore for exporting a RAM_KEY the reserved fields for the security flags and the counter have to be set to 0. For further explanation on message details and contents as well as for arithmetic and padding, see Chapter 4.9. No other values than M_1, M_2, M_3, M_4, M_5 may leave SHE. *Note: due to setting counter and flags to zero by definition in case of RAM_KEY updates, the first block of the encrypted message M_2 becomes a constant zero block.*		
$K_1 = \mathrm{KDF}(\mathrm{KEY}_{\mathrm{SECRET\_KEY}}, \mathrm{KEY\_UPDATE\_ENC\_C})$ $K_2 = \mathrm{KDF}(\mathrm{KEY}_{\mathrm{SECRET\_KEY}}, \mathrm{KEY\_UPDATE\_MAC\_C})$ $C_{ID} = 0$ (28 bits) $F_{ID} = 0$ (5 bits) $M_1 = \mathrm{UID} \mid \mathrm{ID}_{\mathrm{RAM\_KEY}} \mid \mathrm{ID}_{\mathrm{SECRET\_KEY}}$ $M_2 = \mathrm{ENC}_{\mathrm{CBC},K1,IV=0}(C_{ID} \mid F_{ID} \mid \text{"0...0"}_{95} \mid \mathrm{KEY}_{\mathrm{RAM\_KEY}}) =$ $\mathrm{ENC}_{\mathrm{CBC},K1,IV=0}(\text{"0...0"}_{128} \mid \mathrm{KEY}_{\mathrm{RAM\_KEY}})$ $M_3 = \mathrm{CMAC}_{K2}(M_1 \mid M_2)$ $K_3 = \mathrm{KDF}(\mathrm{KEY}_{\mathrm{RAM\_KEY}}, \mathrm{KEY\_UPDATE\_ENC\_C})$ $K_4 = \mathrm{KDF}(\mathrm{KEY}_{\mathrm{RAM\_KEY}}, \mathrm{KEY\_UPDATE\_MAC\_C})$ $M_4 = \mathrm{UID} \mid \mathrm{ID}_{\mathrm{RAM\_KEY}} \mid \mathrm{ID}_{\mathrm{SECRET\_KEY}} \mid \mathrm{ENC}_{\mathrm{ECB},K3}(C_{ID})$ $M_5 = \mathrm{CMAC}_{K4}(M_4)$		
Error codes: ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_KEY_INVALID, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_KEY_EMPTY, ERC_GENERAL_ERROR		

▽

4.7.10 Initialize random number generator: CMD_INIT_RNG

Parameter	Direction	Width
`none`		
The function initializes the seed and derives a key for the `PRNG`. The function must be called before CMD_RND after every power cycle/reset. See Chapter 4.5.1.1 or Chapter 4.5.1.2 respectively for details on the initialization of the random number generator. The command has to ignore active debugger protection or secure boot protection flags on SECRET_KEY. *Note: The function may need several hundred milliseconds to return.*		
See Chapter 5.1.1 or Chapter 5.1.2 respectively for operation details.		
Error codes: `ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_MEMORY_FAILURE,ERC_BUSY, ERC_GENERAL_ERROR`		

4.7.11 Extend the PRNG seed: CMD_EXTEND_SEED

Parameter	Direction	Width
`ENTROPY`	`IN`	`128`
The function extends the seed of the `PRNG` by compressing the former seed value and the supplied entropy into a new seed which will be used to generate the following random numbers. The random number generator has to be initialized by CMD_INIT_RNG before the seed can be extended.		
See Chapter 5.3 for operation details.		
Error codes: `ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_RNG_SEED, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR`		

4.7.12 Generate random number: CMD_RND

Parameter	Direction	Width
`RND`	`OUT`	`128`
The function returns a vector of 128 random bits. The random number generator has to be initialized by CMD_INIT_RNG before random numbers can be supplied.		

`IF(`$\texttt{SREG}_{\texttt{RND\_INIT}}$` == 1)`
$\texttt{PRNG\_STATE}_i = \texttt{ENC}_{\texttt{ECB,KEY\_PRNG\_KEY}}(\texttt{PRNG\_STATE}_{i-1})$
$\texttt{RND} = \texttt{PRNG\_STATE}_i$
`END IF`		

Error codes: `ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_RNG_SEED,ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR`		

4.7.13 Bootloader verification (secure booting): CMD_SECURE_BOOT

Parameter	Direction	Width
`SIZE`	`IN`	`32`
`DATA`	`IN`	`SIZE * 8`
The function starts the secure boot process. `SHE` verifies the `MAC` of the Bootloader supplied in DATA of the size SIZE in bytes. The function must only be used once after every power cycle/reset and has to be locked afterwards.		
See Chapter 10 for operation details.		
Error codes: `ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_NO_SECURE_BOOT, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR`		

4.7.14 Impose sanctions during invalid boot: CMD_BOOT_FAILURE

Parameter	**Direction**	**Width**
none		
The command will impose the same sanctions as if CMD_SECURE_BOOT would detect a failure but can be used during later stages of the boot process, see Chapter 10.4 for details. CMD_BOOT_FAILURE may only be called once after every power cycle/reset and may only be called if CMD_SECURE_BOOT did not detect any errors before and if CMD_BOOT_OK was not called. *Note: If the secure booting is segmented into several stages and SHE does only provide the first, autonomous stage, this function can be used during the following stages, to provide the boot status to SHE.*		

IF($SREG_{SECURE\_BOOT}$ == 1 AND $SREG_{BOOT\_OK}$ == 1 AND $SREG_{BOOT\_FINISHED}$ == 0)
$SREG_{BOOT\_FINISHED}$ = 1
$SREG_{BOOT\_OK}$ = 0
END IF		

Error codes: ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_NO_SECURE_BOOT, ERC_BUSY, ERC_GENERAL_ERROR		

4.7.15 Finish boot verification: CMD_BOOT_OK

Parameter	**Direction**	**Width**
none		
The command is used to mark successful boot verification for later stages than CMD_SECURE_BOOT. In particular it is meant to lock the command CMD_BOOT_FAILURE. CMD_BOOT_OK may only be called once after every power cycle/reset and may only be called if CMD_SECURE_BOOT did not detect any errors before and if CMD_BOOT_FAILURE was not called. *Note: If the secure booting is segmented into several stages and SHE does only provide the first, autonomous stage, this function can be used during the following stages, to provide the boot status to SHE.*		

IF($SREG_{SECURE\_BOOT}$ == 1 AND $SREG_{BOOT\_OK}$ == 1 AND $SREG_{BOOT\_FINISHED}$ == 0)
$SREG_{BOOT\_FINISHED}$ = 1
END IF		

Error codes: ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_NO_SECURE_BOOT, ERC_BUSY, ERC_GENERAL_ERROR		

4.7.16 Read status of SHE: CMD_GET_STATUS

Parameter	Direction	Width
SREG	OUT	8
The function returns the contents of the status register. The function may be called in every state.		
Error codes: ERC_NO_ERROR, ERC_GENERAL_ERROR		

4.7.17 Get identity: CMD_GET_ID

Parameter	Direction	Width
CHALLENGE	IN	128
ID	OUT	120
SREG	OUT	8
MAC	OUT	128
The function returns the identity (UID) and the value of the status register protected by a MAC over a challenge and the data. If MASTER_ECU_KEY is empty, the returned MAC has to be set to 0.		

