em microelectronic

A COMPANY OF THE SWATCH GROUP

EM9305 CryptoLib

Release 2.1

EM

Oct 23, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

EM9305 provides several cryptographic libraries. It includes:

- **AES**
  - With all the key sizes
  - Either using an explicit key or a key stored in a key container
- **Several AES modes**
  - AES ECB
  - AES CBC
  - AES CTR
  - AES CMAC
  - AES CCM
  - AES GCM
- **Hash functions**
  - SHA-1
  - SHA-224
  - SHA-256
  - SHA-384
  - SHA-512
- **ECC functions based on the curve P-256**
  - ECDH
  - ECDSA signature verification
  - ECC Schnorr's external authentication protocol
  - ECC key management
- **Pseudo Random Number Generator**

The following picture illustrates the general architecture of the crypto library:

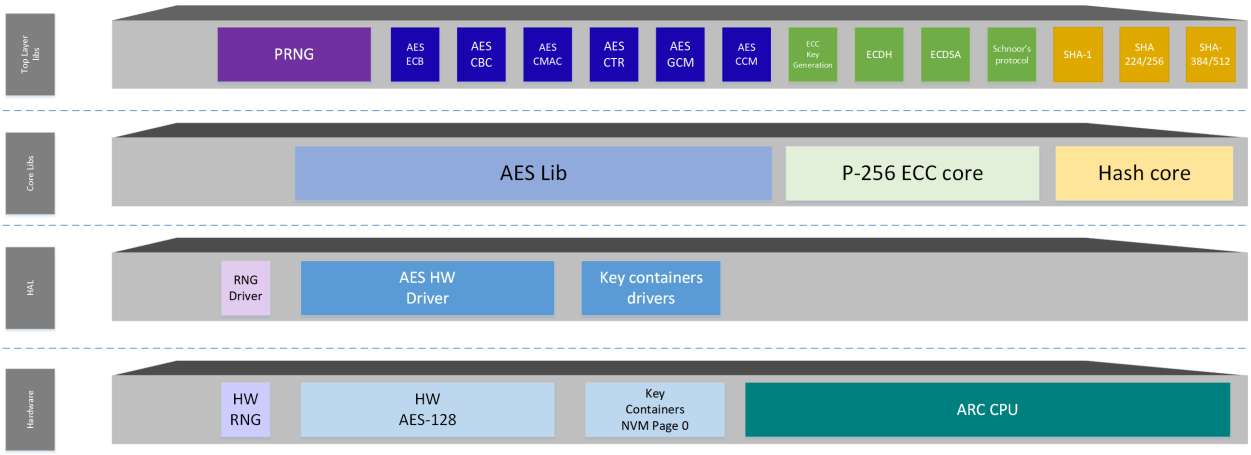


Figure: Cryptolib overall architecture

CHAPTER

TWO

AES (ONE BLOCK)

2.1 Bibliography

[1] FIPS197: Specification for the Advanced Encryption Standard (AES)

FIPS197: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

2.2 Reminders

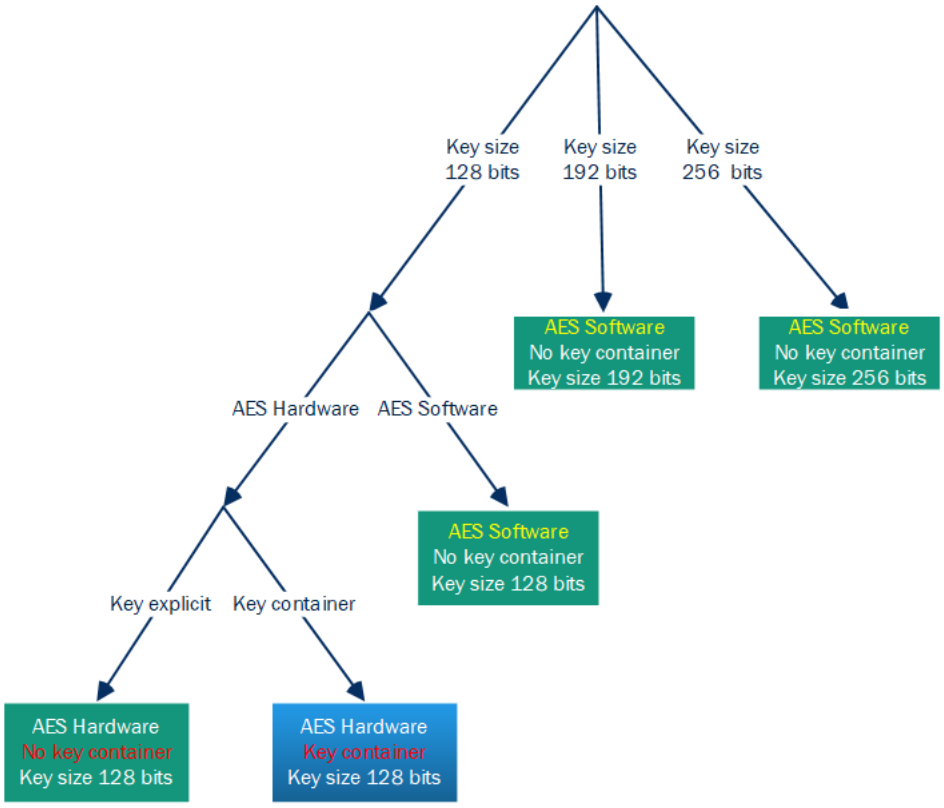
EM9305 embeds 3 AES:

1. A **hardware AES** dedicated to the link layer encryption (AES-CCM). This AES is not available for multi-usages.
2. A **hardware AES-128**, connected to the crypto key containers. This AES can be used for generic usages. This document, together with the crypto container documentation describes how to use this AES.
3. A **software AES** that supports the 3 key sizes: **128, 192, 256 bits**.

In this document, we describe the API that interfaces the AES-128(2) and the AES software(3).

> **Warning:** The API described in this document **does not** manage the execution of the hardware AES-128 with the **key containers**. AES key containers are described in the dedicated documentation: *AES Crypto Containers*

Next figure shows the various possibilities to execute AES algorithm. The green options are described in this chapter, while the blue one is described in the specific crypto container chapter.



2.3 Goal of the document

The goal of this document is to describe the functionality of the **AES** library.

AES library is located in ROM.

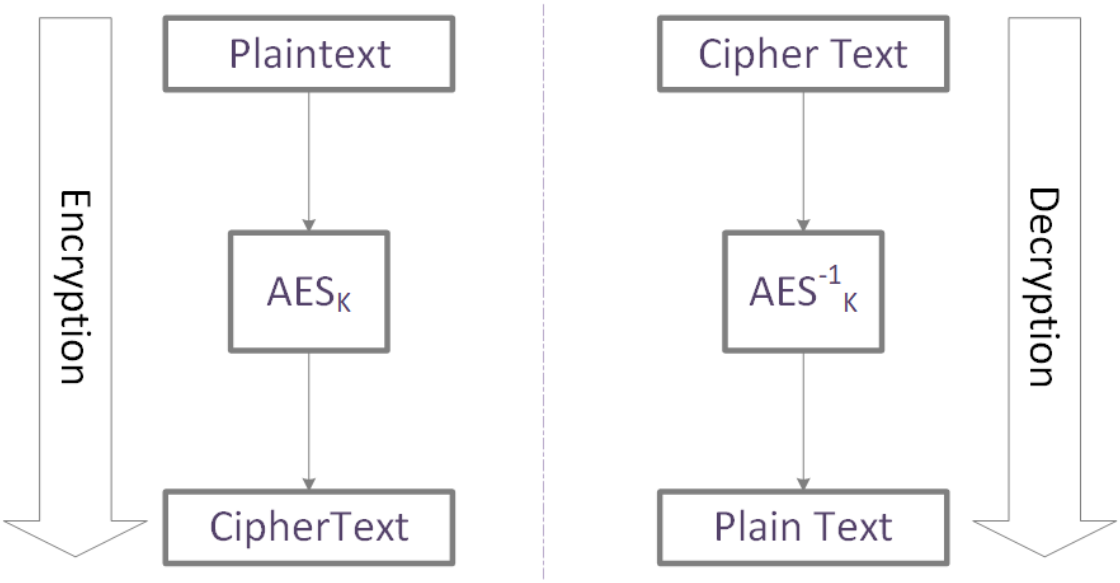
This document:

- describes the API, including the prototype, the parameters, the error codes etc…
- provides the performances of the function.

2.3.1 Supported features

AES library supports the encryption or the decryption of a **block only**.

Next figure shows the basic principle of AES.



Regarding the key size and the modes:

- **AES** library implements all the variants, i.e. it supports the three key sizes **128, 192 and 256 bits**. For a 128-bit key, the user can choose to use the AES hardware or the AES software. For 192-bit and 256-bit keys, only the software AES is supported.
- **AES** library supports both the **encryption** and **decryption** modes.

Next table summarizes the capabilities of the AES library.

Mode	Key size in bits	AES Hardware	AES Software
Encryption	128	Supported	Supported
Encryption	192	**Not Supported**	Supported
Encryption	256	**Not Supported**	Supported
Decryption	128	Supported	Supported
Decryption	192	**Not Supported**	Supported
Decryption	256	**Not Supported**	Supported

2.3.2 AES chaining modes and MAC

EM9305 embeds several AES chaining modes and MAC computations. It includes AES ECB, CBC, CTR, GCM, CCM and CMAC. Those modes are built on top of the AES library. For further details on those modes, please refer to the dedicated chapters. See *AES Electronic Code Book mode (ECB)*, *AES Cipher Block Chaining mode (CBC)*, *AES Counter mode(CTR)*, *AES CCM*, *AES GCM* and *AES CMAC*

2.4 API

2.4.1 Purpose

The **AES** library contains a unique function "AES". This function computes AES on a block of 16 bytes.

- In encryption mode, it encrypts the plain text to generate a 16-byte cipher text.
- In decryption mode, it decrypts a cipher text to retrieve the 16-byte plain text.

The input block **must always be 16 bytes long**.

The library does not support any padding schemes, or chaining options.

Unlike the input block, the key size can be chosen by the user among the three lengths specified by AES. To be more precise, the key size can be:

- 16 bytes (128 bits)
- 24 bytes (192 bits)
- 32 bytes (256 bits)

2.4.2 Prototype

AES_Lib_error_t **AES**(*AES_Select_t* AESSelect, uint8_t *Key, uint8_t *Message, uint8_t *Result, *AES_key_size_bit_t* KeyLength, *AES_OpMode_t* Mode)

Perform AES in encryption or decryption mode with 128,192 or 256-bit key.

Parameters

- **AESSelect** – **[in]** Select the AES to execute. Either the AES Hardware or the AES Software
- **Key** – **[in]** 16, 24 or 32-byte key array according to the key size
- **Message** – **[in]** 16-byte message to encrypt or decrypt according to the mode
- **Result** – **[out]** 16-byte result
- **KeyLength** – **[in]** Length of the key in bits: 128, 192 or 256
- **Mode** – **[in]** Mode of operation. Either AES_ENCRYPTION or AES_DECRYPTION

Return values

- **AES_SUCCESS** – Successful computation
- **AES_INCORRECT_RESULT_POINTER** – Result pointer not initialized
- **AES_INCORRECT_KEY_LENGTH** – Incorrect key length. It should be 128,192 or 256
- **AES_INCORRECT_AES_TYPE** – AES is either hardware or software
- **AES_KEY_SIZE_NOT_SUPPORTED_WITH_HARDWARE** – AES Hardware is only support for 128 bit keys
- **AES_INCORRECT_MODE** – Mode is either encryption or decryption

Returns Error status

2.4.3 Enumerations

2.4.3.1 AES type

enum **AES_Select_t**

Select the AES between AES hardware(128 bit only) and the AES software(all key sizes). This module only concerns the encryption or decryption of a 16-byte block. See documentation of the AES container for implicit keys.

Values:

enumerator **AES_HARDWARE**

AES HARDWARE (only valid for 128-bit keys)

enumerator **AES_SOFTWARE**

AES SOFTWARE (valid for all key sizes)

2.4.3.2 AES mode

enum **AES_OpMode_t**

Select the mode of operation of AES.

Values:

enumerator **AES_ENCRYPTION**

AES encryption mode.

enumerator **AES_DECRYPTION**

AES decryption mode.

2.4.3.3 AES Key size in bits

enum **AES_key_size_bit_t**

Select the AES key size in bits.

Values:

enumerator **AES_KEY_128**

128-bit key size

enumerator **AES_KEY_192**

192-bit key size

enumerator **AES_KEY_256**

256-bit key size

2.4.3.4 AES Key size in bytes

enum **AES_key_size_byte_t**

AES key sizes in byte.

Values:

enumerator **AES_BYTE_SIZE_KEY128**

128-bit key size(16 bytes)

enumerator **AES_BYTE_SIZE_KEY192**

192-bit key size(24 bytes)

enumerator **AES_BYTE_SIZE_KEY256**

256-bit key size(32 bytes)

2.4.3.5 AES error status

enum **AES_Lib_error_t**

Error status words for AES.

Values:

enumerator **AES_SUCCESS**

AES computation successful.

enumerator **AES_KEY_SIZE_NOT_SUPPORTED_WITH_HARDWARE**

AES Hardware only supports the key size 128-bit.

enumerator **AES_HW_ERROR**

An error occured in the AES hardware.

enumerator **AES_INCORRECT_RESULT_POINTER**

Result pointer is null.

enumerator **AES_INCORRECT_AES_TYPE**

AES type Hardware or Software is not correct.

enumerator **AES_INCORRECT_KEY_LENGTH**

AES key length is not correct.

enumerator **AES_INCORRECT_MODE**

AES mode is not correct.

2.4.4 Defines

AES_BLOCK_SIZE_BYTE

Size of an AES block in byte.

2.4.5 Return values

Type	Description	OK \ NOK
AES_SUCCESS	AES computation successful.	OK
AES_KEY_SIZE_NOT_SUPPORTED_WITH_HARDWARE	AES Hardware only supports the key size 128-bit.	NOK
AES_HW_ERROR	An error occurred in the AES hardware.	NOK
AES_INCORRECT_RESULT_POINTER	Result pointer is null.	NOK
AES_INCORRECT_AES_TYPE	AES type Hardware or Software is not correct.	NOK
AES_INCORRECT_KEY_LENGTH	AES key length is not correct.	NOK
AES_INCORRECT_MODE	AES mode is not correct.	NOK

2.5 Performances

2.5.1 Library location

The lib is located in ROM.

2.5.2 Code size

Size in bytes
3628 bytes in ROM

2.5.3 RAM

Size in bytes
No usage of global RAM

2.5.4 Stack

Size in bytes
Approximately 100 bytes

2.5.5 Execution time

The execution time of AES function depends on the key size, the mode and the underlying algorithm. On the other hand, for a given variant, the execution time is constant. There is no variability in the computation. Next tables show the number of cycles and the execution time at 48 Mhz.

2.5.5.1 AES Hardware

Mode	Key size in bits	Time in us at 48Mhz
Encryption	128	Approximately 5us
Decryption	128	Approximately 5us

2.5.5.2 AES Software

Mode	Key size in bits	Number of cycles	Time in us at 48Mhz
Encryption	128	5283	110
Encryption	192	6547	136
Encryption	256	7414	154
Decryption	128	6668	138
Decryption	192	8478	176
Decryption	256	9767	203

2.6 Dependencies

AES library does not have any dependency.

2.7 Example

Next example shows:

- how to use the library function with various key sizes,
- how to select the AES hardware or the AES software when relevant,
- how to select the mode encryption or decryption.

```
//*****************************************************************************************
//
// FILE : ExampleAES.c
//
```

(continues on next page)

(continued from previous page)

```
//Provide examples of use of AES
//*****************************************************************************************/
#include <stdint.h>
#include "AES.h"

//=========================================================================================
//                      TEST SUITE
//=========================================================================================

//*****************************************************************************************
//Test patterns
//*****************************************************************************************/
#define NB_TEST     6
#define TEST_OK     0
#define TEST_NOK    1

typedef struct {
        const uint8_t *Message;
        const uint8_t *Result;
        const uint16_t KeyLength;
        const uint8_t *key;
        const uint8_t mode;
} Patterns;

//keys
const uint8_t key128[16] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };
const uint8_t key192[24] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,
                0x14, 0x15, 0x16, 0x17 };
const uint8_t key256[32] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,
                0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f };

//message
const uint8_t message[16] = { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
                0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff };

//result encryption = input for decryption
const uint8_t result128[16] = { 0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30,
                0xd8, 0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a };
const uint8_t result192[16] = { 0xdd, 0xa9, 0x7c, 0xa4, 0x86, 0x4c, 0xdf, 0xe0,
                0x6e, 0xaf, 0x70, 0xa0, 0xec, 0x0d, 0x71, 0x91 };
const uint8_t result256[16] = { 0x8e, 0xa2, 0xb7, 0xca, 0x51, 0x67, 0x45, 0xbf,
                0xea, 0xfc, 0x49, 0x90, 0x4b, 0x49, 0x60, 0x89 };

Patterns Pat[NB_TEST] = { { message, result128, AES_KEY_128, key128,
                AES_ENCRYPTION }, { message, result192, AES_KEY_192, key192,
                AES_ENCRYPTION }, { message, result256, AES_KEY_256, key256,
                AES_ENCRYPTION }, { result128, message, AES_KEY_128, key128,
                AES_DECRYPTION }, { result192, message, AES_KEY_192, key192,
                AES_DECRYPTION }, { result256, message, AES_KEY_256, key256,
```

(continues on next page)

(continued from previous page)

```
                AES_DECRYPTION }, };

/*****************************************************************************************
 Function: uint8_t ExampleAES(void)

 - Shows example of use of the AES function
 - Validate the above patterns
 - Count the number of errors
 -
 Return:
 0 if everything is ok
 1 otherwise

 *****************************************************************************************/
uint8_t ExampleAES(void) {
        //Number of errors
        uint16_t Error = 0;
        //loops
        uint8_t u8Loop1;
        uint8_t u8Loop2;
        //result variable
        uint8_t ResultAES[16];
        AES_Select_t AESHWorSW;
        AES_Lib_error_t sw;

        //First we run the tests with the AES Software
        AESHWorSW = AES_SOFTWARE;
        for (u8Loop1 = 0; u8Loop1 < NB_TEST; u8Loop1++) {
                //Perform AES with the defined patterns
                sw = AES(AESHWorSW, (uint8_t*) Pat[u8Loop1].key,
                                (uint8_t*) Pat[u8Loop1].Message, ResultAES,
                                Pat[u8Loop1].KeyLength, Pat[u8Loop1].mode);
                if (sw != AES_SUCCESS)
                        Error++;
                //Check if the result is correct
                for (u8Loop2 = 0; u8Loop2 < 16; u8Loop2++) {
                        if (ResultAES[u8Loop2] != Pat[u8Loop1].Result[u8Loop2])
                                Error++;
                }
        }

        //Second we run the tests with the AES Hardware
        AESHWorSW = AES_HARDWARE;
        for (u8Loop1 = 0; u8Loop1 < NB_TEST; u8Loop1++) {
                //Perform AES with the defined patterns
                sw = AES(AESHWorSW, (uint8_t*) Pat[u8Loop1].key,
                                (uint8_t*) Pat[u8Loop1].Message, ResultAES,
                                Pat[u8Loop1].KeyLength, Pat[u8Loop1].mode);
                if (Pat[u8Loop1].KeyLength != AES_KEY_128) {
                        if (sw != AES_KEY_SIZE_NOT_SUPPORTED_WITH_HARDWARE)
                                Error++;
                } else {
```

(continues on next page)

(continued from previous page)

```
                        if (sw != AES_SUCCESS)
                                Error++;

                        //Check if the result is correct
                        for (u8Loop2 = 0; u8Loop2 < 16; u8Loop2++) {
                                if (ResultAES[u8Loop2] != Pat[u8Loop1].Result[u8Loop2])
                                        Error++;
                        }
                }
        }
        if (Error != 0)
                return (TEST_NOK);
        else
                return (TEST_OK);
}
```

CHAPTER

THREE

AES CRYPTO CONTAINERS

3.1 Bibliography

[1] FIPS197: Specification for the Advanced Encryption standard(AES)

FIPS197: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

3.2 Goal of the document

The purpose of this document is to describe the concept of the AES crypto containers. It describes:

- The structure of the crypto containers
- The software APIs that ease the usage of the crypto containers.

3.3 Concept

The AES crypto containers aim at **storing securely AES-128 keys** in the Non-Volatile Memory.

The main security principle is that:

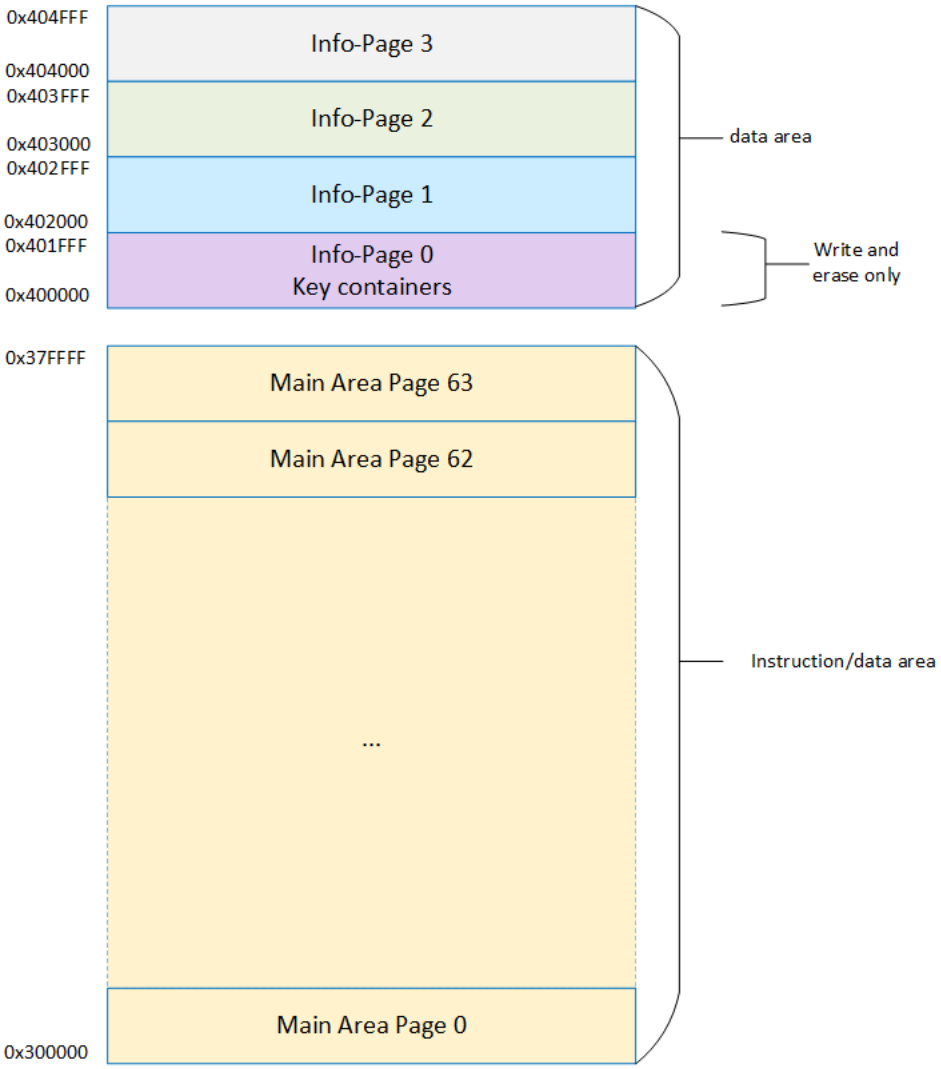
- **The CPU can write** inside the key containers
- **The CPU cannot read** the key containers.
- **Only the AES hardware** can execute the keys located in the key containers.

3.3.1 NVM structure (reminder)

As a reminder, the NVM of EM9305 is composed of:

- 64 pages of 8KBytes in the main area, for a total of 512 KBytes. The main area aims at storing code and data.
- 4 pages of 8KBytes each, referred as Info Pages. Those pages are dedicated for data and especially for the product configuration data.

Next figure shows the general NVM structure.



The Info-Page 0 contains the key containers. Each container is 256 bits long, so that the Info-Page0 contains 256 key containers.

3.3.2 Crypto container structure

Next figure shows the structure of the InfoPage0 and illustrates that up to 256 AES-128 keys can be stored in the key containers.

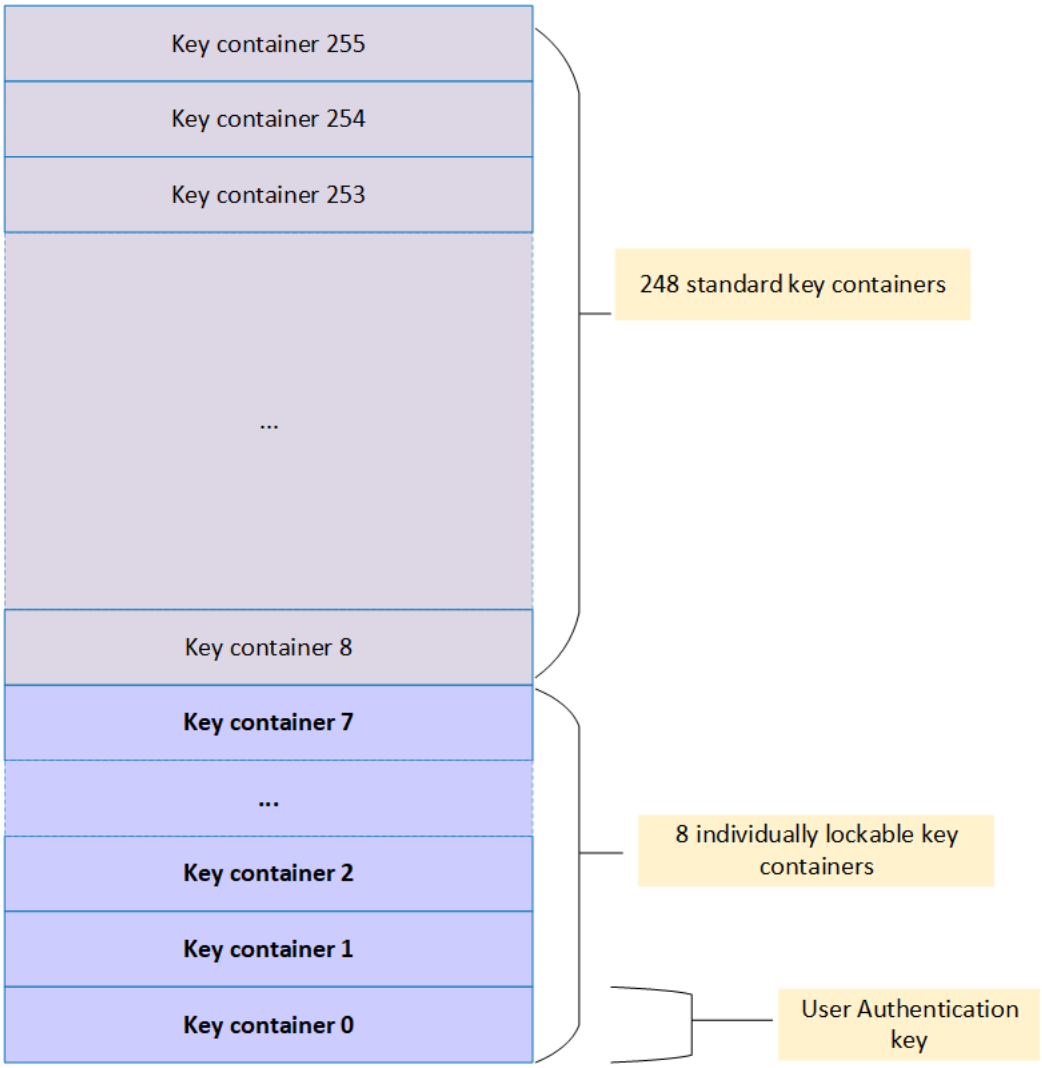


Fig: Structure of InfoPage0

3.3.3 Security properties

The security properties of the key containers InfoPage0 are the following:

- Unless locked against write, the key containers can be written by the CPU.
- The key containers content can **never be read by the CPU**.
- The key value and the attributes are directly connected to the hardware AES. Only the hardware AES can fetch data in a key container. The hardware AES can then be executed with the given key.
- A key stored in a key container can be typed to limit the use of this key to AES encryption or AES decryption.
- A key stored in a key container can be invalidated.
- For the 8 first keys (0..7), it is possible to prevent the key container to be written.
- The full page InfoPage0 can be locked either against write, against erase or against write & erase.

Next figure displays the general crypto containers architecture.

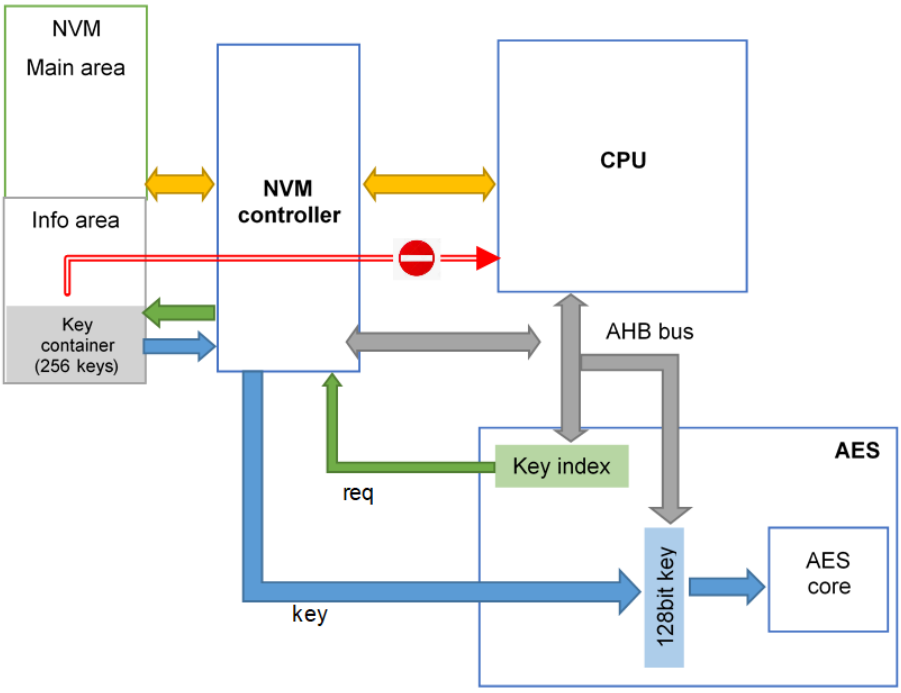


Fig: Illustration of the crypto containers architecture. The CPU can write into the key containers through the NVM controller. Key containers reading from the CPU is forbidden. The AES hardware is directly connected to the key containers and can be executed with a key located in the key containers.

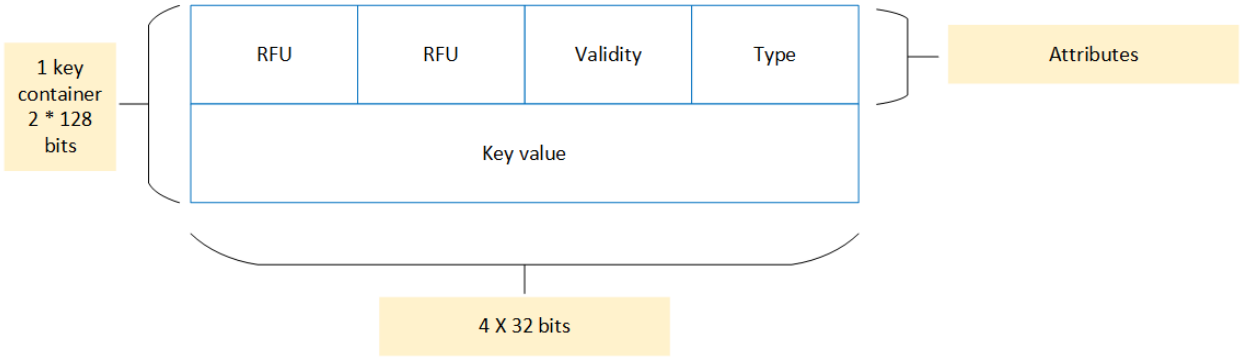
3.3.4 Key container structure

Each container is composed of 2 words of 128 bits:

- A first word stores an AES-128 bits **key value**
- A second word stores the **key attributes**

The key attributes are the following ones:

- Type:
  - The key can be **ENCRYPTION ONLY**
  - The key can be **DECRYPTION ONLY**
  - The key can be **ENCRYPTION and DECRYPTION**
- Validity:
  - The key can be **Valid**.
  - The key can be **Invalid**
- Two words are reserved for further usage.



If the key is typed ENCRYPTION ONLY, trying to operate AES in decryption mode with this key would be rejected. Similarly, if the key is typed DECRYPTION ONLY, trying to operate AES in encryption mode with this key would be rejected.

Initially, a key should be set as **valid**. Then, in the life cycle of the product, it might be necessary to disable or kill a key. The key should be set as **invalid**. Once the key is invalidated, any trial to execute AES with this key will be rejected.

3.3.5 "Reading a key"

As mentioned above, the InfoPage0 **cannot** be read by the CPU.

It implies that the key containers (key value, key attributes) cannot be read by the CPU. To execute AES with the key stored in a key container, **the key is referred with key ID**.

For a given key ID, the hardware AES, which is directly connected to the InfoPage0, fetches the referred key and loads it into core registers. The hardware AES also verifies the attributes and the consistency of the attributes. Note that the key is managed fully by the hardware digital logic.

3.3.6 Locks

Several locks are implemented:

- The entire InfoPage0 can be locked **against write**.
- The entire InfoPage0 can be locked **against erase**.
- The entire InfoPage0 can be locked **against write and erase**.
- The 8 first keys, with ID in range [0..7] can be **individually** locked against **write**.

3.3.7 Granularity aspects

InfoPage0 is a NVM page as the other ones. It implies some granularity limitations:

- Unless the page is locked against "erase", InfoPage0 can be erased. Nevertheless, it is not possible to erase parts of the page. Only the complete page can be erased.
- Unless the page is locked against "write" (or the specific key container (0..7) is locked), it is possible to write in InfoPage0 32-bit word by 32-bit word. Nevertheless:
  - A word can be written if and only if its initial content is erased.
  - **It implies that to update a key container, one needs to:**
    - Erase the complete InfoPage0

- Write the new data in the key container.

Note: The **validity** field can be written without erasing the page. Indeed, the "**valid**" value is set to the erased state 0xFFFFFFFF. Invalidation of a key is possible.

Warning: To update a key container and to avoid losing the content of the other containers, one may want to make a backup of the full InfoPage0 before the complete erasing of the Page0.

Nonetheless, the InfoPage0 is **never** readable by principle. Therefore, the back-up is impossible.

3.3.8 How do I know a key was correctly written?

Often, one wants to verify that the key writing was correctly executed. The usual process is to read back the data written in the NVM. Nevertheless, in this case, since the InfoPage0 is **never** readable, the read-back and comparison is impossible.

If the integrity of the key has to be verified, the recommendation is to implement a KCV (Key Check Value). The principle is that:

- One pre-computes the result of the encryption\decryption of a given message, typically the null message.
- One keeps typically the 2 first bytes of the previous result. We denote by KCV these 2 bytes.
- When one wants to check that the key has been correctly set up in the key containers, he executes AES with this key and the null message. Then, he compares the first bytes of the result with the reference KCV. A match means that the key has been correctly inserted.

3.3.9 How do I know a key container is used?

Since the InfoPage0 is not readable, it is not easily possible to know if a key container is used or not. There are 2 possible strategies:

- The application logs which key container is used.
- In case of doubt, one can execute the AES with the key indexed by the given ID. If no key is present, the AES execution fails and the API returns an error (see the API chapter).

3.3.10 Key container 0

The key container 0 is reserved to store the *USER configuration mode authentication key*. The USER configuration is performed with a challenge-response protocol with AES as underlying algorithm. See the configurations modes documentation for further details.

3.4 AES chaining modes and MAC

EM9305 embeds a cryptographic toolbox that includes various AES modes and AES-based MAC computations.

The toolbox includes:

- AES ECB mode
- AES CBC mode
- AES CTR mode
- AES CMAC
- AES CCM mode
- AES GCM mode

All those libs interface the crypto containers. It is possible to execute those modes and MAC algorithms with a key contained in a crypto container as long as the key is an AES-128 long key.

Those libs also permit to provide:

- an explicit key given by its value
- an implicit key given by its ID

For further details, please referred to the dedicated documentations. See *AES Electronic Code Book mode (ECB)*, *AES Cipher Block Chaining mode (CBC)*, *AES Counter mode(CTR)*, *AES CCM*, *AES GCM* and *AES CMAC*

3.5 Typical usages of key containers

Key containers purpose is to secure the AES keys.

Nonetheless, because of the difficulty to update a key value, the key containers are mostly devoted to store **permanent or master keys**.

Temporary or session keys can be stored in the key containers. Nonetheless:

- Typically those keys are secondary assets and therefore have less value than master keys. They may reside in RAM.
- **Due to the update limitation, the session keys cannot be easily updated.**
  - Either the InfoPage0 is completely erased, in order to update the specific key. One will take care not to lose the other potential keys located in the key containers.
  - Or a cycling mechanism is put in place. The keys are invalidated when obsolete. The new session key is written in the next free key container. When all the key containers are full, the InfoPage0 is erased.

Warning: The AES key containers are directly connected to the AES CPU only. It implies that:

- Key containers can ONLY store AES-128 keys. Keys of 192 bits or 256 bits are not supported by the key containers.
- The AES radio (hardware AES-CCM) used for the link layer payload protection cannot access the key containers. Keys for the link layer encryption shall not be stored in the key containers.
- Other algorithms keys, such ECC keys shall not be stored in the key containers.

3.6 APIs

3.6.1 Key type enumerations

enum **AES_KeyContainerType_t**

AES key container types(enc/dec,enc,dec)

Values:

enumerator **AES_HW_ENC_DEC**

Key type is encryption and decryption.

enumerator **AES_HW_DEC_ONLY**

Key type is only decryption.

enumerator **AES_HW_ENC_ONLY**

Key type is only encrytion.

3.6.2 Validity enumerations

enum **AES_KeyContainerValidity_t**

AES key container validity.

Values:

enumerator **AES_HW_KEY_INVALID**

Key is invalid.

enumerator **AES_HW_KEY_VALID**

Key is valid.