ID = UID
MAC = $\text{CMAC}_{\text{KEY\_MASTER\_ECU\_KEY}}$(CHALLENGE|ID |SREG)		

Error codes: ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_KEY_NOT_AVAILABLE, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR		

4.7.18 Cancel function: CMD_CANCEL

Parameter	Direction	Width
`none`		
The CMD_CANCEL will interrupt any given function and discard all calculations and results. If CMD_CANCEL is called during boot verification, the same actions have to be performed as on failed boot measurement, see Chapter 4.10.4. If CMD_CANCEL is called during update of a memory slot the former value of the memory slot has to be kept if called before the physical write operation or the new value has to be persisted if called during or after the physical write operation. The command may be called in every state. The latency (min/average/max) of CMD_CANCEL, i.e. the time between issuing CMD_CANCEL and clearing SREG$_{BUSY}$, should be denoted in the datasheets. *Note: After calling CMD_CANCEL there may be some latency due to stopping data transfers and cleaning up internal data before the busy flag (SREG$_{BUSY}$) is cleared and SHE becomes available for processing further commands. The application needs to be able to handle this situation.*		
Error codes: `ERC_NO_ERROR, ERC_GENERAL_ERROR`		

4.7.19 Debugger activation: CMD_DEBUG

Parameter	Direction	Width
CHALLENGE	OUT	128
AUTHORIZATION	IN	128
CMD_DEBUG is used to activate any internal debugging facilities of SHE, see Chapter 4.11. Activating the internal debugger implies the deletion of all keys except SECRET_KEY, UID and PRNG_SEED and will only work, if no key is write-protected. The command has to ignore active debugger protection and secure boot protection flags as well as empty keys. After successful authentication and deletion of all keys, the bit $\mathrm{SREG}_{\texttt{RND\_INIT}}$ has to be reset to ’0’. *Note: The command can also be used to reset SHE to the factory defaults if no key is writeprotected.*		

SHE & Backend:
K = KDF(KEY$_{\texttt{MASTER\_ECU\_KEY}}$, DEBUG_KEY_C)
 	
SHE:
CHALLENGE = CMD_RND()
 	
Backend:
AUTHORIZATION = CMAC$_K$ (CHALLENGE |UID)		

Error codes: ERC_NO_ERROR, ERC_SEQUENCE_ERROR, ERC_WRITE_PROTECTED, ERC_RNG_SEED, ERC_NO_DEBUGGING, ERC_MEMORY_FAILURE, ERC_BUSY, ERC_GENERAL_ERROR		

4.8 Error Codes

The SHE coprocessor will provide error codes after calling a command. The following error codes are defined and should be handled appropriately by the calling application. Table 6 shows which error codes may occur with the single functions of SHE.

4.8.1 ERC_NO_ERROR

No error has occurred and the command will be executed.

4.8.2 ERC_SEQUENCE_ERROR

This error code is returned by SHE whenever the sequence of commands or subcommands is out of sequence, e.g. when a function is called while another function is still running.

4.8.3 ERC_KEY_NOT_AVAILABLE

This error code is returned if a key is locked due to failed boot measurement or an active debugger.

4.8.4 ERC_KEY_INVALID

This error code is returned by SHE whenever a function is called to perform an operation with a key that is not allowed for the given operation.

4.8.5 ERC_KEY_EMPTY

This error code is returned by SHE if the application attempts to use a key that has not been initialized yet.

4.8.6 ERC_NO_SECURE_BOOT

This error is returned by the command CMD_SECURE_BOOT (see Chapter 4.7.13) when the conditions for a secure boot process are not met and the secure boot process has to be canceled. It is also returned if a function changing the boot status is called without secure booting or after finishing the secure boot process.

4.8.7 ERC_KEY_WRITE_PROTECTED

This error is returned when a key update is attempted on a memory slot that has been write protected or when an attempt to active the debugger is started when a key is write-protected.

4.8.8 ERC_KEY_UPDATE_ERROR

This error is returned when a key update did not succeed due to errors in verification of the messages.

4.8.9 ERC_RNG_SEED

The error code is returned by CMD_RND and CMD_DEBUG if the seed has not been initialized before.

4.8.10 ERC_NO_DEBUGGING

The error code is returned if internal debugging is not possible because the authentication with the challenge response protocol did not succeed.

4.8.11 ERC_BUSY

This error code is returned whenever a function of SHE is called while another function is still processing, i.e., when SREGBUSY = 1. It should not matter if the order of commands is correct. If CMD_CANCEL or CMD_GET_STATUS is called while another function is processing they should not return ERC_BUSY but be processed in parallel, as stated in Chapter 4.7.

4.8.12 ERC_MEMORY_FAILURE

This error code can be returned if the underlying memory technology is able to detect physical errors, e.g. flipped bits etc., during memory read or write operations to notify the application.

4.8.13 ERC_GENERAL_ERROR

This error code is returned if an error not covered by the error codes above is detected inside SHE.

4.9 Memory update protocol

SHE provides several memory slots to store keys and other required information with the possibility to change the contents whenever necessary. The memory slots are protected by different policies, depending on the use-case of the memory slot. However, when updating a memory slot a secret has to be known to authorize for changing the memory. There is a single protocol for updating all memory slots. Only a single memory slot can be updated by running the protocol at once. The protocol is described in the following subchapters.

The protocol consists of two parts: the memory update itself as well as a verification message which can be passed back to the issuer of the update to prove the successful update. The protocol is secured against replay attacks by including a counter value stored within SHE. Furthermore it provides confidentiality, integrity and authenticity. By transferring messages back to the external party, the successful update of a memory position can be proven.

The number of updates for a single memory slot is only limited by the width of the counter and the physical memory write endurance.

The protocol described in the following sections has to be implemented to update non-volatile memory slots or the RAM_KEY with the command CMD_LOAD_KEY.

The generation of the messages M_1, M_2, M_3, M_4, M_5 has to be implemented for the command CMD_EXPORT_RAM_KEY.

The memory update protocol is segmented into two parts, first the actual memory update to transfer a confidential and authentic key to SHE and a second verification part to provide evidence of a successful key update to the backend.

4.9.1 Description of the memory update protocol

To update a memory slot, e.g. a key, the external entity, e.g. the backend, must have knowledge of a valid authentication secret, i.e. another key, which is identified by AuthID. See Table 4.5 for details on which authentication secret must be known to update a certain memory slot.

The backend has to derive two keys K_1 and K_2 (see Chapter 4.3.2) from $KEY_{\mathtt{AuthID}}$ with the constants KEY_UPDATE_ENC_C or KEY_UPDATE_MAC_C respectively. Then three messages M_1..M_3 are generated. M_1 is a concatenation of the UID of the addressed SHE module, the ID of the memory slot to be updated and the AuthID. M_2 is the CBC-encrypted concatenation of the new counter value $C_{\mathtt{ID}}$', the according flags $F_{\mathtt{ID}}$', a pattern to fill the first block with '0' bits and the new key $K_{\mathtt{ID}}$'. The initialization vector for the encryption is IV = 0, the key is K_1 the message is padded by concatenating a single "1"-bit followed "0"-bits on the LSB side. The flags $F_{\mathtt{ID}}$' are given as

F_{ID}' = WRITE_PROTECTION | BOOT_PROTECTION | DEBUGGER_PROTECTION | KEY_USAGE | WILDCARD

The last message M_3 is a verification message and is calculated as a $CMAC_{K2}$ over the concatenation of M_1 and M_2.

All three messages must be transferred to SHE.

SHE checks the write protection of the memory slot ID and only proceeds if the write protection is not set.

If the key AuthID is empty, the key update must only work if AuthID = ID, otherwise ERC_KEY_EMPTY is returned.

The debugger protection and secure boot protection (see Chapter 4.4.1.3 and Chapter 4.4.1.2) have to be checked if ID = ID_{RAM_KEY}. If one of the protections is active, ERC_KEY_NOT_AVAILABLE is returned. For other IDs and for all AuthIDs the check of debugger and secure boot protection is omitted.

In case of memory failures, at least AuthID, C_{ID}, F_{ID} and U_{ID} (if not in wildcard mode) have to be readable to perform an update, otherwise ERC_MEMORY_FAILURE is returned.

Note: A key update of non-volatile keys can be performed even if the security bits for failed secure boot measurement or active debugger would prevent access since the update process is authentic and confidential by itself.

In the following SHE also derives K_2 and verifies the message M_3. If the verification is successful, SHE first checks if the supplied UID' matches the wildcard value 0. If the UID' is a wildcard SHE checks if wildcards are allowed for that particular key by checking the stored flag and proceeds to read the AuthID or interrupts the protocol. If the UID' in M_1 is no wildcard it is compared to the actual UID of SHE and the protocol only proceeds if the values match.

In the next step K_1 is derived by SHE to decrypt M_2. If the new counter value CID' from M_2 is bigger than the internal counter value C_{ID}, SHE has to proceed to store the transmitted counter value, key and flags.