3.6.3 Mode of operations

enum **AES_Mode_t**

AES modes of operation.

Values:

enumerator **AES_MODE_ENCRYPT**

Encrypt the given data.

enumerator **AES_MODE_DECRYPT**

Decrypt the given data.

3.6.4 Error status

enum **AES_Error_t**

AES Error status words.

Values:

enumerator **AES_STATUS_SUCCESS**

No Error- Sucesss.

enumerator **AES_STATUS_ERROR**

Unknown error.

enumerator **AES_STATUS_WRONG_PARAM**

Incorrect conditions. Incorrect parameters.

enumerator **AES_STATUS_NVM_ERROR**

Error writing in nvm. Key or the page may be locked.

enumerator **AES_STATUS_NOT_LOCKABLE**

Key is not lockable. Only key ID in [0..7] can be locked.

enumerator **AES_STATUS_KEY_LOCKED**

Key is locked.

3.6.5 Set key

3.6.5.1 Purpose

The function AES_SetKeyContainer permits to set the key and its attributes in a key container.

3.6.5.2 Prototype

AES_Error_t **AES_SetKeyContainer**(const uint32_t key[AES_BLOCK_WORD_SIZE], uint8_t keyIndex, *AES_KeyContainerType_t* keyType)

Set a 128-bit AES key in a key container.

Parameters

- **key** – 16-byte key to set
- **keyIndex** – keyID to store the key [0..255]
- **keyType** – Type of the key. Encryption/decryption, encryption only, or decryption only

Returns status of the operation

3.6.5.3 Parameters

key : 16 bytes (4 times 32-bit words) representing the key value.

keyIndex: ID of the key container to set.

keyType: type of the key among the types *Key type enumerations*

3.6.5.4 Return values

Type	Description	OK \ NOK
AES_STATUS_SUCCESS	Key container correctly set.	OK
AES_STATUS_WRONG_PARAM	Incorrect key type	NOK
AES_STATUS_KEY_LOCKED	The key container is locked	NOK
AES_STATUS_NVM_ERROR	Error writing in NVM	NOK

3.6.5.5 Remarks

- When set, the key is automatically set as valid.

Warning: Trying to write a key container that was not previously erased may lead to errors that are not detected. Ensure that Info-Page0 was erased before writing any key container.

3.6.6 AES_ProcessBlockKeyContainer

3.6.6.1 Purpose

The function AES_ProcessBlockKeyContainer computes AES with the given key, with the given data and in the required mode,

3.6.6.2 Prototype

AES_Error_t **AES_ProcessBlockKeyContainer**(uint8_t keyIndex, const void *pInData, const void *pOutData, *AES_Mode_t* mode)

Execute AES-128 with a implicit key given by its ID.

Parameters

- **keyIndex** – keyID of the key to use
- **pInData** – 16-byte data to encrypt or decrypt
- **pOutData** – 16-byte result data
- **mode** – Encryption or decryption mode

Returns status of the operation

3.6.6.3 Parameters

keyIndex: ID of the key container to set.

pInData : Data to encrypt or to decrypt

pOutData: Result of the operation

mode : Either encryption or decryption as defined in *Mode of operations*

3.6.6.4 Return values

Type	Description	OK \ NOK
AES_STATUS_SUCCESS	Successful computation.	OK
AES_STATUS_WRONG_PARAM	Incorrect mode required	NOK
AES_STATUS_KEY_LOCKED	The key container is locked	NOK
AES_STATUS_WRONG_PARAM	No key present at this ID	NOK
AES_STATUS_WRONG_PARAM	Key is invalidated	NOK
AES_STATUS_WRONG_PARAM	Key type is not compatible with the required mode	NOK
AES_STATUS_WRONG_PARAM	Key is locked	NOK
AES_STATUS_ERROR	Error in AES	NOK

3.6.7 AES_InvalidateKeyContainer

3.6.7.1 Purpose

The function AES_InvalidateKeyContainer invalidates a given key.

> **Warning:** The key is invalidated until the InfoPage0 is erased.

3.6.7.2 Prototype

AES_Error_t **AES_InvalidateKeyContainer**(uint8_t keyIndex)

Invalidate a key container.

Parameters **keyIndex** – keyID of the key to invalidate [0..255]

Returns status of the operation

3.6.7.3 Parameters

keyIndex: ID of the key container to invalidate.

3.6.7.4 Return values

Type	Description	OK \ NOK
AES_STATUS_SUCCESS	Successful key invalidation	OK
AES_STATUS_NVM_ERROR	Error in NVM writing	NOK

3.6.8 AES_LockKeyContainer

3.6.8.1 Purpose

The function AES_LockKeyContainer protects the given key against write.

3.6.8.2 Prototype

AES_Error_t **AES_LockKeyContainer**(uint8_t keyIndex)

Write lock of a given key container.

Parameters **keyIndex** – keyID of the key to lock

3.6.8.3 Parameters

keyIndex: ID of the key container to lock. It must be in the range [0..7].

3.6.8.4 Return values

Type	Description	OK \ NOK
AES_STATUS_SUCCESS	Key successfully locked	OK
AES_STATUS_NOT_LOCKABLE	Index is not in the correct range	NOK

3.6.8.5 Remark

Warning: The lock information is normally stored in 2 different locations:

- In the list of locks located in InfoPage2 and InfoPage3. Those locks are loaded into the PML registers at boot time, after a POR.
- In dedicated PML registers. PML registers are semi-persistent. Their values are not lost when going to sleep or deep sleep mode.

When one tries to execute AES with a given key, only the PML registers are evaluated. The content of the InfoPage2 and InfoPage3 is **not** loaded.

This function only sets the PML registers.

It results that the locking is valid only until the next reset\POR. If one wants to permanently lock a key container, he shall configure accordingly the InfoPage2/3.

3.6.9 AES_KeyContainersEraseLock

3.6.9.1 Purpose

The function AES_KeyContainersEraseLock prevents the InfoPage0 to be erased.

3.6.9.2 Prototype

AES_Error_t **AES_KeyContainersEraseLock**(void)

Lock the key containers page(Info page) 0 against erase.

Returns status of the operation

3.6.10 Return values

Type	Description	OK \ NOK
AES_STATUS_SUCCESS	InfoPage0 successfully locked against erase	OK

3.6.10.1 Remark

Warning: The lock information is normally stored in 2 different locations:

- In the list of locks located in InfoPage2 and InfoPage3. Those locks are loaded into the PML registers at boot time, after a POR.
- In dedicated PML registers. PML registers are semi-persistent. Their values are not lost when going to sleep or deep sleep mode.

When one tries to erase the InfoPage0, only the PML registers are evaluated. The content of the InfoPage2 and InfoPage3 is **not** loaded.

This function only sets the PML registers.

It results that the locking is valid only until the next reset\POR. If one wants to permanently lock the InfoPage0, he shall configure accordingly the InfoPage2\3.

3.6.11 AES_KeyContainersWriteLock

3.6.11.1 Purpose

The function AES_KeyContainersWriteLock prevents the InfoPage0 to be written.

3.6.11.2 Prototype

AES_Error_t **AES_KeyContainersWriteLock**(void)

Lock the key containers page(Info page) 0 against write.

Returns status of the operation

3.6.12 Return values

Type	Description	OK \ NOK
AES_STATUS_SUCCESS	InfoPage0 successfully locked against write	OK

3.6.13 Remark

Warning: The lock information is normally stored in 2 different locations:

- In the list of locks located in InfoPage2 and InfoPage3. Those locks are loaded into the PML registers at boot time, after a POR.
- In dedicated PML registers. PML registers are semi-persistent. Their values are not lost when going to sleep or deep sleep mode.

When one tries to write the InfoPage0, only the PML registers are evaluated. The content of the InfoPage2 and InfoPage3 is **not** loaded.

This function only sets the PML registers.

It results that the locking is valid only until the next reset\POR. If one wants to permanently lock the InfoPage0, he shall configure accordingly the InfoPage2\3.

3.6.14 AES_KeyContainersLockPage

3.6.14.1 Purpose

The function AES_KeyContainersLockPage prevents the InfoPage0 to be written **and** erased.

3.6.14.2 Prototype

AES_Error_t **AES_KeyContainersLockPage**(void)

Lock the key containers page(Info page) 0 against write and erase.

Returns status of the operation

3.6.15 Return values

Type	Description	OK \ NOK
AES_STATUS_SUCCESS	InfoPage0 successfully locked against write and erase	OK

3.6.16 Remark

Warning: The lock information is normally stored in 2 different locations:

- In the list of locks located in InfoPage2 and InfoPage3. Those locks are loaded into the PML registers at boot time, after a POR.
- In dedicated PML registers. PML registers are semi-persistent. Their values are not lost when going to sleep or deep sleep mode.

When one tries to write or erase the InfoPage0, only the PML registers are evaluated. The content of the InfoPage2 and InfoPage3 is **not** loaded.

This function only sets the PML registers.

It results that the locking is valid only until the next reset\POR. If one wants to permanently lock the InfoPage0, he shall configure accordingly the InfoPage2\3.

3.7 Example

Next basic example illustrates:

- how to set keys
- how to lock them
- how to invalidate them
- how to execute AES with those keys
- how to "type" them

```
void Example_KeyContainers(void) {
        uint8_t result[AES_BLOCK_BYTE_SIZE];
        AES_Error_t sw;
        uint8_t keyIndex;
        uint8_t error = 0;
        const uint8_t key1[AES_BLOCK_BYTE_SIZE] = { 0x2b, 0x7e, 0x15, 0x16, 0x28,
                        0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c␣
↪};
        const uint8_t key2[AES_BLOCK_BYTE_SIZE] = { 0x69, 0xc4, 0xe0, 0xd8, 0x6a,
                        0x7b, 0x04, 0x30, 0xd8, 0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a␣
↪};

        uint8_t message1[AES_BLOCK_BYTE_SIZE] = { 0x32, 0x43, 0xf6, 0xa8, 0x88,
```

(continues on next page)

(continued from previous page)

```
                        0x5a, 0x30, 0x8d, 0x31, 0x31, 0x98, 0xa2, 0xe0, 0x37, 0x07, 0x34␣
↪};
        uint8_t expected1[AES_BLOCK_BYTE_SIZE] = { 0x39, 0x25, 0x84, 0x1d, 0x02,
                        0xdc, 0x09, 0xfb, 0xdc, 0x11, 0x85, 0x97, 0x19, 0x6a, 0x0b, 0x32␣
↪};

        uint8_t message2[AES_BLOCK_BYTE_SIZE] = { 0x00, 0x11, 0x22, 0x33, 0x44,
                        0x55, 0x66, 0x77, 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff␣
↪};
        uint8_t expected2[AES_BLOCK_BYTE_SIZE] = { 0x78, 0xcf, 0x9c, 0x98, 0x7f,
                        0x9c, 0x7f, 0xeb, 0x51, 0x4f, 0xe4, 0xa4, 0x19, 0x7b, 0x72, 0x83␣
↪};

        uint8_t i;

//A- Let's set 2 keys. This operation is typically performed once at the configuration of␣
↪the product

//1- Set a key container in range [0..7]
//Set the key container #2 with the key value key1. For this key, we allow encryption␣
↪and decryption.
        keyIndex = (uint8_t) 2U;
        sw = AES_SetKeyContainer((uint32_t*) key1, keyIndex, AES_HW_ENC_DEC);
        if (AES_STATUS_SUCCESS != sw)
                error++;

//2-Lock the key container #2 to avoid any corruption. The key #2 is in the range [0..7],
↪ so it can be locked individually.
        sw = AES_LockKeyContainer(keyIndex);
        if (AES_STATUS_SUCCESS != sw)
                error++;

//3-Set a 2nd key container
//Set the key container #19 with the key value key2. For this key, we allow encryption␣
↪only
        keyIndex = (uint8_t) 19U;
        sw = AES_SetKeyContainer((uint32_t*) key2, keyIndex, AES_HW_ENC_ONLY);
        if (AES_STATUS_SUCCESS != sw)
                error++;
//4-try to lock it but operation should be rejected as the key ID is not in the range [0.
↪.7]
        sw = AES_LockKeyContainer(keyIndex);
        if (AES_STATUS_NOT_LOCKABLE != sw)
                error++;

//B- Let's use the keys

//5-Let's use the first key in encryption
        sw = AES_ProcessBlockKeyContainer((uint8_t) 2U, message1, result,
                        AES_MODE_ENCRYPT);
        if (AES_STATUS_SUCCESS != sw)
                error++;
```

(continues on next page)

(continued from previous page)

```
//check the result
        for (i = 0; i < 16; i++) {
                if (result[i] != expected1[i])
                        error++;
        };

//6-Let's use the first key in decryption
        sw = AES_ProcessBlockKeyContainer((uint16_t) 2U, expected1, result,
                        AES_MODE_DECRYPT);
        if (AES_STATUS_SUCCESS != sw)
                error++;
//check the result
        for (i = 0; i < 16; i++) {
                if (result[i] != message1[i])
                        error++;
        };

//7-Invalidate the first key
        sw = AES_InvalidateKeyContainer((uint16_t) 2U);
        if (AES_STATUS_SUCCESS != sw)
                error++;

//8-Let's try to operate the first key. As the key is invalidated, it should not work␣
↪anymore.
        sw = AES_ProcessBlockKeyContainer((uint8_t) 2U, message1, result,
                        AES_MODE_ENCRYPT);
        if (AES_STATUS_WRONG_PARAM != sw)
                error++;

//9-let's try to use the second key.
        sw = AES_ProcessBlockKeyContainer((uint8_t) 19U, message2, result,
                        AES_MODE_ENCRYPT);
        if (AES_STATUS_SUCCESS != sw)
                error++;
//check the result
        for (i = 0; i < 16; i++) {
                if (result[i] != expected2[i])
                        error++;
        };

//10-Let's use the second key in decryption. As the key was typed as encryption only, it␣
↪should not work
        sw = AES_ProcessBlockKeyContainer((uint8_t) 19U, expected2, result,
                        AES_MODE_DECRYPT);
        if (AES_STATUS_WRONG_PARAM != sw)
                error++;
}
```

CHAPTER

FOUR

AES MODES AND MAC

4.1 AES Electronic Code Book mode (ECB)

4.1.1 Bibliography

[1] NIST SP 800-38A: Recommendation for Block Cipher Modes of Operation

NIST SP800-38A: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

4.1.2 Goal of the document

The goal of this chapter is to describe the functionality of the library **AES ECB**.

This chapter:

- describes the API, including the prototype, the parameters, the error codes etc…
- provides the performances of the function.

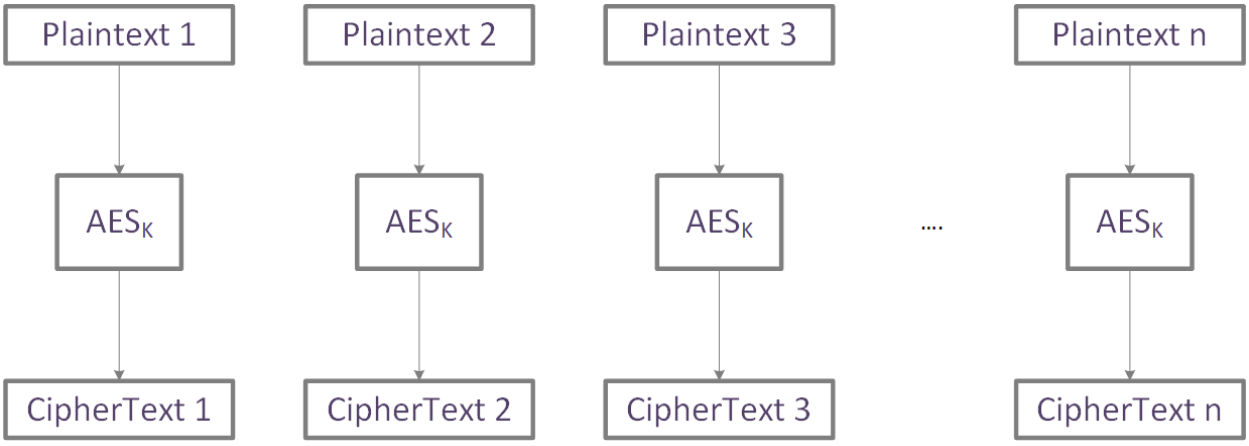
4.1.3 The Electronic Codebook Mode (ECB)

This library implements AES ECB algorithm according to the specification NIST SP 800-38A[1].

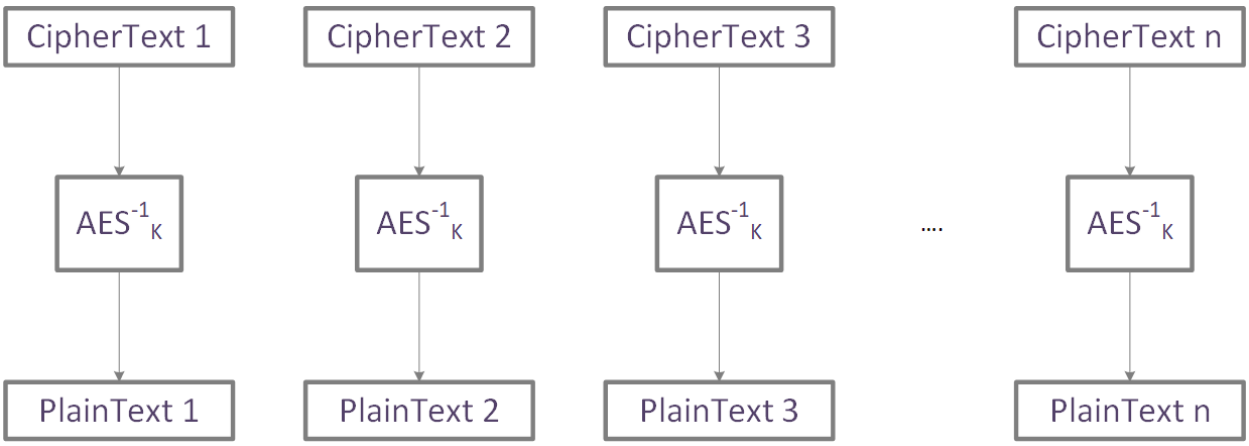
In a nutshell,

- ECB mode is a mode that **encrypts** or **decrypts** messages.
- ECB mode encrypts or decrypts **16-byte blocks** individually. Blocks are not chained.
- ECB mode encrypts or decrypts messages of size a multiple of 16 bytes.

Next figure shows the encryption process. Each 16-byte block of plaintext is encrypted individually using AES in encryption mode.



Next figure shows the decryption process. Each 16-byte block of ciphertext is decrypted individually using AES in decryption mode.



4.1.4 Underlying AES

EM9305 embeds:

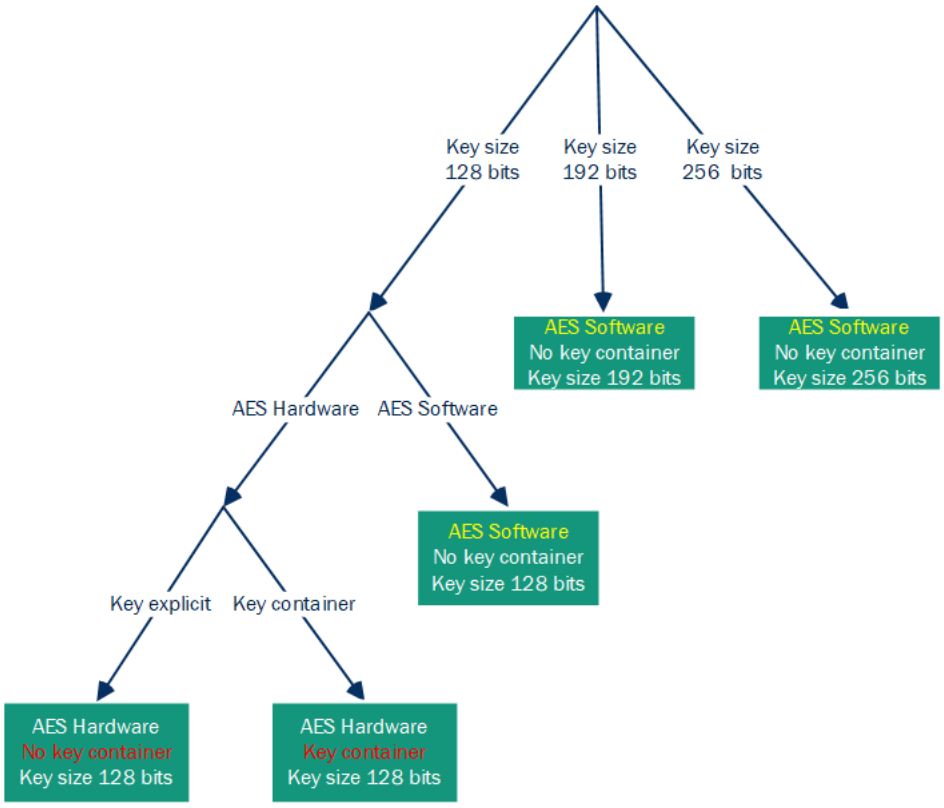
- A software AES that manages the key sizes(128, 192 and 256).
- A hardware AES that only manages 128-bit key size.

The hardware AES can be invoked:

- either with a key given explicitly
- or with a key stored in a key container. In this case, the ID of the key is provided to the AES.

AES ECB APIs interface both the AES Software and the AES hardware. When AES hardware is used, the APIs allows the use of an explicit key or the use of a key container.

Next figure shows which AES is executed, according to the various options.



4.1.5 APIs

4.1.5.1 Enumerations

4.1.5.1.1 AES Type

The underlying algorithm is chosen with the following macro: *AES type*

4.1.5.1.2 AES Key size in bits

The key size is chosen with the macro defined here: *AES Key size in bits*

4.1.5.1.3 Explicit key or key container

The choice between a key provided by value or implicitly with a key container ID is given by the following enumeration:

enum **AES_ECB_KeyType_t**

Select if the key is passed by value(explicitly) or if the key is contained in a key container.

Values:

enumerator **AES_ECB_KEY_VALUE**

Key value is provided explicitly by value.

enumerator **AES_ECB_KEY_ID**

Key value is provided by its ID. The key is in a key container.

4.1.5.1.4 Error status

The API error status are given by:

enum **AES_ECB_Lib_error_t**

Error status words for AES ECB mode.

Values:

enumerator **AES_ECB_SUCCESS**

AES ECB computation successful.

enumerator **AES_ECB_INCOMPATIBLE_PARAMETER**

Incompatible key size with key type and AES type.

enumerator **AES_ECB_INCORRECT_LENGTH**

Incorrect length- Data must be 16 bytes long.

enumerator **AES_ECB_INCORRECT_KEY_LENGTH**

Key length is incorrect.It shall be 128,192 or 256.

enumerator **AES_ECB_INCORRECT_RESULT_POINTER**

Result pointer is null.

4.1.5.2 Context

A context that contains the key size, the underlying AES , the key type and the key value \ key ID is defined as follows:

struct **AES_ECB_CTX**

AES ECB context

Public Members

AES_key_size_bit_t **keySize**

Key size.

AES_Select_t **AESType**

AES Type: Either AES_HARDWARE or AES_SOFTWARE.

AES_ECB_KeyType_t **keyType**

Key type. Either KEY_ID or KEY_VALUE

uint8_t **key**[32]

The key value or the key index

4.1.5.3 AES_ECB_InitCtx

4.1.5.3.1 Goal of the function

This function initializes an AES context with the different options: key size, underlying AES and key type. It also sets the key.

4.1.5.3.2 Prototype

AES_ECB_Lib_error_t **AES_ECB_InitCtx**(*AES_ECB_CTX* *ctx, *AES_Select_t* AESSelect, *AES_ECB_KeyType_t* keyType, *AES_key_size_bit_t* keySize, uint8_t *key)

Initialize an AES ECB context. Choose the underlying AES, the key type and key size. Set the key value.

Parameters

- **ctx** – **[out]** AES ECB context to set
- **AESSelect** – **[in]** The underlying AES: Either AES_HARDWARE or AES_SOFTWARE
- **keyType** – **[in]** The type of the key: Either KEY_VALUE for explicitly key value or KEY_ID for a key in a key container
- **keySize** – **[in]** The size of the key in bits : AES_KEY_128,AES_KEY_192 or AES_KEY_256
- **key** – **[in]** The key value(16 to 32 bytes) or the key ID (first byte)

Return values

- **AES_ECB_SUCCESS** – No error occurred
- **AES_ECB_INCORRECT_KEY_LENGTH** – Incorrect key length
- **AES_ECB_INCOMPATIBLE_PARAMETER** – Hardware/software selection is not compatible with the key length

Returns Error code

4.1.5.3.3 Parameters

- **Ctx** : The AES ECB context to initialize
- **AESSelect**: Select the AES hardware or the AES software
- **keyType** : Select if the key is given explicitly by value or if it is referred by its key container ID.
- **keySize** : Key size in bits
- **key:**
  - When the key is given explicitly: 16, 24 or 32 bytes representing the key value.
  - When the key is given by its ID: 1 byte with the key ID.

4.1.5.3.4 Return values

Table 1: :header: “Type”, “Description”, “OK \\ NOK” :widths: 50,25,15

AES_ECB_SUCCESS	Initialization successful	OK
AES_ECB_INCORRECT_KEY_LENGTH	Incorrect key length	NOK
AES_ECB_INCOMPATIBLE_PARAMETER	Hardware AES is not compatible with this key length	NOK

4.1.5.4 AES_ECB_Encrypt

4.1.5.4.1 Goal of the function

This function encrypts a message in ECB mode.

4.1.5.4.2 Prototype

AES_ECB_Lib_error_t **AES_ECB_Encrypt**(*AES_ECB_CTX* *ctx, uint8_t *plainText, uint8_t *cipherText, uint32_t sizeInBytes)

Encrypt data in ECB mode.

Parameters

- **ctx** – **[inout]** AES ECB context
- **plainText** – **[in]** Pointer on data to encrypt. It should be sizeInBytes long
- **cipherText** – **[out]** Pointer on the result. It should be sizeInBytes long
- **sizeInBytes** – **[in]** Size of the plaintext=Size of the cipher text in bytes. It must be a multiple of 16.

Return values

- **AES_ECB_SUCCESS** – No error occurred
- **AES_ECB_INCORRECT_RESULT_POINTER** – Result pointer not initialized
- **AES_ECB_INCORRECT_LENGTH** – Length is not a multiple of 16

Returns Error code

4.1.5.4.3 Parameters

- **Ctx** : The AES ECB context
- **plainText**: Message to encrypt
- **cipherText** : Encrypted message.
- **sizeInBytes** : Size of the message to encrypt in bytes.

Note:

- Since ECB mode acts on block of 16 bytes, *sizeInBytes* must be a multiple of 16. If the message size is not a multiple of 16, the caller must previously pad the message. The padding scheme is the choice of the user.
- The ciphertext size is the same as the plaintext size.

- Several calls of AES_ECB_Encrypt can be performed consecutively.
- In addition, several AES ECB computations with different contexts can be interlaced.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_ENC_ONLY(Encryption). The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.1.5.4.4 Return values

Type	Description	OK \ NOK
AES_ECB_SUCCESS	Successful encryption	OK
AES_ECB_INCORRECT_RESULT_POINTER	Ciphertext pointer is not initialized	NOK
AES_ECB_INCORRECT_LENGTH	The size in bytes of the plaintext is not a multiple of 16	NOK

4.1.5.5 AES_ECB_Decrypt

4.1.5.5.1 Goal of the function

This function decrypts a message in ECB mode.

4.1.5.5.2 Prototype

AES_ECB_Lib_error_t **AES_ECB_Decrypt**(*AES_ECB_CTX* *ctx, uint8_t *cipherText, uint8_t *plainText, uint32_t sizeInBytes)

Decrypt data in ECB mode.

Parameters

- **ctx** – **[inout]** AES ECB context
- **cipherText** – **[in]** Pointer on data to decrypt. It should be sizeInBytes long
- **plainText** – **[out]** Pointer on the result. It should be sizeInBytes long
- **sizeInBytes** – **[in]** Size of the cipher text=Size of the plaintex in bytes. It must be a multiple of 16.

Return values

- **AES_ECB_SUCCESS** – No error occurred
- **AES_ECB_INCORRECT_RESULT_POINTER** – Result pointer not initialized
- **AES_ECB_INCORRECT_LENGTH** – Length is not a multiple of 16

Returns Error code

4.1.5.5.3 Parameters

- **Ctx** : The AES ECB context
- **cipherText**: Message to decrypt
- **plainText** : Decrypted message.
- **sizeInBytes** : Size of the message to decrypt in bytes.

Note:

- Since ECB mode acts on block of 16 bytes, *sizeInBytes* must be a multiple of 16.
- The last decrypted plaintext may include padding bytes. The user is in charge of removing it when applicable.
- The plaintext size is the same as the ciphertext size.
- Several calls of AES_ECB_Decrypt can be performed consecutively.
- In addition, several AES ECB computations with different contexts can be interlaced.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or AES_DEC_ONLY(Decryption). The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.1.5.5.4 Return values

Type	Description	OK \ NOK
AES_ECB_SUCCESS	Successful decryption	OK
AES_ECB_INCORRECT_RESULT_POINTER	Plaintext pointer is not initialized	NOK
AES_ECB_INCORRECT_LENGTH	The size in bytes of the ciphertext is not a multiple of 16	NOK

4.1.6 Performances

4.1.6.1 Library location

The lib is located in ROM.

4.1.6.2 Code size

Size in bytes
324 bytes

4.1.6.3 RAM

Size in bytes
No usage of global RAM except the AES_ECB_CTX

4.1.6.4 Stack

Size in bytes
128 bytes

4.1.6.5 Execution time

The execution time clearly depends on the size of the message, the key size and the underlying algorithm. In next table, we provide the average time for one block (16 bytes).

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_ECB_InitCtx	Any	Any	5s
AES_ECB_Encrypt	AES Hardware	128	46
AES_ECB_Encrypt	AES Software	128	5
AES_ECB_Encrypt	AES Software	192	136
AES_ECB_Encrypt	AES Software	256	155
AES_ECB_Decrypt	AES Hardware	128	8
AES_ECB_Decrypt	AES Software	128	139
AES_ECB_Decrypt	AES Software	192	177
AES_ECB_Decrypt	AES Software	256	198

4.1.7 Dependencies

AES ECB lib depends on the AES lib.

4.1.8 Example

Next code shows basic examples:

- using the AES hardware with an explicit key
- using the AES hardware with key stored in key containers
- using the AES software

```
//////////////////////////////////////////////////////////////////////////////
///
/// @file       ExampleAES_ECB.c
///
///
/// @brief          Example of use of AES ECB
///
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
///
//////////////////////////////////////////////////////////////////////////////
///
/// @copyright Copyright (C) 2022 EM Microelectronic
/// @cond
///
/// All rights reserved.
///
/// Redistribution and use in source and binary forms, with or without
/// modification, are permitted provided that the following conditions are met:
/// 1. Redistributions of source code must retain the above copyright notice,
/// this list of conditions and the following disclaimer.
/// 2. Redistributions in binary form must reproduce the above copyright notice,
/// this list of conditions and the following disclaimer in the documentation
/// and/or other materials provided with the distribution.
///
//////////////////////////////////////////////////////////////////////////////
///
/// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
/// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
/// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
/// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
/// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
/// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
/// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
/// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
/// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
/// POSSIBILITY OF SUCH DAMAGE.
/// @endcond
//////////////////////////////////////////////////////////////////////////////

#include <stdint.h>
#include "AES.h"
#include "hw_aes.h"
#include "AES_ECB.h"

uint8_t ExampleAES_ECB(void) {

        uint8_t key[AES_BYTE_SIZE_KEY128] = { 0x77, 0x23, 0xd8, 0x7d, 0x77, 0x3a,
                        0x8b, 0xbf, 0xe1, 0xae, 0x5b, 0x08, 0x12, 0x35, 0xb5, 0x66 };

        uint8_t plaintext[AES_BLOCK_SIZE_BYTE * 2] = { 0x1b, 0x0a, 0x69, 0xb7, 0xbc,
                        0x53, 0x4c, 0x16, 0xce, 0xcf, 0xfa, 0xe0, 0x2c, 0xc5, 0x32, 0x31,
```

(continues on next page)

(continued from previous page)

```
                0x90, 0xce, 0xb4, 0x13, 0xf1, 0xdb, 0x3e, 0x9f, 0x0f, 0x79, 0xba,
                0x65, 0x4c, 0x54, 0xb6, 0x0e };

    uint8_t ciphertext[AES_BLOCK_SIZE_BYTE * 2] = { 0xad, 0x5b, 0x08, 0x95,
                0x15, 0xe7, 0x82, 0x10, 0x87, 0xc6, 0x16, 0x52, 0xdc, 0x47, 0x7a,
                0xb1, 0xf2, 0xcc, 0x63, 0x31, 0xa7, 0x0d, 0xfc, 0x59, 0xc9, 0xff,
                0xb0, 0xc7, 0x23, 0xc6, 0x82, 0xf6 };

    uint8_t key256[AES_BYTE_SIZE_KEY256] = { 0xf9, 0x84, 0xb0, 0xf5, 0x34, 0xfc,
                0x0a, 0xe2, 0xc0, 0xa8, 0x59, 0x3e, 0x16, 0xab, 0x83, 0x65, 0xf2,
                0x5f, 0xcc, 0x9c, 0x59, 0x47, 0xf9, 0xa2, 0xdb, 0x45, 0xb5, 0x88,
                0x16, 0x0d, 0x35, 0xc3 };
    uint8_t plaintext256[AES_BLOCK_SIZE_BYTE * 4] = { 0x35, 0x1f, 0xee, 0x09,
                0x91, 0x22, 0xe3, 0x71, 0xc4, 0x83, 0x0f, 0x40, 0x9c, 0x6c, 0x44,
                0x11, 0x18, 0x6d, 0x22, 0x17, 0x6f, 0x71, 0x38, 0xb0, 0x54, 0xf1,
                0x6b, 0x3c, 0x79, 0x67, 0x9c, 0x2f, 0x52, 0x06, 0x85, 0x65, 0x1b,
                0xa8, 0xe4, 0xb6, 0x1c, 0x08, 0xdc, 0xcb, 0x2c, 0x31, 0x98, 0x2f,
                0x74, 0x36, 0x31, 0xa9, 0x75, 0x24, 0xd2, 0xca, 0x4d, 0x35, 0x1a,
                0xc2, 0x35, 0x46, 0xc1, 0x78 };

    uint8_t ciphertext256[AES_BLOCK_SIZE_BYTE * 4] = { 0x8b, 0x9c, 0x9e, 0x69,
                0x2c, 0x16, 0xe7, 0x05, 0x98, 0x18, 0xe2, 0x85, 0xe8, 0x5d, 0x8f,
                0xa5, 0x43, 0x3d, 0xee, 0x2a, 0xff, 0x9f, 0xec, 0x61, 0xd6, 0xa0,
                0xa7, 0x81, 0xe2, 0x4b, 0x24, 0xf6, 0x49, 0x02, 0xfb, 0xd1, 0x8c,
                0xef, 0x74, 0x61, 0xad, 0x77, 0x60, 0xcf, 0xb2, 0x44, 0x2f, 0xb7,
                0x4f, 0xfd, 0x9b, 0xe1, 0x08, 0xa3, 0x86, 0x54, 0x5f, 0x2a, 0x21,
                0x64, 0x30, 0xef, 0x16, 0xfb };

    AES_ECB_CTX Ctx;
    AES_ECB_Lib_error_t sw;
    uint8_t error = 0;
    uint8_t i;

    uint8_t result[AES_BLOCK_SIZE_BYTE * 4];
    uint8_t keyId[1];

    //------------------------------------------------------------------------
    // First example
    //
    // AES ECB with a 128-bit key in encryption mode
    //
    // We use the AES hardware but not the crypto containers.
    //
    // We encrypt all the block in a unique call
    //------------------------------------------------------------------------

    //initialize the context
    sw = AES_ECB_InitCtx(&Ctx, AES_HARDWARE, AES_ECB_KEY_VALUE, AES_KEY_128,
                key);
    if (sw != AES_ECB_SUCCESS)
            error++;
    //encrypt the data
```

(continues on next page)

(continued from previous page)

```
        sw = AES_ECB_Encrypt(&Ctx, plaintext, result, 2 * AES_BLOCK_SIZE_BYTE);
        if (sw != AES_ECB_SUCCESS)
                error++;

        //check the result
        for (i = 0; i < 2 * AES_BLOCK_SIZE_BYTE; i++) {
                if (result[i] != ciphertext[i])
                        error++;
        }

        //------------------------------------------------------------------------------
        // Second example
        //
        // AES ECB with a 128-bit key in decryption mode
        //
        // We use the AES software
        //
        // We decrypt the blocks one by one
        //------------------------------------------------------------------------------

        //initialize the context
        sw = AES_ECB_InitCtx(&Ctx, AES_SOFTWARE, AES_ECB_KEY_VALUE, AES_KEY_128,
                        key);
        if (sw != AES_ECB_SUCCESS)
                error++;
        //decrypt the first block
        sw = AES_ECB_Decrypt(&Ctx, ciphertext, result, AES_BLOCK_SIZE_BYTE);
        if (sw != AES_ECB_SUCCESS)
                error++;
        //decrypt the second block
        sw = AES_ECB_Decrypt(&Ctx, ciphertext + AES_BLOCK_SIZE_BYTE,
                        result + AES_BLOCK_SIZE_BYTE, AES_BLOCK_SIZE_BYTE);
        if (sw != AES_ECB_SUCCESS)
                error++;
        //check the result
        for (i = 0; i < 2 * AES_BLOCK_SIZE_BYTE; i++) {
                if (result[i] != plaintext[i])
                        error++;
        }

        //------------------------------------------------------------------------------
        // Third example

        //
        // AES ECB with a 128-bit key in encryption mode
        //
        // We use the AES hardware and the crypto container
        //
        // We encrypt all the block in a unique call
        //------------------------------------------------------------------------------
        //A priori the key has been written previously in a key container.
        //Here we write the key in container 0x02
```

(continues on next page)

(continued from previous page)

```
        AES_SetKeyContainer((uint32_t*) key, 0x02, AES_HW_ENC_DEC);
        keyId[0] = 0x02;

        //initialize the context
        sw = AES_ECB_InitCtx(&Ctx, AES_HARDWARE, AES_ECB_KEY_ID, AES_KEY_128,
                        keyId);
        //encrypt the data
        sw = AES_ECB_Encrypt(&Ctx, plaintext, result, 2 * AES_BLOCK_SIZE_BYTE);
        if (sw != AES_ECB_SUCCESS)
                error++;

        //check the result
        for (i = 0; i < 2 * AES_BLOCK_SIZE_BYTE; i++) {
                if (result[i] != ciphertext[i])
                        error++;
        }

        //-------------------------------------------------------------------------
        // Fourth example
        //
        // AES ECB with a 256-bit key in decryption mode
        //
        // We necessarily use the AES software
        //
        // We decrypt the 4 blocks with two unbalanced calls.
        // One deals with 3 blocks, the other with one
        //-------------------------------------------------------------------------
        //initialize the context
        sw = AES_ECB_InitCtx(&Ctx, AES_SOFTWARE, AES_ECB_KEY_VALUE, AES_KEY_256,
                        key256);
        if (sw != AES_ECB_SUCCESS)
                error++;
        //decrypt the first block
        sw = AES_ECB_Decrypt(&Ctx, ciphertext256, result, AES_BLOCK_SIZE_BYTE);
        if (sw != AES_ECB_SUCCESS)
                error++;
        //decrypt the 3 next blocks
        sw = AES_ECB_Decrypt(&Ctx, ciphertext256 + AES_BLOCK_SIZE_BYTE,
                        result + AES_BLOCK_SIZE_BYTE, 3 * AES_BLOCK_SIZE_BYTE);
        if (sw != AES_ECB_SUCCESS)
                error++;
        //check the result
        for (i = 0; i < 4 * AES_BLOCK_SIZE_BYTE; i++) {
                if (result[i] != plaintext256[i])
                        error++;
        }

        if (error)
                return (1);
        else
                return (0);
}
```

(continues on next page)

(continued from previous page)

4.2 AES Cipher Block Chaining mode (CBC)

4.2.1 Bibliography

[1] NIST SP 800-38A: Recommendation for Block Cipher Modes of Operation

NIST SP800-38A: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

4.2.2 Goal of the document

The goal of this chapter is to describe the functionality of the library **AES CBC**.