Note: A physical write operation should only be issued if the new value is different from the current value

All intermediate values, e.g. K_x and decrypted values, may not leave SHE. The update of a certain memory slot may not affect any other key slot.

See Figure 4.5 for a flow chart of the protocol. After successful storage of all data a verification message is generated, see next chapter for details. If the protocol is used to load a RAM_KEY the flags and counter value have to be set to 0 and SHE must ignore their value.

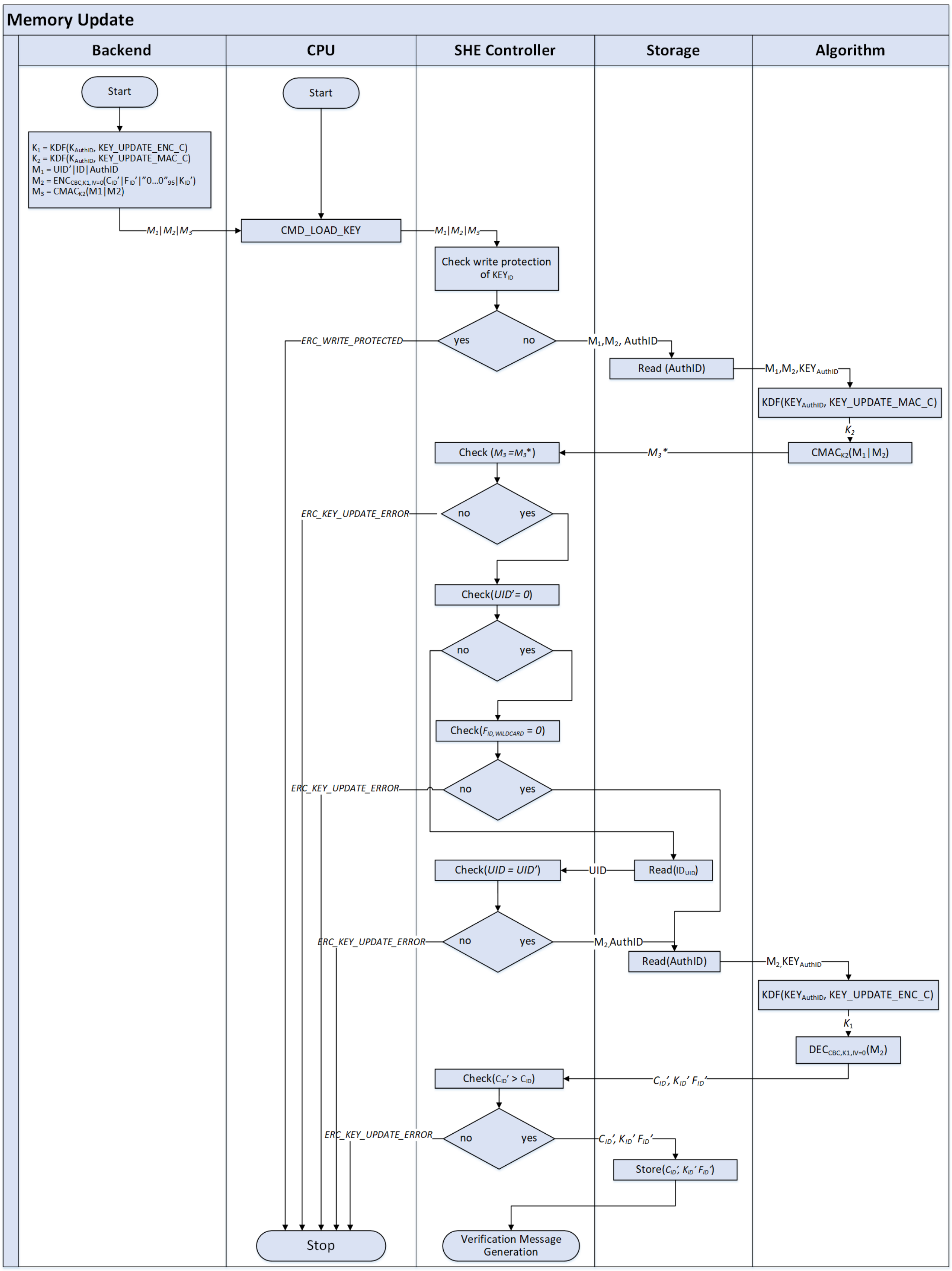
Memory Update
Backend
CPU
SHE Controller
Storage
Algorithm
Start
Start
K_1 = KDF(K_{AuthID}, KEY_UPDATE_ENC_C)
K_2 = KDF(K_{AuthID}, KEY_UPDATE_MAC_C)
M_1 = UID'|ID|AuthID
M_2 = $ENC_{CBC,K1,IV=0}$(C_{ID}'|F_{ID}'|"0...0"$_{95}$|K_{ID}')
M_3 = $CMAC_{K2}$(M1|M2)
$M_1|M_2|M_3$
CMD_LOAD_KEY
$M_1|M_2|M_3$
Check write protection of KEY_{ID}
ERC_WRITE_PROTECTED
yes
no
M_1,M_2, AuthID
Read (AuthID)
M_1,M_2,KEY_{AuthID}
KDF(KEY_{AuthID}, KEY_UPDATE_MAC_C)
K_2
Check (M_3 =M_3*)
M_3*
$CMAC_{K2}$(M_1|M_2)
ERC_KEY_UPDATE_ERROR
no
yes
Check(UID' = 0)
no
yes
Check($F_{ID,WILDCARD}$ = 0)
ERC_KEY_UPDATE_ERROR
no
yes
Check(UID = UID')
UID
Read(ID_{UID})
ERC_KEY_UPDATE_ERROR
no
yes
M_2,AuthID
Read(AuthID)
M_2,KEY_{AuthID}
KDF(KEY_{AuthID}, KEY_UPDATE_ENC_C)
K_1
$DEC_{CBC,K1,IV=0}$(M_2)
Check(C_{ID}' > C_{ID})
C_{ID}', K_{ID}' F_{ID}'
ERC_KEY_UPDATE_ERROR
no
yes
C_{ID}', K_{ID}' F_{ID}'
Store(C_{ID}', K_{ID}' F_{ID}')
Stop
Verification Message Generation

Figure 4.5: Memory update protocol

4.9.2 Description of the update verification message generation

After updating a memory slot SHE has to generate a verification message which can be transferred to the backend to prove the successful update. See Figure 4.6 for a flow chart of the generation.

SHE first derives a key K_3 (see Chapter 4.3.3) from the updated memory slot ID and KEY_UPDATE_ENC_C to encrypt the stored counter value C_{ID} in ECB mode. The counter value has to be padded with a single "1"-bit followed by "0"-bits on the LSB side.

Next a message M_4 is generated by concatenating the UID, the ID of the updated memory slot, the used authentication secret AuthID and the encrypted counter value M_4*. Before encryption, the counter value is padded by concatenating a single "1"-bit followed "0"-bits on the LSB side.

Finally M_5 is generated by calculating a CMAC over the message M_4 with a key K_4 derived from the updated memory slot ID and KEY_UPDATE_MAC_C.

The messages M_4 and M_5 are then transferred to the backend.