This chapter:

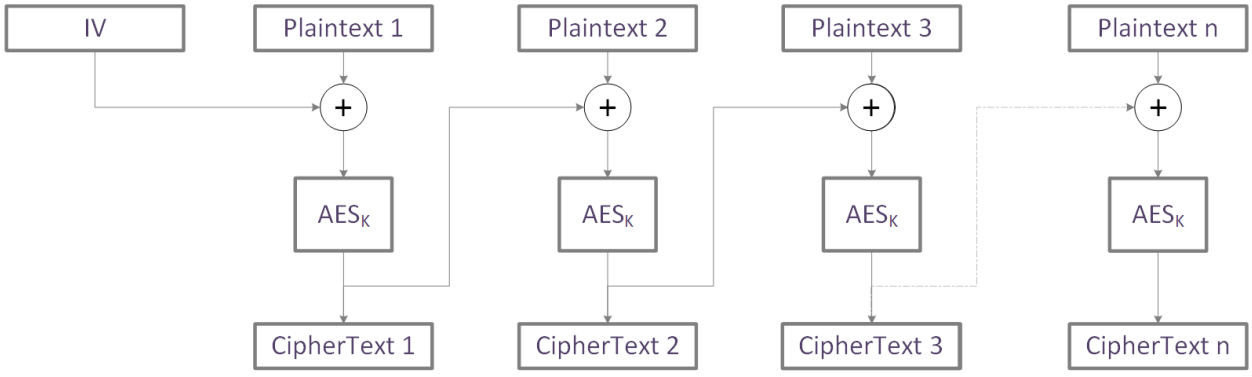
- describes the API, including the prototype, the parameters, the error codes etc…
- provides the performances of the function.

4.2.3 The Cipher Block Chaining mode (CBC)

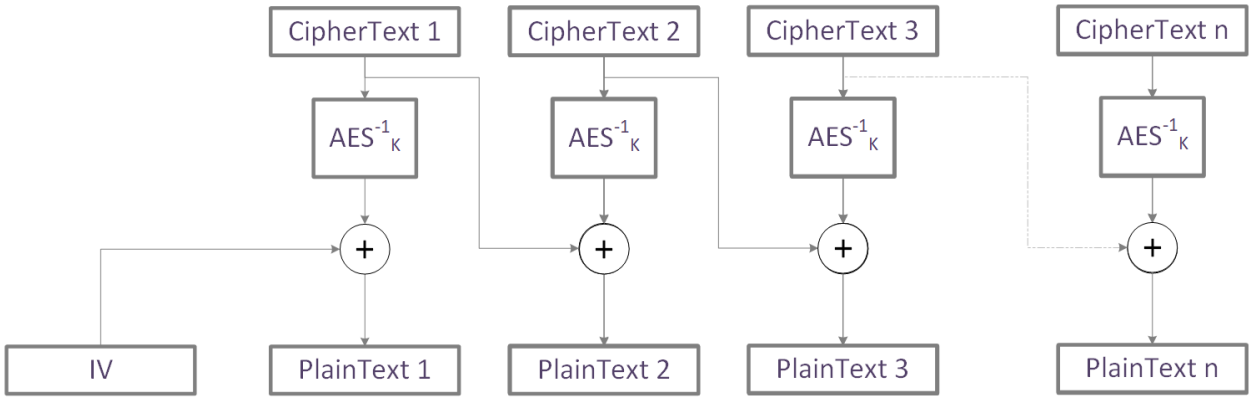
This library implements AES CBC algorithm according to the specification NIST SP 800-38A[1].

- The CBC mode is a mode that encrypts or decrypts messages.
- In encryption in CBC mode combines the plaintext blocks with the previous ciphertext blocks. Combination of blocks is performed with XOR operation.
- The decryption in CBC is the reverse operation of the encryption. It involves combining ciphertext and decrypted blocks with XOR operation.
- The CBC mode requires an initialization vector(IV) of 16 bytes to combine with the first plaintext block.

Next figure shows the encryption process. Each 16-byte block of plaintext is XORed with previous ciphertext and encrypted using AES in encryption mode.



Next figure shows the decryption process. Each 16-byte block of ciphertext is decrypted and then XORed with the previous ciphertext to produce the plaintext block.



4.2.4 Underlying AES

EM9305 embeds:

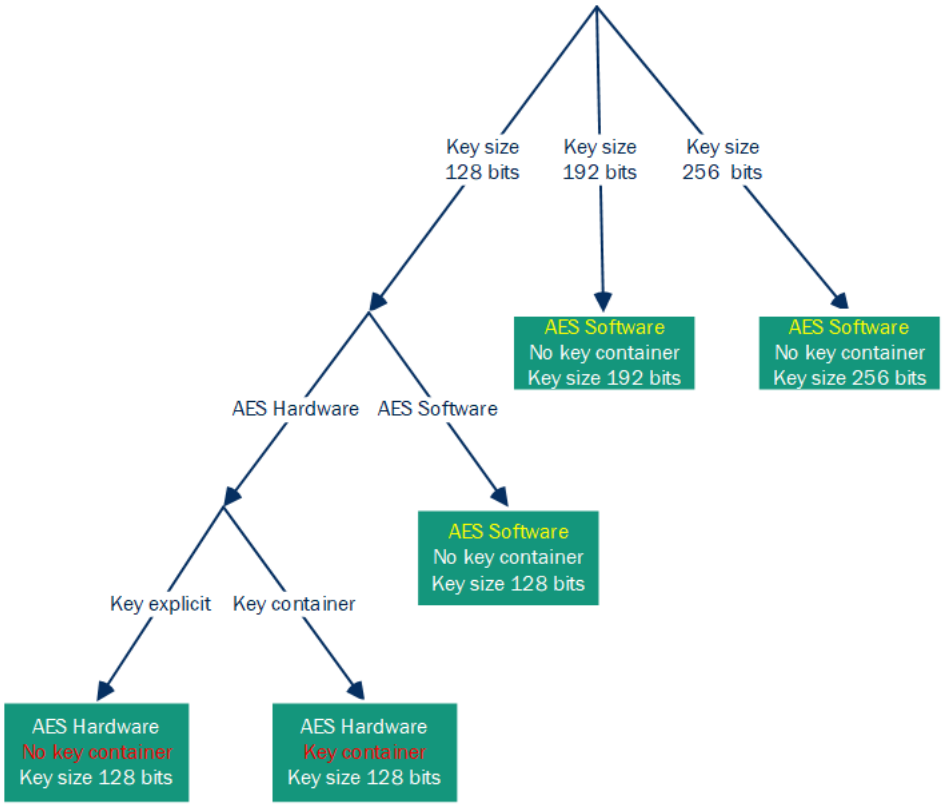
- A software AES that manages the key sizes (128, 192 and 256).
- A hardware AES that only manages 128-bit key size.

The hardware AES can be invoked:

- either with a key given explicitly
- or with a key stored in a key container. In this case, the ID of the key is provided to the AES.

AES CBC APIs interface both the AES Software and the AES hardware. When AES hardware is used, the APIs allow the use of an explicit key or the use of a key container.

Next figure shows which AES is executed, according to the various options.



4.2.5 APIs

4.2.5.1 Enumerations

4.2.5.1.1 AES Type

The underlying algorithm is chosen with the following macro: *AES type*

4.2.5.1.2 AES Key size in bits

The key size is chosen with the macro defined here: *AES Key size in bits*

4.2.5.1.3 Explicit key or key container

The choice between a key provided by value or implicitly with a key container ID is given by the following enumeration:

enum **AES_CBC_KeyType_t**

Select if the key is passed by value(explicitly) or if the key is contained in a key container.

Values:

enumerator **AES_CBC_KEY_VALUE**

Key value is provided explicitly by value.

enumerator **AES_CBC_KEY_ID**

Key value is provided by its ID. The key is in a key container.

4.2.5.1.4 Error status

The API error status is given by:

enum **AES_CBC_Lib_error_t**

Error status words for AES CBC mode.

Values:

enumerator **AES_CBC_SUCCESS**

AES CBC computation successful.

enumerator **AES_CBC_INCOMPATIBLE_PARAMETER**

Incompatible key size with key type and AES type.

enumerator **AES_CBC_INCORRECT_LENGTH**

Incorrect length- Data must be 16 bytes long.

enumerator **AES_CBC_INCORRECT_KEY_LENGTH**

Key length is incorrect.It shall be 128,192 or 256.

enumerator **AES_CBC_INCORRECT_RESULT_POINTER**

Result pointer is null.

4.2.5.2 Context

A context that contains the key size, the underlying AES , the key type and the key value \ key ID is defined as follows. It also defines the initial value.

struct **AES_CBC_CTX**

AES CBC context

Public Members

AES_key_size_bit_t **keySize**

Key size.

AES_Select_t **AESType**

AES Type: Either AES_HARDWARE or AES_SOFTWARE.

AES_CBC_KeyType_t **keyType**

Key type. Either KEY_ID or KEY_VALUE

uint8_t **key**[32]

The key value or the key index

uint8_t **IV**[16]

Initial value

4.2.5.3 AES_CBC_InitCtx

4.2.5.3.1 Goal of the function

This function initializes an AES context with the different options: key size, underlying AES and key type. It also sets the key and the initial value.

4.2.5.3.2 Prototype

AES_CBC_Lib_error_t **AES_CBC_InitCtx**(*AES_CBC_CTX* *ctx, *AES_Select_t* AESSelect, *AES_CBC_KeyType_t* keyType, *AES_key_size_bit_t* keySize, uint8_t *IV, uint8_t *key)

Initialize an AES CBC context. Choose the underlying AES, the key type and key size. Set the key value. Set the initial value.

Parameters

- **ctx** – **[out]** AES CBC context to set
- **AESSelect** – **[in]** The underlying AES: Either AES_HARDWARE or AES_SOFTWARE
- **keyType** – **[in]** The type of the key: Either KEY_VALUE for explicitly key value or KEY_ID for a key in a key container
- **keySize** – **[in]** The size of the key in bits : AES_KEY_128,AES_KEY_192 or AES_KEY_256
- **IV** – **[in]** The 16-byte initial value
- **key** – **[in]** The key value(16 to 32 bytes) or the key ID (first byte)

Return values

- **AES_CBC_SUCCESS** – No error occurred
- **AES_CBC_INCORRECT_KEY_LENGTH** – Incorrect key length
- **AES_CBC_INCOMPATIBLE_PARAMETER** – Hardware/software selection is not compatible with the key length

Returns Error code

4.2.5.3.3 Parameters

- **Ctx** : The AES CBC context to initialize
- **AESSelect**: Select the AES hardware or the AES software
- **keyType**: Select if the key is given explicitly by value or if it refers by its key container ID.
- **keySize**: Key size in bits
- **key:**
  - When the key is given explicitly: 16, 24 or 32 bytes representing the key value.
  - When the key is given by its ID: 1 byte with the key ID.
- **IV**: 16-byte initial value

4.2.5.3.4 Return values

Type	Description	OK \ NOK
AES_CBC_SUCCESS	Initialization successful	OK
AES_CBC_INCORRECT_KEY_LENGTH	Incorrect key length	NOK
AES_CBC_INCOMPATIBLE_PARAMETER	Hardware AES is not compatible with this key length	NOK

4.2.5.4 AES_CBC_Encrypt

4.2.5.4.1 Goal of the function

This function encrypts a message in CBC mode.

4.2.5.4.2 Prototype

AES_CBC_Lib_error_t **AES_CBC_Encrypt**(*AES_CBC_CTX* *ctx, uint8_t *plainText, uint8_t *cipherText, uint32_t sizeInBytes)

Encrypt data in CBC mode.

Parameters

- **ctx** – **[inout]** AES CBC context
- **plainText** – **[in]** Pointer on data to encrypt. It should be sizeInBytes long
- **cipherText** – **[out]** Pointer on the result. It should be sizeInBytes long
- **sizeInBytes** – **[in]** Size of the plaintext=Size of the cipher text in bytes. It must be a multiple of 16.

Return values

- **AES_CBC_SUCCESS** – No error occurred
- **AES_CBC_INCORRECT_RESULT_POINTER** – Result pointer not initialized
- **AES_CBC_INCORRECT_LENGTH** – Length is not a multiple of 16

Returns Error code

4.2.5.4.3 Parameters

- **Ctx** : The AES CBC context
- **plainText**: Message to encrypt
- **cipherText** : Encrypted message.
- **sizeInBytes** : Size of the message to encrypt in bytes.

Note:

- Since CBC mode acts on block of 16 bytes, *sizeInBytes* must be a multiple of 16. If the message size is not a multiple of 16, the caller must previously pad the message. The padding scheme is the choice of the user.
- The ciphertext size is the same as the plaintext size.
- Several calls of AES_CBC_Encrypt can be performed consecutively.
- In addition, several AES CBC computations with different contexts can be interlaced.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_ENC_ONLY(Encryption). The key must not have been invalidated. In case those conditions are not respected, this function returns an error of AES_Error_t.

4.2.5.4.4 Return values

Type	Description	OK \ NOK
AES_CBC_SUCCESS	Successful encryption	OK
AES_CBC_INCORRECT_RESULT_POINTER	Ciphertext pointer is not initialized	NOK
AES_CBC_INCORRECT_LENGTH	The size in bytes of the plaintext is not a multiple of 16	NOK

4.2.5.5 AES_CBC_Decrypt

4.2.5.5.1 Goal of the function

This function decrypts a message in CBC mode.

4.2.5.5.2 Prototype

AES_CBC_Lib_error_t **AES_CBC_Decrypt**(*AES_CBC_CTX* *ctx, uint8_t *cipherText, uint8_t *plainText, uint32_t sizeInBytes)

Decrypt data in CBC mode.

Parameters

- **ctx** – **[inout]** AES CBC context
- **cipherText** – **[in]** Pointer on data to decrypt. It should be sizeInBytes long

- **plainText** – **[out]** Pointer on the result. It should be sizeInBytes long
- **sizeInBytes** – **[in]** Size of the ciphertext=Size of the plaintext in bytes. It must be a multiple of 16.

Return values

- **AES_CBC_SUCCESS** – No error occurred
- **AES_CBC_INCORRECT_RESULT_POINTER** – Result pointer not initialized
- **AES_CBC_INCORRECT_LENGTH** – Length is not a multiple of 16

Returns Error code

4.2.5.5.3 Parameters

- **Ctx** : The AES CBC context
- **cipherText**: Message to decrypt
- **plainText** : Decrypted message.
- **sizeInBytes** : Size of the message to decrypt in bytes.

Note:

- Since CBC mode acts on block of 16 bytes, *sizeInBytes* must be a multiple of 16.
- The last decrypted plaintext may include padding bytes. The user is in charge of removing it when applicable.
- The plaintext size is the same as the ciphertext size.
- Several calls of AES_CBC_Decrypt can be performed consecutively.
- In addition, several AES CBC computations with different contexts can be interlaced.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_DEC_ONLY(Decryption). The key must not have been invalidated. In case those conditions are not respected, this function returns an error of AES_Error_t.

4.2.5.5.4 Return values

Type	Description	OK \ NOK
AES_CBC_SUCCESS	Successful decryption	OK
AES_CBC_INCORRECT_RESULT_POINTER	Plaintext pointer is not initialized	NOK
AES_CBC_INCORRECT_LENGTH	The size in bytes of the ciphertext is not a multiple of 16	NOK

4.2.6 Performances

4.2.6.1 Library location

The lib is located in ROM.

4.2.6.2 Code size

Size in bytes
448 bytes

4.2.6.3 RAM

Size in bytes
No usage of global RAM except the AES_CBC_CTX

4.2.6.4 Stack

Size in bytes
144 bytes

4.2.6.5 Execution time

The execution time clearly depends on the size of the message, the key size and the underlying algorithm. In next table, we provide the figure for one block(16 bytes).

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_CBC_InitCtx	Any	Any	7
AES_CBC_Encrypt	AES Hardware	128	10
AES_CBC_Encrypt	AES Software	128	114
AES_CBC_Encrypt	AES Software	192	141
AES_CBC_Encrypt	AES Software	256	159
AES_CBC_Decrypt	AES Hardware	128	TBD
AES_CBC_Decrypt	AES Software	128	143
AES_CBC_Decrypt	AES Software	192	181
AES_CBC_Decrypt	AES Software	256	11

4.2.7 Dependencies

AES CBC lib depends on the AES lib.

4.2.8 Example

Next code shows basic examples:

- using the AES hardware with an explicit key
- using the AES hardware with key stored in key containers
- using the AES software

```
////////////////////////////////////////////////////////////////////////////////
///
/// @file       ExampleAES_CBC.c
///
///
/// @brief           Example of use of AES CBC
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
///
////////////////////////////////////////////////////////////////////////////////
///
/// @copyright Copyright (C) 2022 EM Microelectronic
/// @cond
///
/// All rights reserved.
///
/// Redistribution and use in source and binary forms, with or without
/// modification, are permitted provided that the following conditions are met:
/// 1. Redistributions of source code must retain the above copyright notice,
/// this list of conditions and the following disclaimer.
/// 2. Redistributions in binary form must reproduce the above copyright notice,
/// this list of conditions and the following disclaimer in the documentation
/// and/or other materials provided with the distribution.
///
////////////////////////////////////////////////////////////////////////////////
///
/// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
/// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
/// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
/// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
/// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
/// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
/// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
/// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
/// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
/// POSSIBILITY OF SUCH DAMAGE.
/// @endcond
////////////////////////////////////////////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
#include <stdint.h>
#include "AES.h"
#include "hw_aes.h"
#include "AES_CBC.h"

uint8_t ExampleAES_CBC(void) {

        uint8_t key[AES_BYTE_SIZE_KEY128] = { 0x33, 0x48, 0xaa, 0x51, 0xe9, 0xa4,
                        0x5c, 0x2d, 0xbe, 0x33, 0xcc, 0xc4, 0x7f, 0x96, 0xe8, 0xde };
        uint8_t iv[AES_BYTE_SIZE_KEY128] = { 0x19, 0x15, 0x3c, 0x67, 0x31, 0x60,
                        0xdf, 0x2b, 0x1d, 0x38, 0xc2, 0x80, 0x60, 0xe5, 0x9b, 0x96 };

        uint8_t plaintext[AES_BLOCK_SIZE_BYTE * 3] = { 0x9b, 0x7c, 0xee, 0x82, 0x7a,
                        0x26, 0x57, 0x5a, 0xfd, 0xbb, 0x7c, 0x7a, 0x32, 0x9f, 0x88, 0x72,
                        0x38, 0x05, 0x2e, 0x36, 0x01, 0xa7, 0x91, 0x74, 0x56, 0xba, 0x61,
                        0x25, 0x1c, 0x21, 0x47, 0x63, 0xd5, 0xe1, 0x84, 0x7a, 0x6a, 0xd5,
                        0xd5, 0x41, 0x27, 0xa3, 0x99, 0xab, 0x07, 0xee, 0x35, 0x99 };

        uint8_t ciphertext[AES_BLOCK_SIZE_BYTE * 3] = { 0xd5, 0xae, 0xd6, 0xc9,
                        0x62, 0x2e, 0xc4, 0x51, 0xa1, 0x5d, 0xb1, 0x28, 0x19, 0x95, 0x2b,
                        0x67, 0x52, 0x50, 0x1c, 0xf0, 0x5c, 0xdb, 0xf8, 0xcd, 0xa3, 0x4a,
                        0x45, 0x77, 0x26, 0xde, 0xd9, 0x78, 0x18, 0xe1, 0xf1, 0x27, 0xa2,
                        0x8d, 0x72, 0xdb, 0x56, 0x52, 0x74, 0x9f, 0x0c, 0x6a, 0xfe, 0xe5␣
↪};

        uint8_t key192[AES_BYTE_SIZE_KEY192] = { 0x16, 0xc9, 0x3b, 0xb3, 0x98, 0xf1,
                        0xfc, 0x0c, 0xf6, 0xd6, 0x8f, 0xc7, 0xa5, 0x67, 0x3c, 0xdf, 0x43,
                        0x1f, 0xa1, 0x47, 0x85, 0x2b, 0x4a, 0x2d };
        uint8_t iv192[AES_BLOCK_SIZE_BYTE] = { 0xea, 0xae, 0xca, 0x2e, 0x07, 0xdd,
                        0xed, 0xf5, 0x62, 0xf9, 0x4d, 0xf6, 0x3f, 0x0a, 0x65, 0x0f };
        uint8_t plaintext192[AES_BLOCK_SIZE_BYTE * 3] = { 0xc5, 0xce, 0x95, 0x86,
                        0x13, 0xbf, 0x74, 0x17, 0x18, 0xc1, 0x74, 0x44, 0x48, 0x4e, 0xba,
                        0xf1, 0x05, 0x0d, 0xdc, 0xac, 0xb5, 0x9b, 0x95, 0x90, 0x17, 0x8c,
                        0xbe, 0x69, 0xd7, 0xad, 0x79, 0x19, 0x60, 0x8c, 0xb0, 0x3a, 0xf1,
                        0x3b, 0xbe, 0x04, 0xf3, 0x50, 0x6b, 0x71, 0x8a, 0x30, 0x1e, 0xa0␣
↪};
        uint8_t ciphertext192[AES_BLOCK_SIZE_BYTE * 3] = { 0xed, 0x6a, 0x50, 0xe0,
                        0xc6, 0x92, 0x1d, 0x52, 0xd6, 0x64, 0x7f, 0x75, 0xd6, 0x7b, 0x4f,
                        0xd5, 0x6a, 0xce, 0x1f, 0xed, 0xb8, 0xb5, 0xa6, 0xa9, 0x97, 0xb4,
                        0xd1, 0x31, 0x64, 0x05, 0x47, 0xd2, 0x2c, 0x5d, 0x88, 0x4a, 0x75,
                        0xe6, 0x75, 0x2b, 0x58, 0x46, 0xb5, 0xb3, 0x3a, 0x51, 0x81, 0xf4␣
↪};

        AES_CBC_CTX Ctx;
        AES_CBC_Lib_error_t sw;
        uint8_t error = 0;
        uint8_t i;

        uint8_t result[AES_BLOCK_SIZE_BYTE * 3];
        uint8_t keyId[1];

        //------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
        // First example
        //
        // AES CBC with a 128-bit key in encryption mode
        //
        // We use the AES hardware but not the crypto containers.
        //
        // We encrypt all the block in a unique call
        //------------------------------------------------------------------------------

        //initialize the context
        sw = AES_CBC_InitCtx(&Ctx, AES_HARDWARE, AES_CBC_KEY_VALUE, AES_KEY_128, iv,
                        key);
        if (sw != AES_CBC_SUCCESS)
                error++;
        //encrypt the data
        sw = AES_CBC_Encrypt(&Ctx, plaintext, result, 3 * AES_BLOCK_SIZE_BYTE);
        if (sw != AES_CBC_SUCCESS)
                error++;

        //check the result
        for (i = 0; i < 3 * AES_BLOCK_SIZE_BYTE; i++) {
                if (result[i] != ciphertext[i])
                        error++;
        }

        //------------------------------------------------------------------------------
        // Second example
        //
        // AES CBC with a 128-bit key in decryption mode
        //
        // We use the AES software
        //
        // We decrypt the blocks one by one
        //------------------------------------------------------------------------------

        //initialize the context
        sw = AES_CBC_InitCtx(&Ctx, AES_SOFTWARE, AES_CBC_KEY_VALUE, AES_KEY_128, iv,
                        key);
        if (sw != AES_CBC_SUCCESS)
                error++;
        //decrypt the first block
        sw = AES_CBC_Decrypt(&Ctx, ciphertext, result, AES_BLOCK_SIZE_BYTE);
        if (sw != AES_CBC_SUCCESS)
                error++;
        //decrypt the second block
        sw = AES_CBC_Decrypt(&Ctx, ciphertext + AES_BLOCK_SIZE_BYTE,
                        result + AES_BLOCK_SIZE_BYTE, AES_BLOCK_SIZE_BYTE);
        if (sw != AES_CBC_SUCCESS)
                error++;
        sw = AES_CBC_Decrypt(&Ctx, ciphertext + 2 * AES_BLOCK_SIZE_BYTE,
                        result + 2 * AES_BLOCK_SIZE_BYTE, AES_BLOCK_SIZE_BYTE);
        if (sw != AES_CBC_SUCCESS)
```

(continues on next page)

(continued from previous page)

```
        error++;
    //check the result
    for (i = 0; i < 3 * AES_BLOCK_SIZE_BYTE; i++) {
        if (result[i] != plaintext[i])
            error++;
    }

    //-------------------------------------------------------------------------
    // Third example
    //
    // AES CBC with a 128-bit key in encryption mode
    //
    // We use the AES hardware and the crypto container
    //
    // We encrypt all the block in a unique call
    //-------------------------------------------------------------------------
    //A priori the key has been written previously in a key container.
    //Here we write the key in container 0x07
    AES_SetKeyContainer((uint32_t*) key, 0x07, AES_HW_ENC_DEC);
    keyId[0] = 0x07;

    //initialize the context
    sw = AES_CBC_InitCtx(&Ctx, AES_HARDWARE, AES_CBC_KEY_ID, AES_KEY_128, iv,
            keyId);
    //encrypt the data
    sw = AES_CBC_Encrypt(&Ctx, plaintext, result, 3 * AES_BLOCK_SIZE_BYTE);
    if (sw != AES_CBC_SUCCESS)
        error++;

    //check the result
    for (i = 0; i < 3 * AES_BLOCK_SIZE_BYTE; i++) {
        if (result[i] != ciphertext[i])
            error++;
    }

    //-------------------------------------------------------------------------
    // fourth example
    //
    // AES CBC with a 128-bit key in decryption mode
    //
    // We use the AES hardware and the crypto container
    //
    // We decrypt the blocks in two steps
    //-------------------------------------------------------------------------

    //reinitialize the context
    sw = AES_CBC_InitCtx(&Ctx, AES_HARDWARE, AES_CBC_KEY_ID, AES_KEY_128, iv,
            keyId);

    //decrypt 2 blocks
    sw = AES_CBC_Decrypt(&Ctx, ciphertext, result, 2 * AES_BLOCK_SIZE_BYTE);
    if (sw != AES_CBC_SUCCESS)
```

(continues on next page)

(continued from previous page)

```
                error++;
        //decrypt 1 block
        sw = AES_CBC_Decrypt(&Ctx, ciphertext + 2 * AES_BLOCK_SIZE_BYTE,
                        result + 2 * AES_BLOCK_SIZE_BYTE, AES_BLOCK_SIZE_BYTE);
        if (sw != AES_CBC_SUCCESS)
                error++;

        //check the result
        for (i = 0; i < 3 * AES_BLOCK_SIZE_BYTE; i++) {
                if (result[i] != plaintext[i])
                        error++;
        }

        //------------------------------------------------------------------------------
        // Fifth example
        //
        // AES CBC with a 192-bit key in encryption mode
        //
        // We necessarily use the AES software
        //
        // We encrypt in one call
        //------------------------------------------------------------------------------
        //initialize the context
        sw = AES_CBC_InitCtx(&Ctx, AES_SOFTWARE, AES_CBC_KEY_VALUE, AES_KEY_192,
                        iv192, key192);
        if (sw != AES_CBC_SUCCESS)
                error++;
        //encrypt the first block
        sw = AES_CBC_Encrypt(&Ctx, plaintext192, result, 3 * AES_BLOCK_SIZE_BYTE);
        if (sw != AES_CBC_SUCCESS)
                error++;

        //check the result
        for (i = 0; i < 3 * AES_BLOCK_SIZE_BYTE; i++) {
                if (result[i] != ciphertext192[i])
                        error++;
        }

        if (error)
                return (1);
        else
                return (0);
}
```

4.3 AES Counter mode(CTR)

4.3.1 Bibliography

[1] NIST SP 800-38A: Recommendation for Block Cipher Modes of Operation

NIST SP800-38A: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

4.3.2 Goal of the document

The goal of this chapter is to describe the functionality of the library **AES CTR**.

This chapter:

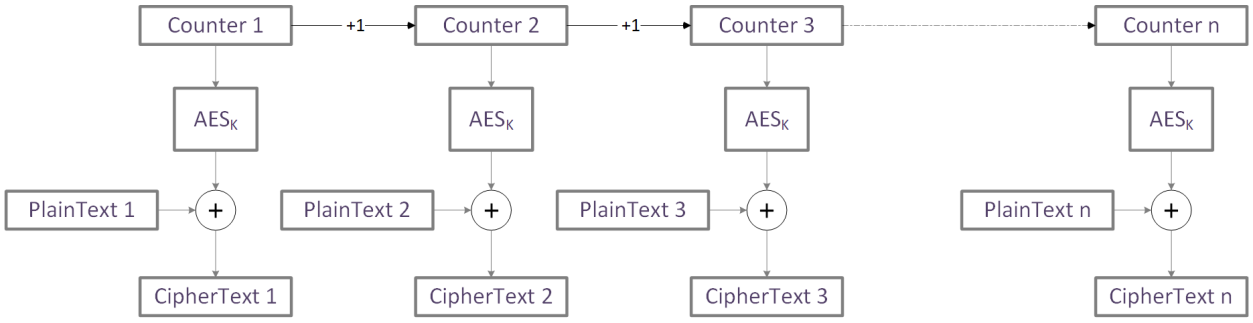
- describes the API, including the prototype, the parameters, the error codes etc…
- provides the performances of the function.

4.3.3 The Counter mode (CTR)

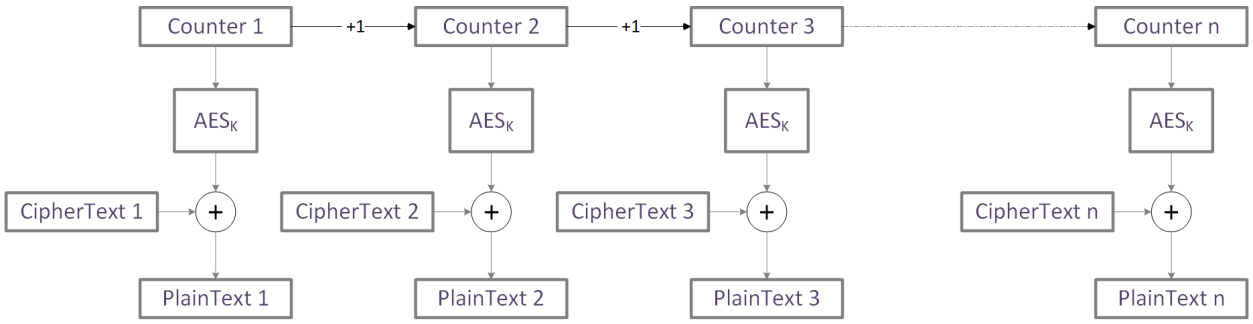
This library implements AES CTR algorithm according to the specification NIST SP 800-38A[1].

- The CTR mode is a mode that encrypts or decrypts messages.
- CTR mode can encrypt and decrypt data of any size.
- In CTR encryption, AES is invoked on each counter block, and the resulting output blocks are XORed with the corresponding plaintext blocks to produce the ciphertext blocks
- In CTR decryption, AES is invoked on each counter block, and the resulting output blocks are XORed with the corresponding ciphertext blocks to produce the plaintext blocks

Next figure shows the encryption process. For each block of 16 bytes, the counter is encrypted with AES in encryption mode. The plaintext block is then XORed with the encryption of the counter to produce the ciphertext. The counter is incremented by one for each block.



Next figure shows the decryption process. For each block of 16 bytes, the counter is encrypted with AES in encryption mode. The ciphertext block is then XORed with the encryption of the counter to retrieve the plaintext block. The counter is incremented by one for each block.



Because the ciphertext is produced XORing byte by byte the plaintext, the encryption granularity is one byte.

4.3.4 Underlying AES

EM9305 embeds:

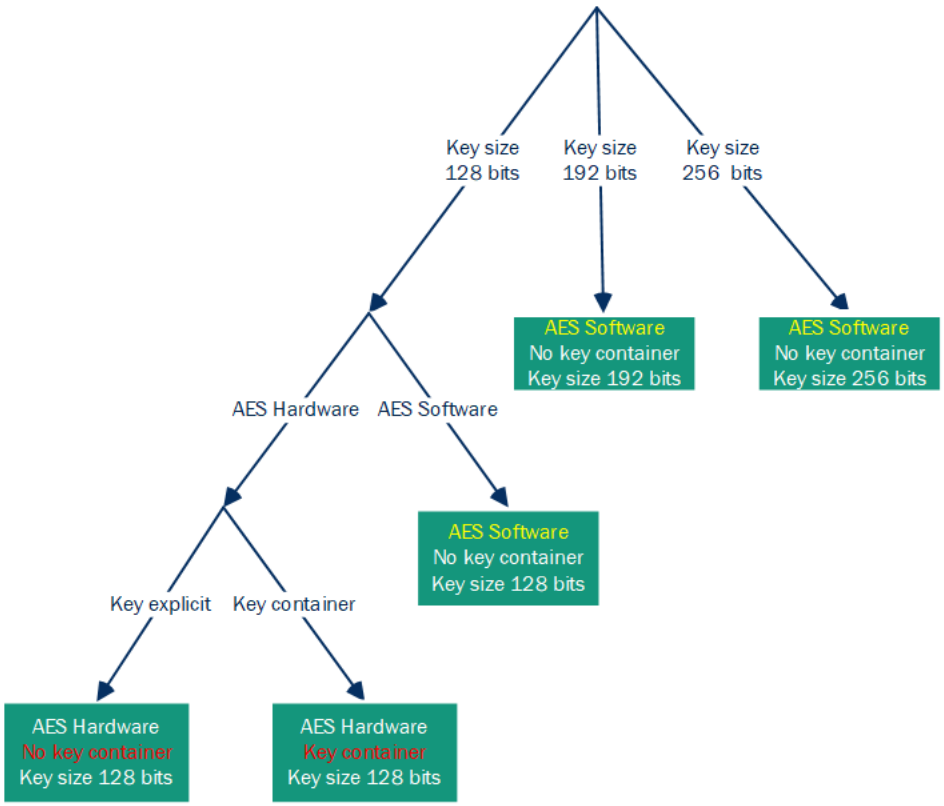
- A software AES that manages the key sizes (128, 192 and 256).
- A hardware AES that only manages 128-bit key size.

The hardware AES can be invoked:

- either with a key given explicitly
- or with a key stored in a key container. In this case, the ID of the key is provided to the AES.

AES CTR APIs interface both the AES Software and the AES hardware. When AES hardware is used, the APIs allow the use of an explicit key or the use of a key container.

Next figure shows which AES is executed, according to the various options.



4.3.5 APIs

4.3.5.1 Enumerations

4.3.5.1.1 AES Type

The underlying algorithm is chosen with the following macro: *AES type*

4.3.5.1.2 AES Key size in bits

The key size is chosen with the macro defined here: *AES Key size in bits*

4.3.5.1.3 Explicit key or key container

The choice between a key provided by value or implicitly with a key container ID is given by the following enumeration:

enum **AES_CTR_KeyType_t**

Select if the key is passed by value(explicitly) or if the key is contained in a key container.

Values:

enumerator **AES_CTR_KEY_VALUE**

Key value is provided explicitly by value.

enumerator **AES_CTR_KEY_ID**

Key value is provied by its ID. The key is in a key container.

4.3.5.1.4 Error status

The API error status is given by:

enum **AES_CTR_Lib_error_t**

Error status words for AES CTR mode.

Values:

enumerator **AES_CTR_SUCCESS**

AES CTR computation successful.

enumerator **AES_CTR_INCOMPATIBLE_PARAMETER**

Incompatible key size with key type and AES type.

enumerator **AES_CTR_INCORRECT_LENGTH**

Incorrect length- Data must be 16 bytes long.

enumerator **AES_CTR_INCORRECT_KEY_LENGTH**

Key length is incorrect.It shall be 128,192 or 256.

enumerator **AES_CTR_INCORRECT_RESULT_POINTER**

Result pointer is null.

4.3.5.2 Context

A context that contains the key size, the underlying AES , the key type , the key value \ key ID and the current counter is defined as follows.

struct **AES_CTR_CTX**

AES CTR context

Public Members

AES_key_size_bit_t **keySize**

Key size.

AES_Select_t **AESType**

AES Type: Either AES_HARDWARE or AES_SOFTWARE.

AES_CTR_KeyType_t **keyType**

Key type. Either KEY_ID or KEY_VALUE

uint8_t **key**[32]

The key value or the key index

uint8_t **counter**[16]

Current Counter Value

uint8_t **buffer**[16]

Current AES result-Accumulation buffer

uint8_t **nbByteReady**

Indicate how many bytes in the buffer are available to encrypt/decrypt the message before executing a new AES

4.3.5.3 AES_CTR_InitCtx

4.3.5.3.1 Goal of the function

This function initializes an AES context with the different options: key size, underlying AES and key type. It also sets the key and the initial counter value.