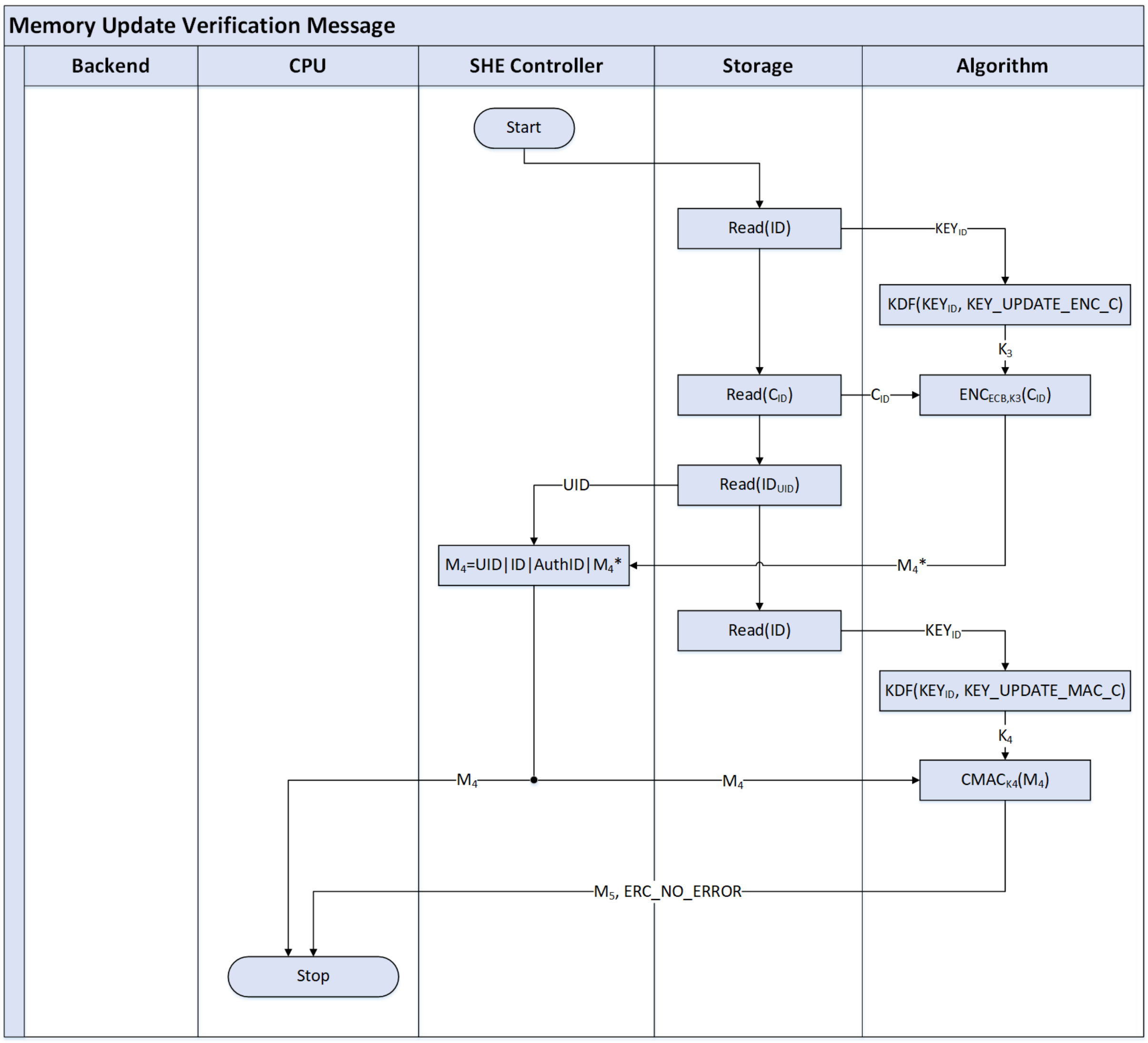

Figure 4.6: Verification message generation during memory update

4.10 Secure booting

The facilities of SHE can be used to secure the boot process, i.e., to monitor the authenticity of the software on every boot cycle. To achieve this, a task has to be run upon reset of the CPU and before handing control over to the application. The task could be implemented as a small addition to the ROM code of the microcontroller, comparable to the "core root of trust for measurement" (CRTM) as defined by the Trusted Computing Group, see [9].

The secure boot process verifies an area of the memory against internal data of SHE and will lock parts of SHE if the verification fails. The verified part is identical to the first user instructions executed right after initialization of the microcontroller and is called "SHE Bootloader" in the following sections

Note: The SHE Bootloader can be an additional Bootloader, an existing Bootloader or a for example a data section that should be protected. However, a data section would need a jump operation to the actual application in its first memory position..

The SHE Bootloader is located at the memory position SHE_BL_START and has the size SHE_BL_SIZE. Both values are not critical for security, hence they are not stored inside of SHE. The SHE Bootloader is not part of the internal ROM but of the user accessible memory.

The size of the Bootloader SHE_BL_SIZE has to be configurable by the application engineer and should not be fixed by the chip manufacturer. The value SHE_BL_START has to be writeable by the application engineer if the microcontroller architecture allows for different positions to boot from.

The actual storage position for these values is controlled by the manufacturer of the microcontroller and may be specific to every microcontroller. However, only the task starting the secure boot process has to deal with the values. The task has always to try to start the secure boot process and evaluate the error code by letting SHE determine if secure boot is configured.

The secure boot process should not interfere with the regular functionality of the CPU, i.e. if not explicitly activated; the CPU should perform as if SHE is not present.

To activate the secure boot feature a key has to be written into the BOOT_MAC_KEY key slot. Upon the next reset of the CPU SHE will personalize itself, see Chapter 4.10.3.

Note: Secure booting does not directly protect the application software but can prevent a malicious application from using certain keys. To protect the software as well, a dependency between the software and the keys has to be generated, e.g., by encrypting parts of the software.

Note: Secure booting provides a way to authorize the use of the stored keys by evaluating the integrity and authenticity of the booted configuration.

There have to be two ways to perform the secure boot which have to be configurable by the software application engineer in a non-volatile memory area provided by the microcontroller, see Chapter 4.10.1 and Chapter 4.10.2 for details.

4.10.1 Measurement before application start-up

In this case, the complete secure boot process is performed before the control over the microcontroller is handed over to the application.

The secure booting must always be finished before the CPU starts the application code. If direct memory access techniques are used to implement secure booting, the CPU must wait the end of the operation before starting the application code. If the DMA transfer is canceled or interrupted by any means, SHE has to get notified and the secure boot has to be marked as failed.

4.10.1.1 Exemplary implementation: extension of the boot code

The "boot code" means the internal ROM code of the microcontroller, executed right after reset to initialize the CPU and the peripherals before starting the user application.

The internal boot code of the microcontroller has to be extended by a small program to start the verification of a SHE Bootloader.

The boot code extension has to fulfill several tasks in the following order:

- Retrieve the size SHE_BL_SIZE of the SHE Bootloader
- Read the Bootloader data from the address SHE_BL_START to SHE_BL_START + SHE_BL_SIZE and use CMD_SECURE_BOOT to send it to SHE
- Start the execution of the SHE Bootloader at SHE_BL_STAR

The extension of the boot code must guarantee to start exactly the same code as verified before, i.e. it has to start execution at SHE_BL_START.

Figure 4.7 should illustrate how the extension of the boot code should work. The following variables are used within the example:

SHE_BL_SIZE	The length in bytes of the bootloader; stored in internal or external memory of the CPU as an unsigned value of 32 bit width
SHE_BL_START	Address of the first byte of the bootloader. The address might be fixed for certain CPU architectures or it might be stored in internal or external memory.
BOOT_POSITION	Address of the first instruction after a regular reset. Can be the same as SHE_BL_START.

Note: The command CMD_SECURE_BOOT has been separated into three sub commands called INIT, UPDATE and FINALIZE. The INIT-part is used to start the function and hand over the necessary parameters. This first sub function will also check the conditions to decide if a secure boot process is performed. The second sub command *is used to transfer the bulk data from the CPU to SHE and the third command is used to tell SHE that the process is finished.*

```
// Existing CPU specific initialization instructions
BL_SIZE = READ_FROM_MEM(@SHE_BL_SIZE)
IF [CMD_SECURE_BOOT__INIT(BL_SIZE) == ERC_NO_ERROR] THEN
    FOR (i = 0, i++, i < BL_SIZE)
        DATA = READ_FROM_MEM(@SHE_BL_START + i)
        CMD_SECURE_BOOT__UPDATE(DATA)
    END FOR
    CMD_SECURE_BOOT__FINALIZE()
    GOTO SHE_BL_START
END IF
// A jump to the regular boot position can be performed here
```

Figure 4.7: Pseudo code extending the ROM code of a non-DMA microcontroller

4.10.2 Measurement during application start-up

This option only has to be implemented on microcontrollers supporting direct memory access techniques.

In this case, the secure boot process is only initialized and started before the control over the microcontroller is handed over to the application, i.e., the direct memory access is parametrized and started. The actual measurement of the code is executed in parallel to the application start-up.