4.3.5.3.2 Prototype

AES_CTR_Lib_error_t **AES_CTR_InitCtx**(*AES_CTR_CTX* *ctx, *AES_Select_t* AESSelect, *AES_CTR_KeyType_t* keyType, *AES_key_size_bit_t* keySize, uint8_t *initialCounter, uint8_t *key)

Initialize an AES CTR context. Choose the underlying AES, the key type and key size. Set the key value. Set the initial value.

Parameters

- **ctx** – **[out]** AES CTR context to set
- **AESSelect** – **[in]** The underlying AES. Either AES_HARDWARE or AES_SOFTWARE
- **keyType** – **[in]** The type of the key: Either KEY_VALUE for explicitly key value or KEY_ID for a key in a key container
- **keySize** – **[in]** The size of the key in bits : AES_KEY_128,AES_KEY_192 or AES_KEY_256
- **initialCounter** – **[in]** The 16-byte initial counter value
- **key** – **[in]** The key value(16 to 32 bytes) or the key ID (first byte)

Return values

- **AES_CTR_SUCCESS** – No error occurred
- **AES_CTR_INCORRECT_KEY_LENGTH** – Incorrect key length
- **AES_CTR_INCOMPATIBLE_PARAMETER** – Hardware/software selection is not compatible with the key length

Returns Error code

4.3.5.3.3 Parameters

- **Ctx** : The AES CTR context to initialize
- **AESSelect**: Select the AES hardware or the AES software
- **keyType** : Select if the key is given explicitly by value or if it refers by its key container ID.
- **keySize** : Key size in bits
- **initialCounter** : 16-byte initial counter value
- **key:**
  - When the key is given explicitly: 16, 24 or 32 bytes representing the key value.
  - When the key is given by its ID: 1 byte with the key ID.

4.3.5.3.4 Return values

Type	Description	OK \ NOK
AES_CTR_SUCCESS	Initialization successful	OK
AES_CTR_INCORRECT_KEY_LENGTH	Incorrect key length	NOK
AES_CTR_INCOMPATIBLE_PARAMETER	Hardware AES is not compatible with this key length	NOK

4.3.5.4 AES_CTR_Encrypt

4.3.5.4.1 Goal of the function

This function encrypts a message in CTR mode.

4.3.5.4.2 Prototype

AES_CTR_Lib_error_t **AES_CTR_Encrypt**(*AES_CTR_CTX* *ctx, uint8_t *plainText, uint8_t *cipherText, uint32_t sizeInBytes)

Encrypt data in CTR mode.

Parameters

- **ctx** – **[inout]** AES CTR context
- **plainText** – **[in]** Pointer on data to encrypt. It should be sizeInBytes long
- **cipherText** – **[out]** Pointer on the result. It should be sizeInBytes long
- **sizeInBytes** – **[in]** Size of the plaintext=Size of the cipher text in bytes. It can be any size

Return values

- **AES_CTR_SUCCESS** – No error occurred
- **AES_CTR_INCORRECT_RESULT_POINTER** – Result pointer not initialized

Returns Error code

4.3.5.4.3 Parameters

- **Ctx** : The AES CTR context
- **plainText**: Message to encrypt
- **cipherText**: Encrypted message.
- **sizeInBytes**: Size of the message to encrypt in bytes.

Note:

- Granularity of CTR mode is 1 byte. There is no constraint on the size of the message to encrypt.
- The ciphertext size is the same as the plaintext size.
- Several calls of AES_CTR_Encrypt can be performed consecutively.
- In addition, several AES CTR computations with different contexts can be interlaced.

- When encrypting data shorter than 16 bytes, it may be possible that the counter encryption has been performed in a previous call. In that case, the execution time of this function will be faster.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_ENC_ONLY(Encryption). The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.3.5.4.4 Return values

Type	Description	OK \ NOK
AES_CTR_SUCCESS	Successful encryption	OK
AES_CTR_INCORRECT_RESULT_POINTER	Ciphertext pointer is not initialized	NOK
AES_CTR_INCORRECT_LENGTH	The size in bytes of the plaintext is not a multiple of 16	NOK

4.3.5.5 AES_CTR_Decrypt

4.3.5.5.1 Goal of the function

This function decrypts a message in CTR mode.

4.3.5.5.2 Prototype

AES_CTR_Lib_error_t **AES_CTR_Decrypt**(*AES_CTR_CTX* *ctx, uint8_t *cipherText, uint8_t *plainText, uint32_t sizeInBytes)

Decrypt data in CTR mode.

Note: When using key container, the key must be typed either AES_HW_ENC_DEC or AES_HW_ENC_ONLY. This is due to the fact that, in CTR mode, the underlying AES performs an encryption even to decrypt.

Parameters

- **ctx** – **[inout]** AES CTR context
- **cipherText** – **[in]** Pointer on data to decrypt. It should be sizeInBytes long
- **plainText** – **[out]** Pointer on the result. It should be sizeInBytes long
- **sizeInBytes** – **[in]** Size of the cipherText=Size of the cipher text in bytes. It can be any size.

Return values

- **AES_CTR_SUCCESS** – No error occurred
- **AES_CTR_INCORRECT_RESULT_POINTER** – Result pointer not initialized

4.3.5.5.3 Parameters

- **Ctx** : The AES CTR context
- **cipherText**: Message to decrypt
- **plainText** : Decrypted message.
- **sizeInBytes** : Size of the message to decrypt in bytes.

Note:

- Granularity of CTR mode is 1 byte. There is no constraint on the size of the message to decrypt.
- The plaintext size is the same as the cipherText size.
- Several calls of AES_CTR_Encrypt can be performed consecutively.
- In addition, several AES CTR computations with different contexts can be interlaced.
- When decrypting data shorter than 16 bytes, it may be possible that the counter encryption has been performed in a previous call. In that case, the execution time of this function will be faster.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_ENC_ONLY(Encryption). **AES_DEC_ONLY(Decryption)** would **not** work since the AES CTR mode only uses the AES core algorithm in encryption mode. The key must not have been invalidated. In case those conditions are not respected, this function returns an error of AES_Error_t.

4.3.5.5.4 Return values

Type	Description	OK \ NOK
AES_CTR_SUCCESS	Successful decryption	OK
AES_CTR_INCORRECT_RESULT_POINTER	Plaintext pointer is not initialized	NOK
AES_CTR_INCORRECT_LENGTH	The size in bytes of the plaintext is not a multiple of 16	NOK

4.3.6 Performances

4.3.6.1 Library location

The lib is located in ROM.

4.3.6.2 Code size

Size in bytes
328 bytes

4.3.6.3 RAM

Size in bytes
No usage of global RAM except the AES_CTR_CTX

4.3.6.4 Stack

Size in bytes
128 bytes

4.3.6.5 Execution time

The execution time clearly depends on the size of the message, the key size and the underlying algorithm. In next table, we provide the average figure for one block (16 bytes). (in that case, only one AES encryption is performed)

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_CTR_InitCtx	Any	Any	10
AES_CTR_Encrypt	AES Hardware	128	10
AES_CTR_Encrypt	AES Software	128	115
AES_CTR_Encrypt	AES Software	192	141
AES_CTR_Encrypt	AES Software	256	159
AES_CTR_Decrypt	AES Hardware	128	10
AES_CTR_Decrypt	AES Software	128	115
AES_CTR_Decrypt	AES Software	192	141
AES_CTR_Decrypt	AES Software	256	159

Naturally, as AES is always performed in encryption mode, the CTR encryption and decryption times are identical.

4.3.7 Dependencies

AES CTR lib depends on the AES lib.

4.3.8 Example

Next code shows basic examples:

- using the AES hardware with an explicit key
- using the AES hardware with key stored in key containers
- using the AES software

```
////////////////////////////////////////////////////////////////////////////////
///
/// @file       ExampleAES_CTR.c
///
///
/// @brief           Example of use of AES CTR
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
///
////////////////////////////////////////////////////////////////////////////////
///
/// @copyright Copyright (C) 2022 EM Microelectronic
/// @cond
///
/// All rights reserved.
///
/// Redistribution and use in source and binary forms, with or without
/// modification, are permitted provided that the following conditions are met:
/// 1. Redistributions of source code must retain the above copyright notice,
/// this list of conditions and the following disclaimer.
/// 2. Redistributions in binary form must reproduce the above copyright notice,
/// this list of conditions and the following disclaimer in the documentation
/// and/or other materials provided with the distribution.
///
////////////////////////////////////////////////////////////////////////////////
///
/// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
/// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
/// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
/// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
/// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
/// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
/// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
/// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
/// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
/// POSSIBILITY OF SUCH DAMAGE.
/// @endcond
////////////////////////////////////////////////////////////////////////////////

#include <stdint.h>
#include "AES.h"
#include "hw_aes.h"
#include "AES_CTR.h"
```

(continues on next page)

(continued from previous page)

```
uint8_t ExampleAES_CTR(void) {


        uint8_t key[AES_BYTE_SIZE_KEY128]={0x76,0x91,0xBE,0x03,0x5E,0x50,0x20,0xA8,0xAC,
↪0x6E,0x61,0x85,0x29,0xF9,0xA0,0xDC};
        uint8_t counter[AES_BLOCK_SIZE_BYTE]={0x00,0xE0,0x01,0x7B,0x27,0x77,0x7F,0x3F,
↪0x4A,0x17,0x86,0xF0,0x00,0x00,0x00,0x01};
        //messages of any size can be encrypted with CTR mode.The result size is the same
        uint8_t plaintext[36]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,
↪0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,
↪0x1C,0x1D,0x1E,0x1F,0x20,0x21,0x22,0x23};
        uint8_t ciphertext[36]={0xC1,0xCF,0x48,0xA8,0x9F,0x2F,0xFD,0xD9,0xCF,0x46,0x52,
↪0xE9,0xEF,0xDB,0x72,0xD7,0x45,0x40,0xA4,0x2B,0xDE,0x6D,0x78,0x36,0xD5,0x9A,0x5C,0xEA,
↪0xAE,0xF3,0x10,0x53,0x25,0xB2,0x07,0x2F};

        uint8_t key256[AES_BYTE_SIZE_KEY256]={0xF6,0xD6,0x6D,0x6B,0xD5,0x2D,0x59,0xBB,
↪0x07,0x96,0x36,0x58,0x79,0xEF,0xF8,0x86,0xC6,0x6D,0xD5,0x1A,0x5B,0x6A,0x99,0x74,0x4B,
↪0x50,0x59,0x0C,0x87,0xA2,0x38,0x84};
        uint8_t counter256[AES_BLOCK_SIZE_BYTE]={0x00,0xFA,0xAC,0x24,0xC1,0x58,0x5E,0xF1,
↪0x5A,0x43,0xD8,0x75,0x00,0x00,0x00,0x01};
        uint8_t plaintext256[32]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,
↪0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,
↪0x1C,0x1D,0x1E,0x1F};
        uint8_t ciphertext256[32]={0xF0,0x5E,0x23,0x1B,0x38,0x94,0x61,0x2C,0x49,0xEE,
↪0x00,0x0B,0x80,0x4E,0xB2,0xA9,0xB8,0x30,0x6B,0x50,0x8F,0x83,0x9D,0x6A,0x55,0x30,0x83,
↪0x1D,0x93,0x44,0xAF,0x1C};


        AES_CTR_CTX Ctx;
        AES_CTR_Lib_error_t sw;
        uint8_t error = 0;
        uint8_t i;

        uint8_t result[36];
        uint8_t keyId[1];

        //-------------------------------------------------------------------------
        // First example
        //
        // AES CTR with a 128-bit key in encryption mode
        //
        // We use the AES hardware but not the crypto containers.
        //
        // We encrypt all the block in a unique call
        //-------------------------------------------------------------------------

        //initialize the context
        sw = AES_CTR_InitCtx(&Ctx, AES_HARDWARE, AES_CTR_KEY_VALUE, AES_KEY_128, counter,
                        key);
        if (sw != AES_CTR_SUCCESS)
                error++;
        //encrypt the data
```

(continues on next page)

(continued from previous page)

```
        sw = AES_CTR_Encrypt(&Ctx, plaintext, result, 36);
        if (sw != AES_CTR_SUCCESS)
                error++;

        //check the result
        for (i = 0; i < 36; i++) {
                if (result[i] != ciphertext[i])
                        error++;
        }

        //------------------------------------------------------------------------------
        // Second example
        //
        // AES CTR with a 128-bit key in decryption mode
        //
        // We use the AES software
        //
        // We decrypt first 10 bytes then 12 and we complete with 36-12-10=14 bytes
        //------------------------------------------------------------------------------

        //initialize the context
        sw = AES_CTR_InitCtx(&Ctx, AES_SOFTWARE, AES_CTR_KEY_VALUE, AES_KEY_128, counter,
                        key);
        if (sw != AES_CTR_SUCCESS)
                error++;
        //decrypt the first block
        sw = AES_CTR_Decrypt(&Ctx, ciphertext, result, 10);
        if (sw != AES_CTR_SUCCESS)
                error++;
        //decrypt the second block
        sw = AES_CTR_Decrypt(&Ctx, ciphertext + 10,
                        result + 10, 12);
        if (sw != AES_CTR_SUCCESS)
                error++;
        //decrypt the third block
        sw = AES_CTR_Decrypt(&Ctx, ciphertext + 10+12,
                        result + 10+12, 14);
        if (sw != AES_CTR_SUCCESS)
                error++;
        //check the result
        for (i = 0; i < 36; i++) {
                if (result[i] != plaintext[i])
                        error++;
        }

        //------------------------------------------------------------------------------
        // Third example
        //
        // AES CTR with a 128-bit key in encryption mode
        //
        // We use the AES hardware and the crypto container
        //
```

(continues on next page)

(continued from previous page)

```
        // We encrypt all the block in a unique call
        //-------------------------------------------------------------------------
        //A priori the key has been written previously in a key container.
        //Here we write the key in container 0x12
        AES_SetKeyContainer((uint32_t*) key, 0x12, AES_HW_ENC_ONLY);
        keyId[0] = 0x12;

        //initialize the context
        sw = AES_CTR_InitCtx(&Ctx, AES_HARDWARE, AES_CTR_KEY_ID, AES_KEY_128, counter,
                        keyId);
        //encrypt the data
        sw = AES_CTR_Encrypt(&Ctx, plaintext, result, 36);
        if (sw != AES_CTR_SUCCESS)
                error++;

        //check the result
        for (i = 0; i < 36; i++) {
                if (result[i] != ciphertext[i])
                        error++;
        }

        //-------------------------------------------------------------------------
        // fourth example
        //
        // AES CTR with a 128-bit key in decryption mode
        //
        // We use the AES hardware and the crypto container
        //
        // We decrypt the blocks in two steps
        //-------------------------------------------------------------------------

        //reinitialize the context
        sw = AES_CTR_InitCtx(&Ctx, AES_HARDWARE, AES_CTR_KEY_ID, AES_KEY_128, counter,
                        keyId);

        //decrypt 22 bytes
        sw = AES_CTR_Decrypt(&Ctx, ciphertext, result,22);
        if (sw != AES_CTR_SUCCESS)
                error++;
        //decrypt 1 block
        sw = AES_CTR_Decrypt(&Ctx, ciphertext + 22,
                        result + 22, 14);
        if (sw != AES_CTR_SUCCESS)
                error++;

        //check the result
        for (i = 0; i < 36; i++) {
                if (result[i] != plaintext[i])
                        error++;
        }

        //-------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
        // fifth example
        //
        // AES CTR with a 128-bit key in decryption mode
        //
        // We use the AES hardware and the crypto container
        // but we set the key as AES_HW_ENC_ONLY.
        //
        // The decryption function returns an error as the AES hardware needs
        // to be executed in encryption mode but the key is only decryption
        //-------------------------------------------------------------------------
        //A priori the key has been written previously in a key container.
        //Here we write the key in container 0x14.
        //Purposely for this example we set the key as AES_HW_DEC_ONLY
        AES_SetKeyContainer((uint32_t*) key, 0x12, AES_HW_DEC_ONLY);
        keyId[0] = 0x14;

        //reinitialize the context
        sw = AES_CTR_InitCtx(&Ctx, AES_HARDWARE, AES_CTR_KEY_ID, AES_KEY_128, counter,
                        keyId);

        //decrypt 22 bytes
        sw = AES_CTR_Decrypt(&Ctx, ciphertext, result,22);
        if (sw != AES_HW_ERROR)
                error++;


        //-------------------------------------------------------------------------
        // sixth example
        //
        // AES CTR with a 256-bit key in encryption mode
        //
        // We necessarily use the AES software
        //
        // We encrypt in one call
        //-------------------------------------------------------------------------
        //initialize the context
        sw = AES_CTR_InitCtx(&Ctx, AES_SOFTWARE, AES_CTR_KEY_VALUE, AES_KEY_256,
                        counter256, key256);
        if (sw != AES_CTR_SUCCESS)
                error++;
        //encrypt the first block
        sw = AES_CTR_Encrypt(&Ctx, plaintext256, result, 3 * AES_BLOCK_SIZE_BYTE);
        if (sw != AES_CTR_SUCCESS)
                error++;

        //check the result
        for (i = 0; i < 32; i++) {
                if (result[i] != ciphertext256[i])
                        error++;
        }

        if (error)
```

(continues on next page)

(continued from previous page)

```
                return (1);
        else
                return (0);
}
```

4.4 AES CMAC

4.4.1 Bibliography

[1] NIST SP 800-38B: Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication

NIST SP800-38B: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38b.pdf

4.4.2 Goal of the document

The goal of this chapter is to describe the functionality of the library **AES CMAC**.

This chapter:

- describes the API, including the prototype, the parameters, the error codes etc…
- provides the performances of the function.

4.4.3 The CMAC authentication mode

This library implements AES CMAC algorithm according to the specification NIST SP 800-38B[1].

- AES CMAC is a message authentication code (MAC) algorithm that is based on AES.
- As a MAC algorithm , CMAC is used to ensure the data origin and the data integrity.
- AES CMAC does **not** encrypt\decrypt data.

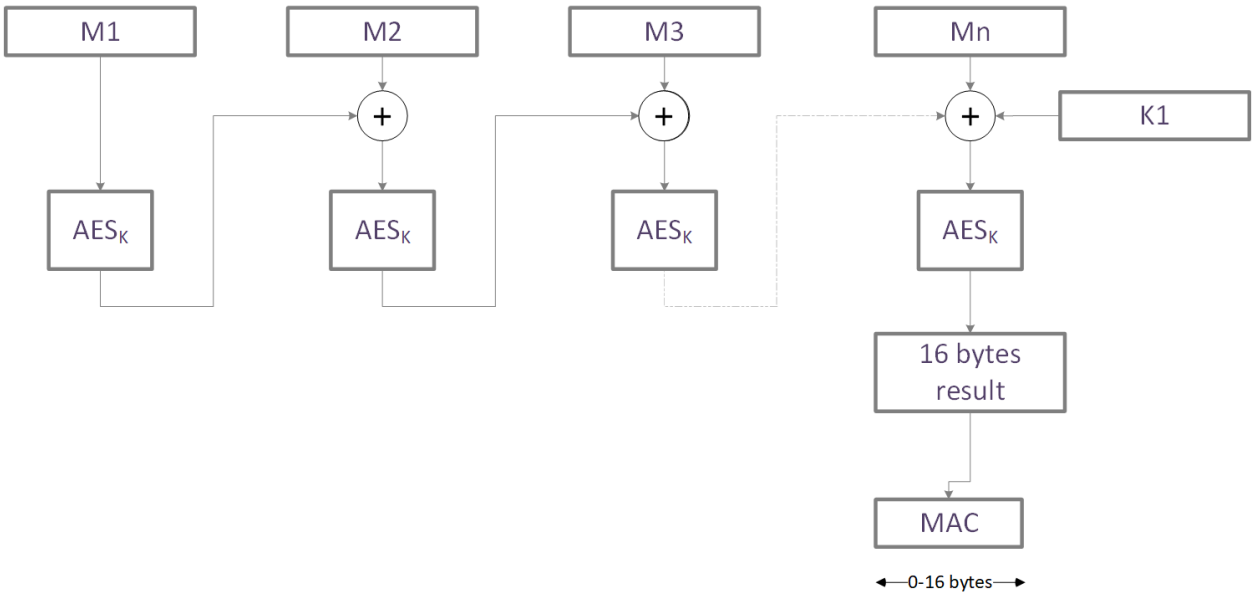
> **Warning:** AES CMAC should not be confused with AES CBC_MAC which uses the last result of AES_CBC to produce a MAC. AES CBC_MAC is insecure to ensure data integrity.