All keys protected with the secure boot flag must be deactivated upon reset and will only be activated after successful completion of CMD_SECURE_BOOT.

Note: Since SHE can only process a single operation at a given time, SHE will not be accessible for the application until the secure boot process is finished. However, the application can boot without delay and can potentially interrupt the secure boot process.

SHE has to stop the secure boot process and treat it as failed if the data transfer operation is canceled or manipulated by the application.

No write operations to the memory area being measured are allowed during secure boot. If the memory is written during secure boot, SHE has to stop the secure boot process and treat it as failed. Deactivating write capabilities during secure boot is an equivalent measure.

4.10.3 Autonomous bootstrap configuration of the secure boot process

If required an automatic/autonomous learning of the individual MACrequired for the secure boot process can be achieved. The mechanism does not require to initialize SHE with a pre-calculated MAC but only with the according key. The bootstrap process is triggered on the first reset of SHE after writing a key to BOOT_MAC_KEY.

The bootstrap configuration for the secure boot process is done autonomously by the following mechanism after the first reset after a key is written into the according key slot.

When SHE detects a boot key but no stored MAC, the CPU will perform the same task as during a secured boot, i.e. it will read the program flash and pipe it to SHE but SHE does not validate the calculated MAC but stores it to its internal memory. On the next reset the boot up will be secured by SHE.

Note: The position and size of the Bootloader must be initialized accordingly.

4.10.4 Sanctions on fail of boot measurement

If SHE detects a failure during the boot process, e.g., if the calculated MAC does not match the stored MAC, the boot process is disturbed or keys are inaccessible due to memory failures (ERC_MEMORY_FAILURE) or debugging protected keys (ERC_KEY_NOT_AVAILABLE), the following sanctions have to be imposed:

The appropriate bits in the status register are set to flag the failed boot process to the application, i.e., $\text{SREG}_{\texttt{BOOT\_FINISHED}} = 1$ and $\text{SREG}_{\texttt{BOOT\_OK}} = 0$.

The commands CMD_BOOT_FAILURE and CMD_BOOT_OK must be locked.

All keys being marked according to Chapter 4.4.1.2 must be locked.

4.10.5 Optional: Enforcing authenticated software

Note: This boot mode is not mandatory for SHE 1.0 compliance.

Note: In this configuration mode the CPU refuses to start any application being not authentic.

The secure boot operation is performed before starting the application (cf. Chapter 4.10.1.1) but if the secure boot operation is not successful, it has to be restarted. Compare pseudo code in Figure 4.8 to code in Figure 4.7 for an example how the boot mode to enforce authenticated software can be implemented if an implementation in ROM code is chosen.

The configuration of boot modes has to be stored in an one-time-programmable area. If this optional boot mode is configured to be performed, the CPU must prevent any other boot option than configured for the secure boot, e.g., reconfiguring the boot memory must not be possible and using boot-strap modes over communication interfaces must not be possible either.

```
// Existing CPU specific initialization instructions
BL_SIZE = READ_FROM_MEM(@SHE_BL_SIZE)
IF [CMD_SECURE_BOOT__INIT(BL_SIZE) == ERC_NO_ERROR] THEN
    LABEL SECURE_BOOT_CHECK:
    FOR (i = 0, i++, i < BL_SIZE)
        DATA = READ_FROM_MEM(@SHE_BL_START + i)
        CMD_SECURE_BOOT__UPDATE(DATA)
    END FOR
    CMD_SECURE_BOOT__FINALIZE()
    //enforce authenticated software upon boot
    SREG = CMD_GET_STATUS()
    IF [SREG.BOOT_OK == 0 AND OTPMEM.ENFORCE_AUTH_BOOT == 1] THEN
        RESET_SHE()
        GOTO SECURE_BOOT_CHECK
    END IF
    GOTO SHE_BL_START
END IF
// A jump to the regular boot position can be performed here
```

Figure 4.8: Pseudo ROM boot code for enforcing an authentic application

The keys protected with the secure boot flag have to be locked if the boot measurement fails, even though the process should be restarted immediately.

Note: To make this boot option useful, the decision if the boot measurement has to be redone should be protected against certain hardware attacks, e.g., glitching attacks. However, since this boot mode is optional no requirements regarding tamper resistance are made.

4.10.6 Optional: Flow charts

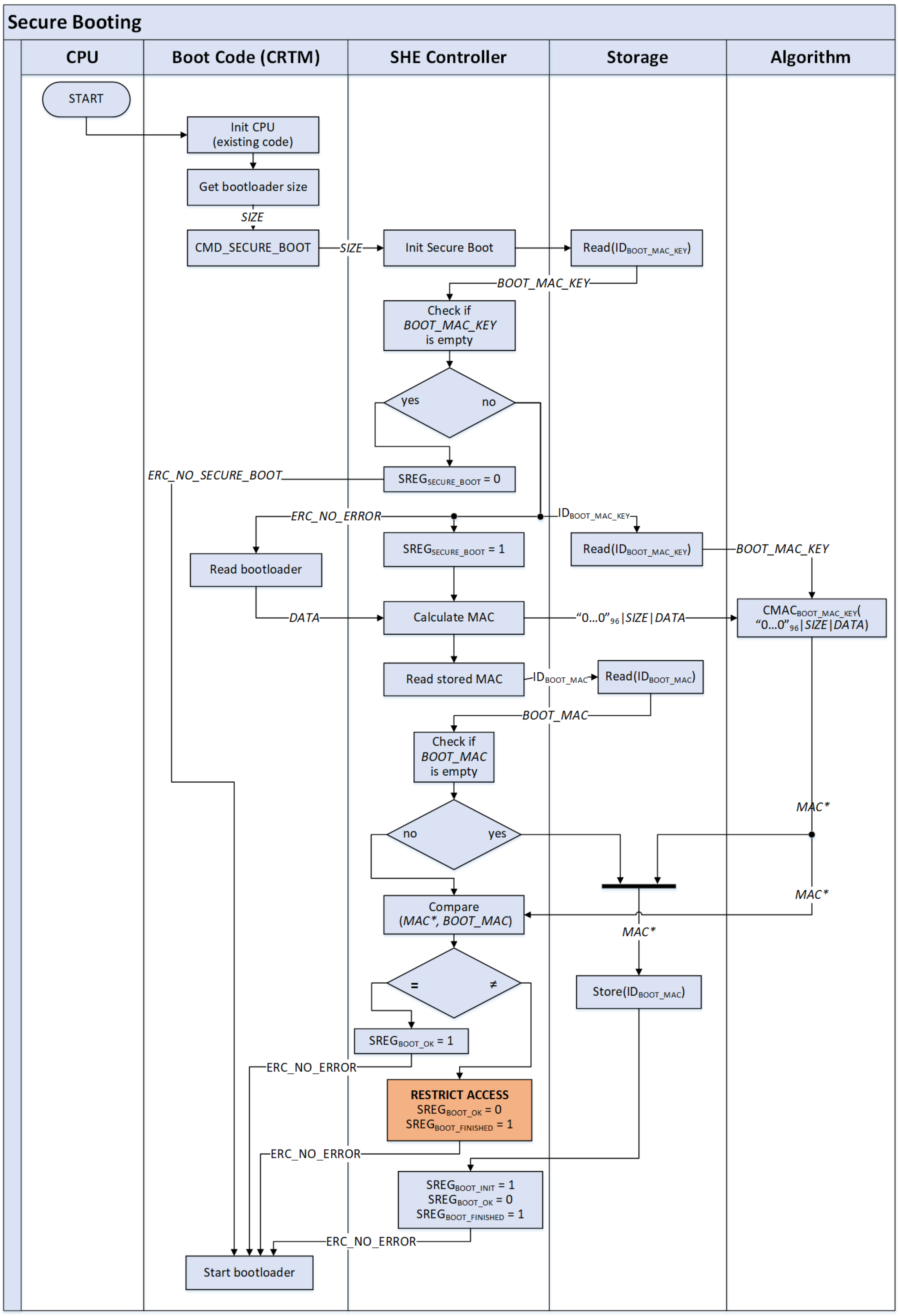

Figure 4.9: Flow chart of the secure boot process (without DMA)

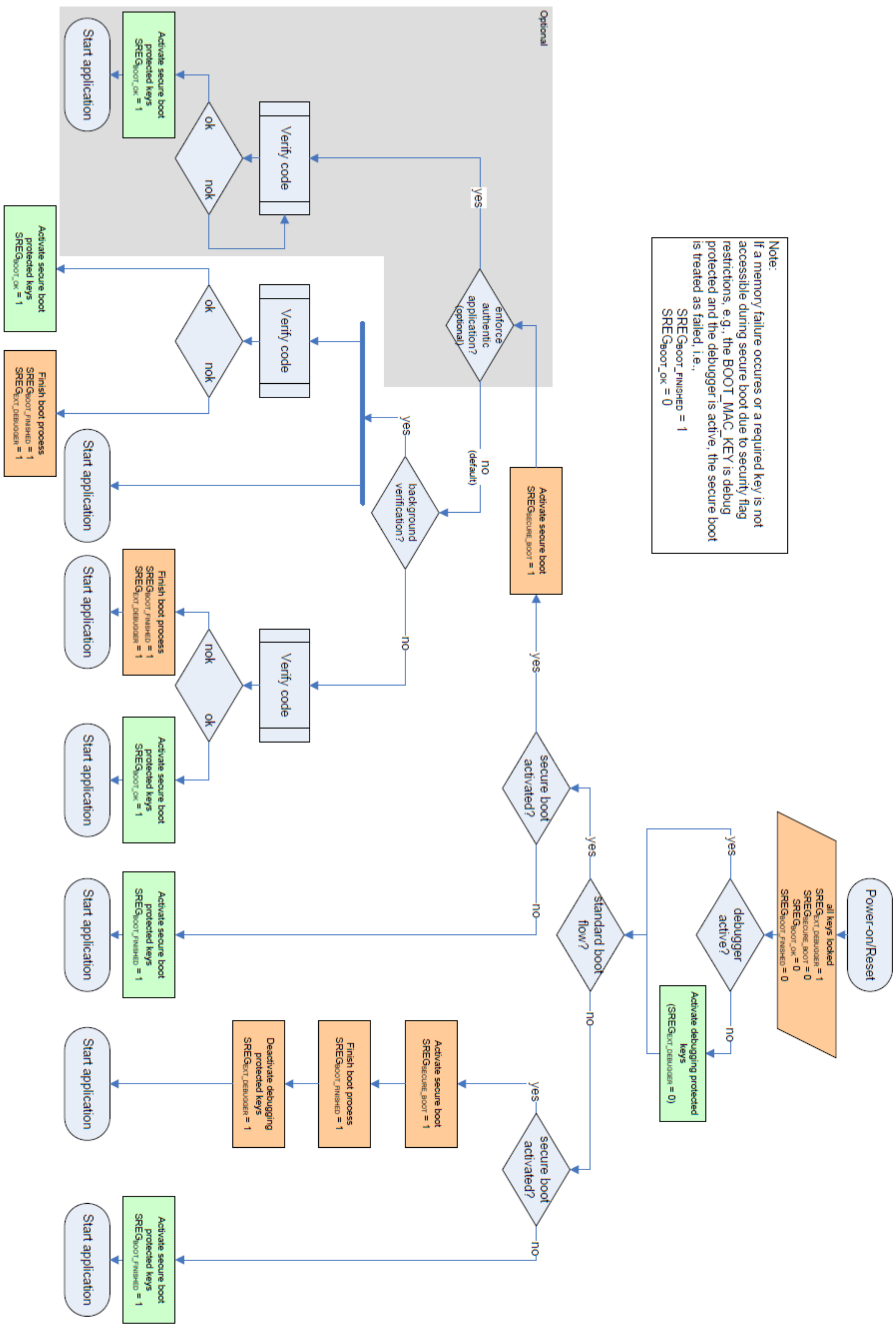

Figure 4.10: Boot flow of SHE enabled microcontrollers

4.11 Failure analysis of SHE/Resetting SHE

Note: The internal parts of SHE may only be accessible to failure analysis systems if they have been explicitly activated by SHE. Prior to $analyser$ *activation the analysis system has to be authenticated via the* $MASTER\_ECU\_KEY$ *and the internal memories of SHE have to be deleted. See function* CMD_DEBUG *in Chapter 4.7.19.*

In this chapter the term "failure analysis system" refers to low-level systems integrated by semiconductor manufacturers to analyse breakdowns. It does not include debugging of software applications.

SHE must be able to control the activation of internal failure analysis facilities. Regular debuggers that are available to developers (e.g. JTAG) may only cover the external interfaces of SHE.

To activate the internal failure analysis facilities, the following steps have to be executed:

1. Check if a write-protection bit is set, only proceed if no key is write-protected
2. A challenge-response protocol is used to unlock the internal failure analysis systems of SHE. The secret used in the debugging protocol is a key derived from MASTER_ECU_KEY with DEBUG_KEY_C [3]
3. After successful authentication all internal memories of SHE have to be deleted, except for SECRET_KEY, UID and PRNG_SEED
4. $SREG_{\mathtt{RND\_INIT}}$ has to be reset to '0'.
5. The debugging interface may be unlocked

The challenge-response protocol consists of the following steps. Prior to executing CMD_DEBUG, the random number generator has to be initialized to allow for the challenge-response protocol.

1. SHE generates a random number CHALLENGE
2. SHE derives a key K_{DEBUG} from MASTER_ECU_KEY and DEBUG_KEY_C
3. SHE sends CHALLENGE to the entity A requesting debugger access
4. A also derives $K_{\mathtt{DEBUG}}$
5. A calculates a MAC over UID and CHALLENGE
   $\mathtt{AUTHORIZATION} = \mathtt{CMAC}_{K_{\mathtt{DEBUG}}}(\mathtt{CHALLENGE} \mid \mathtt{UID})$
6. A sends AUTHORIZATION to SHE
7. SHE verifies AUTHORIZATION. In case of successful verification the internal memories have to be deleted before the debugger access is activated.

[3] See Chapter 4.12 for the values of the constants

The activation must only last until the next reset, i.e. the debugger has to be reactivated on every reset.

The activation must also work if the debugging or secure boot protection flag is set for MASTER_ECU_KEY.

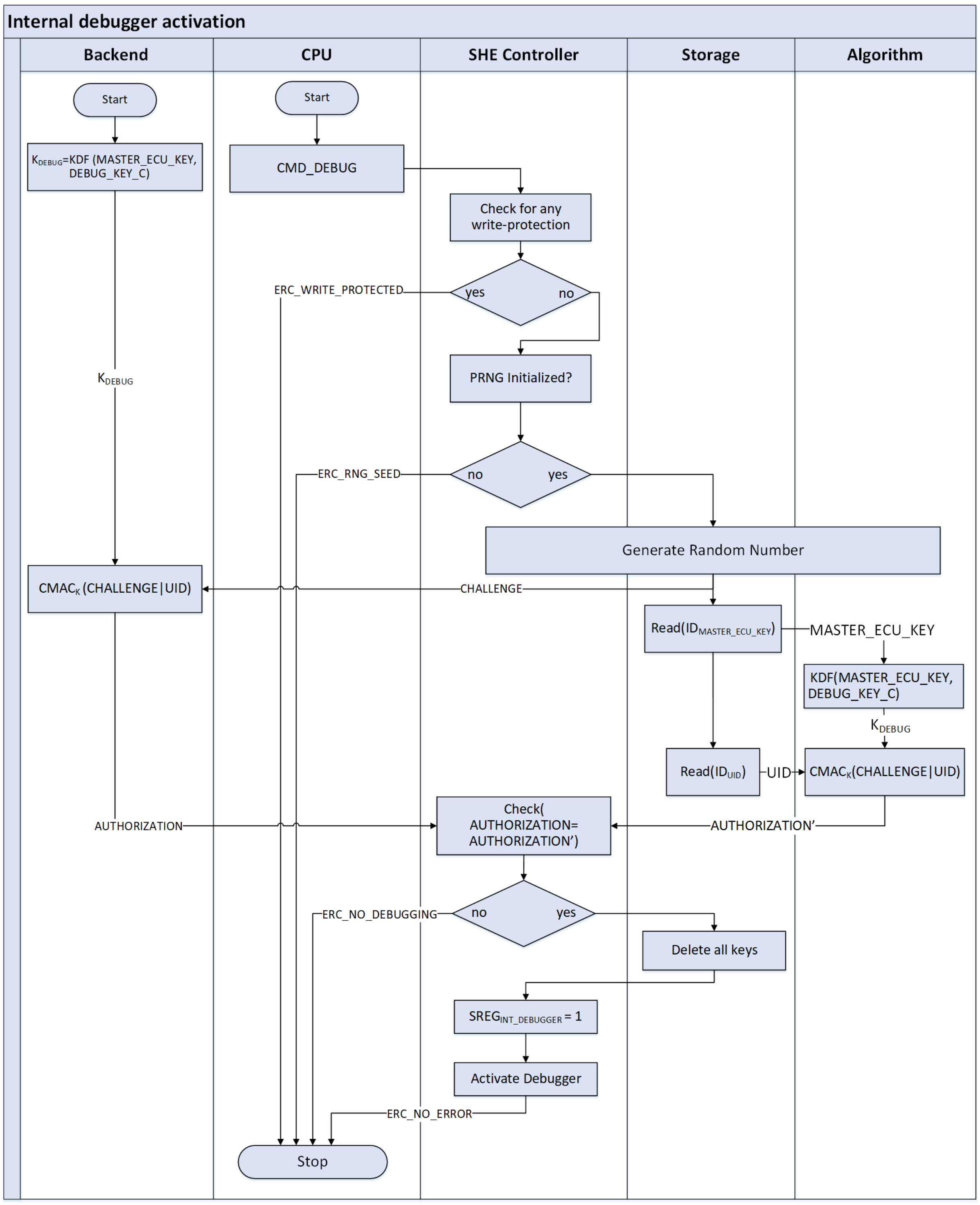

Figure 4.11: Activation of internal debugging facilities

4.12 Constants used within SHE

The constants are predefined to retain compatibility between different implementations of SHE. See Chapter 4.3.3.1 for details on how the constants are constructed.

Constant	Value
KEY_UPDATE_ENC_C	0x01015348 45008000 00000000 000000B0
KEY_UPDATE_MAC_C	0x01025348 45008000 00000000 000000B0
DEBUG_KEY_C	0x01035348 45008000 00000000 000000B0
PRNG_KEY_C	0x01045348 45008000 00000000 000000B0
PRNG_SEED_KEY_C	0x01055348 45008000 00000000 000000B0
PRNG_EXTENSION_C	0x80000000 00000000 00000000 00000100

Table 4.2: Constant values used within SHE

4.13 Examples and Test vectors

To check the correct implementation of SHE, the following sub chapters contain examples for every cryptographic function and protocol of SHE. The test vectors of the referenced algorithms, i.e. AES and CMAC, are taken from the cited specification documents and placed here for convenience.

4.13.1 AES-128, ECB mode

```
PLAINTEXT                00112233445566778899aabbccddeeff
KEY                      000102030405060708090a0b0c0d0e0f
CIPHERTEXT               69c4e0d86a7b0430d8cdb78070b4c55a
```

4.13.2 AES-128, CBC mode

4.13.2.1 encryption

```
KEY                      2b7e151628aed2a6abf7158809cf4f3c
IV                       000102030405060708090a0b0c0d0e0f
```

Block #1

```
PLAINTEXT                6bc1bee22e409f96e93d7e117393172a
AES INPUT                6bc0bce12a459991e134741a7f9e1925
AES OUTPUT               7649abac8119b246cee98e9b12e9197d
CIPHERTEXT               7649abac8119b246cee98e9b12e9197d
```

Block #2

```
PLAINTEXT                ae2d8a571e03ac9c9eb76fac45af8e51
AES INPUT                d86421fb9f1a1eda505ee1375746972c
AES OUTPUT               5086cb9b507219ee95db113a917678b2
CIPHERTEXT               5086cb9b507219ee95db113a917678b2
```

Block #3

```
PLAINTEXT                30c81c46a35ce411e5fbc1191a0a52ef
AES INPUT                604ed7ddf32efdff7020d0238b7c2a5d
AES OUTPUT               73bed6b8e3c1743b7116e69e22229516
CIPHERTEXT               73bed6b8e3c1743b7116e69e22229516
```

Block #4

```
PLAINTEXT                f69f2445df4f9b17ad2b417be66c3710
```

▽

△

AES INPUT	8521f2fd3c8eef2cdc3da7e5c44ea206
AES OUTPUT	3ff1caa1681fac09120eca307586e1a7
CIPHERTEXT	3ff1caa1681fac09120eca307586e1a7

4.13.2.2 decryption

KEY	2b7e151628aed2a6abf7158809cf4f3c
IV	000102030405060708090a0b0c0d0e0f

Block #1

CIPHERTEXT	7649abac8119b246cee98e9b12e9197d
AES INPUT	7649abac8119b246cee98e9b12e9197d
AES OUTPUT	6bc0bce12a459991e134741a7f9e1925
PLAINTEXT	6bc1bee22e409f96e93d7e117393172a

Block #2

CIPHERTEXT	5086cb9b507219ee95db113a917678b2
AES INPUT	5086cb9b507219ee95db113a917678b2
AES OUTPUT	d86421fb9f1a1eda505ee1375746972c
PLAINTEXT	ae2d8a571e03ac9c9eb76fac45af8e51

Block #3

CIPHERTEXT	73bed6b8e3c1743b7116e69e22229516
AES INPUT	73bed6b8e3c1743b7116e69e22229516
AES OUTPUT	604ed7ddf32efdff7020d0238b7c2a5d
PLAINTEXT	30c81c46a35ce411e5fbc1191a0a52ef

Block #4

CIPHERTEXT	3ff1caa1681fac09120eca307586e1a7
AES INPUT	3ff1caa1681fac09120eca307586e1a7
AES OUTPUT	8521f2fd3c8eef2cdc3da7e5c44ea206
PLAINTEXT	f69f2445df4f9b17ad2b417be66c3710

4.13.2.3 CMAC

KEY	2b7e151628aed2a6abf7158809cf4f3c

Subkey generation

$\mathtt{CIPH}_{K}(0^{128})$	7df76b0c1ab899b33e42f047b91b546f

▽

△

K1	fbeed618357133667c85e08f7236a8de
K2	f7ddac306ae266ccf90bc11ee46d513b
Example 1 (message length 128 bit)	
MESSAGE	6bc1bee22e409f96e93d7e117393172a
OUTPUT	070a16b46b4d4144f79bdd9dd04a287c
Example 2 (message length 320 bit)	
MESSAGE	6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51 30c81c46a35ce411
OUTPUT	dfa66747de9ae63030ca32611497c827

4.13.2.4 Miyaguchi-Preneel compression function

M	6bc1bee22e409f96e93d7e117393172a ae2d8a571e03ac9c9eb76fac45af8e51
PADDING	80000000000000000000000000000100
OUTPUT	c7277a0dc1fb853b5f4d9cbd26be40c6

4.13.2.5 Key derivation

K	000102030405060708090a0b0c0d0e0f
C	010153484500800000000000000000b0
$K_{Derived}$	118a46447a770d87828a69c222e2d17e

4.13.2.6 Pseudo random generation/Seed generation

SECRET_KEY	2b7e151628aed2a6abf7158809cf4f3c
PRNG_SEED	6bc1bee22e409f96e93d7e117393172a

4.13.2.7 Calculate new seed

PRNG_SEED_KEY	8abc8f6e2a8264fd38088be622ca0416
PRNG_SEED (new)	41f21213bca0434b3eb3bafcb0a19d74

4.13.2.8 Calculate new random value

PRNG_STATE	41f21213bca0434b3eb3bafcb0a19d74
PRNG_KEY	a1be019264992b2b725a4dd4c7767002
PRNG_STATE (new)	614aae8a7bb8fff31ac3230e6240506b

4.13.2.9 Extend seed

ENTROPY	ae2d8a571e03ac9c9eb76fac45af8e51
PRNG_SEED (ext)	7c92bea252d03015e4f5c2bca69a6f8a
PRNG_STATE (ext)	cf475ceb98f8ba6be1f55f97fdda9634

4.13.2.10 Memory update protocol

KEY_{NEW}	0f0e0d0c0b0a09080706050403020100
KEY_{AuthID}	000102030405060708090a0b0c0d0e0f
KEY_UPDATE_ENC_C	01015348450080000000000000000000b0
KEY_UPDATE_MAC_C	01025348450080000000000000000000b0
UID`	000000000000000000000000000001
ID	4 (KEY_1)
AuthID	1 (MASTER_ECU_KEY)
C_{ID}`	0000001
F_{ID}`	0
K1	118a46447a770d87828a69c222e2d17e
K2	2ebb2a3da62dbd64b18ba6493e9fbe22
M1	00000000000000000000000000000141
M2	2b111e2d93f486566bcbba1d7f7a9797
	c94643b050fc5d4d7de14cff682203c3
M3	b9d745e5ace7d41860bc63c2b9f5bb46

▽

△

K3	ed2de7864a47f6bac319a9dc496a788f
K4	ec9386fefaa1c598246144343de5f26a
M4	00000000000000000000000000000141
	b472e8d8727d70d57295e74849a27917
M5	820d8d95dc11b4668878160cb2a4e23e

4.13.2.11 Failure analysis of SHE/Resetting SHE

MASTER_ECU_KEY	000102030405060708090a0b0c0d0e0f
UID`	000000000000000000000000000001
DEBUG_KEY_C	010353484500800000000000000000b0
CHALLENGE	40abdeab16de77b9599964b3d2dd7261
K_{DEBUG}	1b5f959633c8c39ec42e965132bcec9b
AUTHORIZATION	953cb601d8ffa1954795fab3cad72c53

4.14 Overview Tables

	Write-protection	Secure boot failure	Debugger activation	Wildcard UID	Key usage	Plain key	Counter	Overall data [Bit]
MASTER_ECU_KEY	X	X	X	X			X	160
BOOT_MAC_KEY	X		X	X			X	159
BOOT_MAC	X		X	X			X	159
KEY_<n>	X	X	X	X	X		X	161
PRNG_SEED								128
RAM_KEY						X		129
PRNG_KEY								128
PRNG_STATE								128
SECRET_KEY		X (INH)	X (INH)					128
UID								120

Table 4.3: Information to be stored with keys

The following legend applies to the cells in Table 4.3:

INH SECRET_KEY inherits its protection flags from MASTER_ECU_KEY

	CMD_ENC_ECB	CMD_ENC_CBC	CMD_DEC_ECB	CMD_DEC_CBC	CMD_GENERATE_MAC	CMD_VERIFY_MAC	CMD_LOAD_KEY	CMD_LOAD_PLAIN_KEY	CMD_EXPORT_RAM_KEY	CMD_INIT_RNG	CMD_EXTEND_SEED	CMD_RND	CMD_SECURE_BOOT	CMD_BOOT_FAILURE	CMD_BOOT_OK	CMD_GET_STATUS	CMD_GET_ID	CMD_CANCEL	CMD_DEBUG
MAS-TER_ECU_KEY							X/o										X		X/o
BOOT_MAC_KEY						X	X/o						X						o
BOOT_MAC							o						X/o						o
KEY_<n>	X (DEP)	X (DEP)	X (DEP)	X (DEP)	X (DEP)	X (DEP)	X/o												o

▽

△

	CMD_ENC_ECB	CMD_ENC_CBC	CMD_DEC_ECB	CMD_DEC_CBC	CMD_GENERATE_MAC	CMD_VERIFY_MAC	CMD_LOAD_KEY	CMD_LOAD_PLAIN_KEY	CMD_EXPORT_RAM_KEY	CMD_INIT_RNG	CMD_EXTEND_SEED	CMD_RND	CMD_SECURE_BOOT	CMD_BOOT_FAILURE	CMD_BOOT_OK	CMD_GET_STATUS	CMD_GET_ID	CMD_CANCEL	CMD_DEBUG
PRNG_SEED										X/o	X/o								o
RAM_KEY	X	X	X	X	X	X	o	o	X										o
PRNG_KEY										o		X							X
PRNG_STATE										o	X/o	o							o
SECRET_KEY							X		X	X									
UID							X										X		

Table 4.4: Memory usage by functions (X: used by the function, o: can be modified by the function)

The following legend applies to the cells in Table 4.4:

DEP Depending on the key usage flag, see Chapter 4.4.1.5

	non-volatile					volatile			ROM		
Slot to update	MASTER_ECU_KEY	BOOT_MAC_KEY	BOOT_MAC	KEY_<n>	PRNG_SEED	RAM_KEY	PRNG_KEY	PRNG_STATE	SECRET_KEY	UID	(plaintext)
MASTER_ECU_KEY	X										
BOOT_MAC_KEY	X	X									
BOOT_MAC	X	X									
KEY_<n>	X			X							
PRNG_SEED											
RAM_KEY				X					X		X
PRNG_KEY											
PRNG_STATE											
SECRET_KEY											
UID											

Table 4.5: Memory update policy

	ERC_NO_ERROR	ERC_SEQUENCE_ERROR	ERC_KEY_NOT_AVAILABLE	ERC_KEY_INVALID	ERC_KEY_EMPTY	ERC_NO_SECURE_BOOT	ERC_KEY_WRITE_PROTECTED	ERC_KEY_UPDATE_ERROR	ERC_RNG_SEED	ERC_NO_DEBUGGING	ERC_BUSY	ERC_MEMORY_FAILURE	ERC_GENERAL_ERROR
CMD_ENC_ECB	X	X	X	X	X						X	X	(X)
CMD_ENC_CBC	X	X	X	X	X						X	X	(X)
CMD_DEC_ECB	X	X	X	X	X						X	X	(X)
CMD_DEC_CBC	X	X	X	X	X						X	X	(X)
CMD_GENERATE_MAC	X	X	X	X	X						X	X	(X)
CMD_VERIFY_MAC	X	X	X	X	X						X	X	(X)
CMD_LOAD_KEY	X	X	X	X	X		X	X			X	X	(X)
CMD_LOAD_PLAIN_KEY	X	X									X		(X)
CMD_EXPORT_RAM_KEY	X	X									X	X	(X)
CMD_INIT_RNG	X	X									X	X	(X)
CMD_EXTEND_SEED	X	X							X		X	X	(X)
CMD_RND	X	X							X		X	X	(X)
CMD_SECURE_BOOT	X	X				X					X	X	(X)
CMD_BOOT_FAILURE	X	X				X					X		(X)
CMD_BOOT_OK	X	X				X					X		(X)
CMD_GET_STATUS	X												(X)
CMD_GET_ID	X	X	X								X	X	(X)
CMD_CANCEL	X												(X)
CMD_DEBUG	X	X					X		X	X	X	X	(X)

Table 4.6: Error codes returned by the single functions

	Write-protection	Secure boot failure	Debugger activation	Wildcard UID	Key usage	Plain key	Empty state
CMD_ENC_ECB		X	X		X		X
CMD_ENC_CBC		X	X		X		X
CMD_DEC_ECB		X	X		X		X
CMD_DEC_CBC		X	X		X		X
CMD_GENERATE_MAC		X	X		X		X
CMD_VERIFY_MAC		X	X		X		X
CMD_LOAD_KEY	X	X (RAM)	X (RAM)	X			X (AUTH)
CMD_LOAD_PLAIN_KEY							
CMD_EXPORT_RAM_KEY		X	X			X	
CMD_INIT_RNG							
CMD_EXTEND_SEED							
CMD_RND							
CMD_SECURE_BOOT			X				X
CMD_BOOT_FAILURE							
CMD_BOOT_OK							
CMD_GET_STATUS							
CMD_GET_ID							X
CMD_CANCEL							
CMD_DEBUG	X						

Table 4.7: Functions affected by security bits

The following legend applies to the cells in Table 4.7:

RAM Only if ID = $\text{ID}_{\text{RAM\_KEY}}$, see Chapter 4.9.1

AUTH Empty state is checked for AuthID, see Chapter 4.9.1

A Appendix