Two flavors of CMAC exit depending if the message size is a multiple of 16 bytes or not.

When the message size is a multiple of 16 bytes:

- The first block of 16 bytes is encrypted.
- Each following block of 16 bytes is XORed with the previous AES result, before being encrypted itself.
- The latest block is XORed with K1, and the previous AES result. It is then encrypted.
- The MAC is then composed of the first *l* bytes of the previous result, where *l* is the desired MAC length.

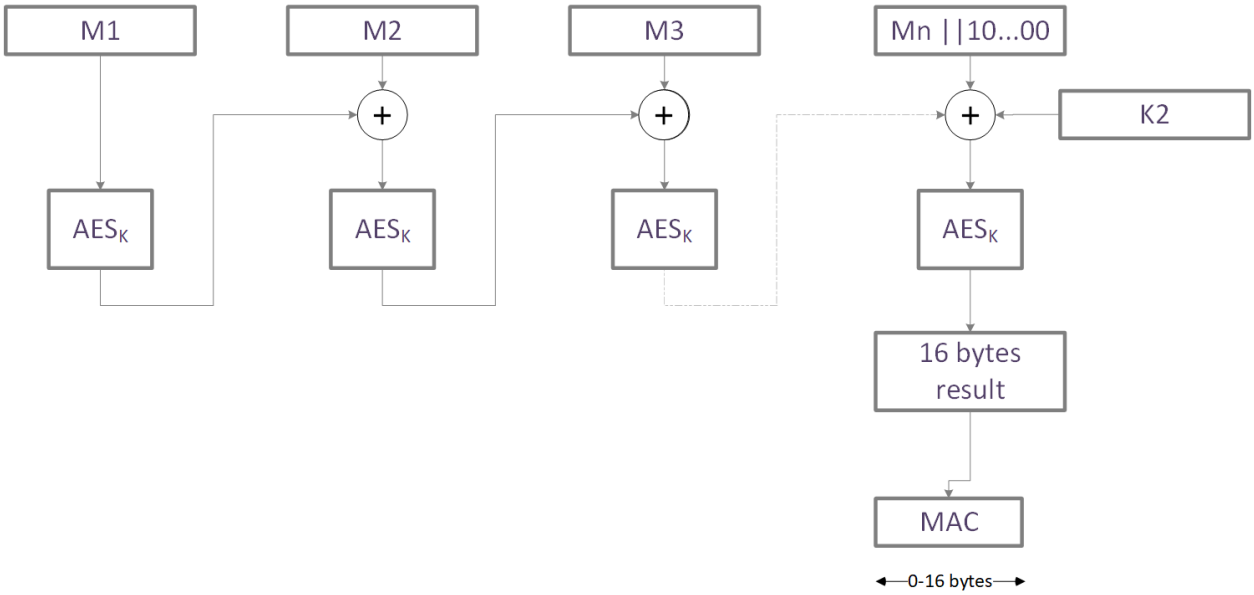
Next figure shows the first flavor.



When the message size is NOT a multiple of 16 bytes:

- The first block of 16 bytes is encrypted.
- Each following block of 16 bytes is XORed with the previous AES result, before being encrypted itself.
- The latest block is padded with 0x10 and as many 0x00 bytes necessary to complete the block,
- The intermediate result is XORed with K2 and previous AES result and then encrypted.
- The MAC is then composed of the first *l* bytes of the previous result, where *l* is the desired MAC length.

Next figure shows the second flavor.



Remarks:

- K1 and K2 are derived from the master key K. The way they are derived is described in [1].
- The message to MAC can be of any size. It is not necessarily a multiple of 16 bytes.

4.4.4 Underlying AES

EM9305 embeds:

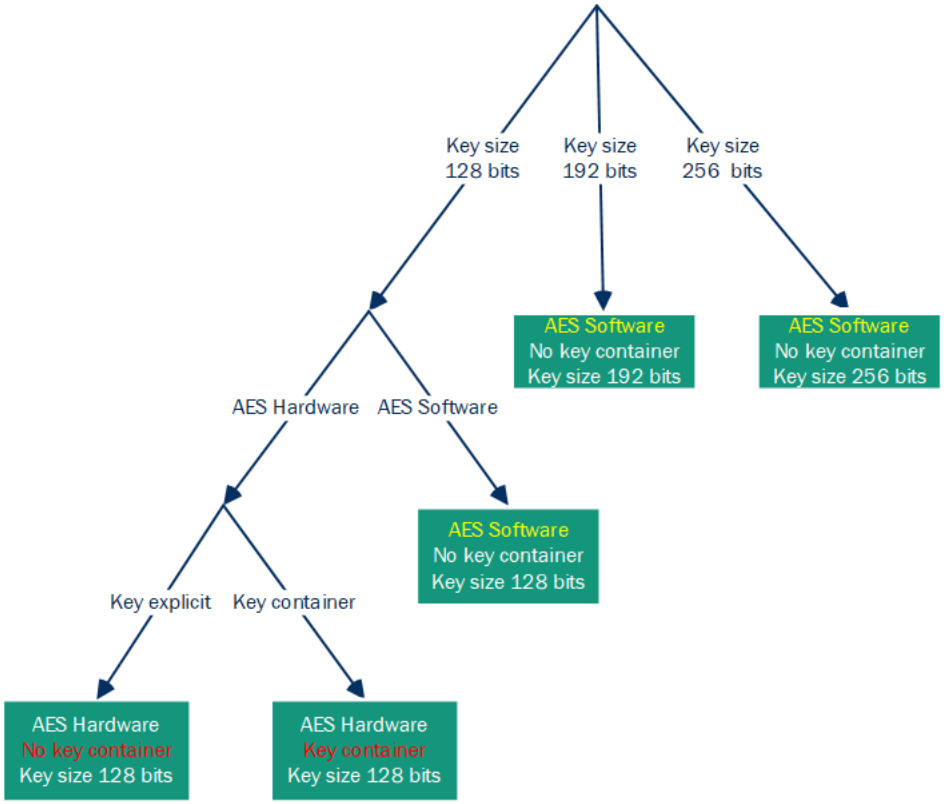
- A software AES that manages the key sizes (128, 192 and 256).
- A hardware AES that only manages 128-bit key size.

The hardware AES can be invoked:

- either with a key given explicitly
- or with a key stored in a key container. In this case, the ID of the key is provided to the AES.

AES CMAC APIs interface both the AES Software and the AES hardware. When AES hardware is used, the APIs allow the use of an explicit key or the use of a key container.

Next figure shows which AES is executed, according to the various options.



4.4.5 APIs

4.4.5.1 Enumerations

4.4.5.1.1 AES Type

The underlying algorithm is chosen with the following macro: *AES type*

4.4.5.1.2 AES Key size in bits

The key size is chosen with the macro defined here: *AES Key size in bits*

4.4.5.1.3 Explicit key or key container

The choice between a key provided by value or implicitly with a key container ID is given by the following enumeration:

enum **AES_CMAC_KeyType_t**

Select if the key is passed by value(explicitly) or if the key is contained in a key container.

Values:

enumerator **AES_CMAC_KEY_VALUE**

Key value is provided explicitly by value.

enumerator **AES_CMAC_KEY_ID**

Key value is provided by its ID. The key is in a key container.

4.4.5.1.4 Error status

The API error status is given by:

enum **AES_CMAC_Lib_error_t**

Error status words for AES CMAC mode.

Values:

enumerator **AES_CMAC_SUCCESS**

AES CMAC computation successful.

enumerator **AES_CMAC_INCOMPATIBLE_PARAMETER**

Incompatible key size with key type and AES type.

enumerator **AES_CMAC_INCORRECT_LENGTH**

Incorrect length- MAC length should be in [0..16].

enumerator **AES_CMAC_INCORRECT_KEY_LENGTH**

Key length is incorrect.It shall be 128,192 or 256.

enumerator **AES_CMAC_INCORRECT_RESULT_POINTER**

Result pointer is null.

4.4.5.2 Context

A context that contains the key size, the underlying AES , the key type , the key value \ key ID and the data accumulated up to now is defined as follows.

struct **AES_CMAC_CTX**

AES CMAC context

Public Members

AES_key_size_bit_t **keySize**

Key size.

AES_Select_t **AESType**

AES Type: Either AES_HARDWARE or AES_SOFTWARE.

AES_CMAC_KeyType_t **keyType**

Key type. Either KEY_ID or KEY_VALUE

uint8_t **key**[32]

The key value or the key index

uint8_t **buffer**[16]

Buffer to accumulate data

uint8_t **nbByteInBuffer**

Number of bytes in the accumulation buffer

uint8_t **isNullMessage**

Special tag to indicate if the message is null

4.4.5.3 AES_CMAC_InitCtx

4.4.5.3.1 Goal of the function

This function initializes an AES context with the different options: key size, underlying AES and key type. It also sets the key and the initial counter value.

4.4.5.3.2 Prototype

AES_CMAC_Lib_error_t **AES_CMAC_InitCtx**(*AES_CMAC_CTX* *ctx, *AES_Select_t* AESSelect, *AES_CMAC_KeyType_t* keyType, *AES_key_size_bit_t* keySize, uint8_t *key)

Initialize an AES CMAC context. Choose the underlying AES, the key type and key size. Set the key value. Set the initial value.

Note: When using key container, the key must be typed either AES_HW_ENC_DEC or AES_HW_ENC_ONLY. This is due to the fact, that AES always encrypts the blocks to compute a CMAC.

Parameters

- **ctx** – **[out]** AES CMAC context to set
- **AESSelect** – **[in]** The underlying AES. Either AES_HARDWARE or AES_SOFTWARE
- **keyType** – **[in]** The type of the key: either KEY_VALUE for explicitly key value or KEY_ID for a key in a key container
- **keySize** – **[in]** The size of the key in bits : AES_KEY_128,AES_KEY_192 or AES_KEY_256
- **key** – **[in]** The key value(16 to 32 bytes) or the key ID (first byte)

Return values

- **AES_CMAC_SUCCESS** – No error occurred
- **AES_CMAC_INCORRECT_KEY_LENGTH** – Incorrect key length
- **AES_CMAC_INCOMPATIBLE_PARAMETER** – Hardware/software selection is not compatible with the key length

Returns Error code

4.4.5.3.3 Parameters

- **Ctx** : The AES CMAC context to initialize
- **AESSelect**: Select the AES hardware or the AES software
- **keyType** : Select if the key is given explicitly by value or if it refers by its key container ID.
- **keySize** : Key size in bits
- **key:**
  - When the key is given explicitly: 16, 24 or 32 bytes representing the key value.
  - When the key is given by its ID: 1 byte with the key ID.

4.4.5.3.4 Return values

Table 2: :header: “Type”, “Description”, “OK \\ NOK” :widths: 50,25,15

AES_CMAC_SUCCESS	Initialization successful	OK
AES_CMAC_INCORRECT_KEY_LENGTH	Incorrect key length	NOK
AES_CMAC_INCOMPATIBLE_PARAMETER	Hardware AES is not compatible with this key length	NOK

4.4.5.4 AES_CMAC_Compute

4.4.5.4.1 Goal of the function

Accumulate data for the MAC computation and compute intermediate MAC.

4.4.5.4.2 Prototype

AES_CMAC_Lib_error_t **AES_CMAC_Compute**(*AES_CMAC_CTX* *ctx, uint8_t *data, uint32_t sizeInBytes)

Accumulate data for the MAC computation and compute intermediate MAC.

Parameters

- **ctx** – **[inout]** AES CMAC context
- **data** – **[in]** Pointer on data to MAC. It should be sizeInBytes long
- **sizeInBytes** – **[in]** Size of the data. It can be any size.

Return values

- **AES_CMAC_SUCCESS** – No error occurred
- **AES_CMAC_INCORRECT_KEY_LENGTH** – Incorrect key length
- **AES_CMAC_INCOMPATIBLE_PARAMETER** – Hardware/software selection is not compatible with the key length

Returns Error code

4.4.5.4.3 Parameters

- **Ctx** : The AES CMAC context
- **data**: Message to MAC
- **sizeInBytes**: Size of the data to deal with during this call.

Note:

- Granularity of CMAC mode is 1 byte. There is no constraint on the size of the message to MAC.
- Several calls of AES_CMAC_Compute can be performed consecutively.
- In addition, several AES CMAC computations with different contexts can be interlaced.
- When accumulating data shorter than 16 bytes, it may be possible that the buffer to encrypt is not yet full. In that case, the execution time of this function will be faster.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_ENC_ONLY(Encryption). The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.4.5.4.4 Return values

Type	Description	OK \ NOK
AES_CMAC_SUCCESS	Successful intermediate computation	OK

4.4.5.5 AES_CMAC_GetMAC

4.4.5.5.1 Goal of the function

This function finalizes the MAC computation.

4.4.5.5.2 Prototype

AES_CMAC_Lib_error_t **AES_CMAC_GetMAC**(*AES_CMAC_CTX* *ctx, uint8_t *MAC, uint8_t sizeMACInByte)

return the sizeMACInByte MAC

Parameters

- **ctx** – **[inout]** AES CMAC context
- **sizeMACInByte** – **[in]** MAC size required . It should be in [0..16]
- **MAC** – **[out]** MAC value(sizeMACInByte byte long)

Return values

- **AES_CMAC_SUCCESS** – No error occurred
- **AES_CMAC_INCORRECT_RESULT_POINTER** – Result pointer not initialized.
- **AES_CMAC_INCORRECT_LENGTH** – MAC length should be lower or equal to 16.

Returns Error code

4.4.5.5.3 Parameters

- **Ctx**: The AES CMAC context.
- **MAC**: Pointer on the MAC buffer.
- **sizeMACInByte**: Size of the desired MAC. It must be in the range[0..16].

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_ENC_ONLY(Encryption). **AES_DEC_ONLY(Decryption)** would **not** work since the AES CMAC mode only uses the AES core algorithm in encryption mode. The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.4.5.5.4 Return values

Type	Description	OK \ NOK
AES_CMAC_SUCCESS	Successful encryption	OK
AES_CMAC_INCORRECT_RESULT_POINTER	MAC pointer is not initialized	NOK
AES_CMAC_INCORRECT_LENGTH	The size of the MAC should be smaller or equal to 16	NOK

4.4.6 Performances

4.4.6.1 Library location

The lib is located in ROM.

4.4.6.2 Code size

Size in bytes
604 bytes

4.4.6.3 RAM

Size in bytes
No usage of global RAM except the AES_CMAC_CTX

4.4.6.4 Stack

Size in bytes
128 bytes

4.4.6.5 Execution time

The AES_CMAC_InitCtx function time is rather constant and almost independent on the key size.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_CMAC_InitCtx	Any	Any	7

The execution time AES_CMAC_Compute clearly depends on the size of the message, the key size and the underlying algorithm. Next table shows the time per complete 16-byte block. Because data are accumulated and processed when the accumulation buffer is full, it may happen that no AES is performed during the call. It may also happen that an AES execution is performed while only 1 byte of data is provided. For a complete CMAC computation, AES_CMAC_Compute will perform (nbByte/16) AES executions or ((nbByte/16)-1) AES executions. The last block is always processed by AES_CMAC_GetMAC.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_CMAC_Compute	AES Hardware	128	12
AES_CMAC_Compute	AES Software	128	117
AES_CMAC_Compute	AES Software	192	143
AES_CMAC_Compute	AES Software	256	161

The AES_CMAC_GetMAC always involves 2 AES computations. One for the computation of K1 or K2 and one to complete the MAC computation.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_CMAC_GetMAC	AES Hardware	128	26
AES_CMAC_GetMAC	AES Software	128	236
AES_CMAC_GetMAC	AES Software	192	289
AES_CMAC_GetMAC	AES Software	256	325

4.4.7 Dependencies

AES CMAC lib depends on the AES lib.

4.4.8 Example

Next code shows basic examples:

- using the AES hardware with an explicit key
- using the AES hardware with key stored in key containers
- using the AES software

```
////////////////////////////////////////////////////////////////////////////////
///
/// @file       ExampleAES_CMAC.c
///
///
/// @brief          Example of use of AES CMAC
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
///
//////////////////////////////////////////////////////////////////////////////
///
/// @copyright Copyright (C) 2022 EM Microelectronic
/// @cond
///
/// All rights reserved.
///
/// Redistribution and use in source and binary forms, with or without
/// modification, are permitted provided that the following conditions are met:
/// 1. Redistributions of source code must retain the above copyright notice,
/// this list of conditions and the following disclaimer.
/// 2. Redistributions in binary form must reproduce the above copyright notice,
/// this list of conditions and the following disclaimer in the documentation
/// and/or other materials provided with the distribution.
///
//////////////////////////////////////////////////////////////////////////////
///
/// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
/// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
/// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
/// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
/// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
/// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
/// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
/// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
/// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
/// POSSIBILITY OF SUCH DAMAGE.
/// @endcond
//////////////////////////////////////////////////////////////////////////////

#include <stdint.h>
#include "AES.h"
#include "hw_aes.h"
#include "AES_CMAC.h"

uint8_t ExampleAES_CMAC(void) {

        uint8_t key[AES_BYTE_SIZE_KEY128] = { 0xc9, 0x40, 0x8a, 0x8b, 0x16, 0x3f,
                        0x1e, 0x60, 0x28, 0x94, 0xb3, 0x23, 0x9c, 0x3f, 0xdb, 0x6d };
        uint8_t message[37] = { 0xae, 0x10, 0x09, 0xf0, 0x36, 0x26, 0xfc, 0xb5,
                        0x4b, 0xf9, 0x8c, 0x32, 0x91, 0x2f, 0x0f, 0x70, 0xbd, 0x39, 0x8c,
                        0x70, 0x9c, 0x3e, 0xd8, 0xbf, 0x57, 0x54, 0xfe, 0x4b, 0xf5, 0xf6,
                        0xe4, 0x75, 0x21, 0xb3, 0x2c, 0x67, 0x2e };
        uint8_t mac[15] = { 0x58, 0xc3, 0xc8, 0x04, 0xc2, 0x98, 0x5d, 0xf4, 0x7c,
                        0x5c, 0x4b, 0xfc, 0xfe, 0x88, 0x37 };

        uint8_t key256[AES_BYTE_SIZE_KEY256] = { 0x6c, 0x0b, 0x2c, 0x3c, 0x5f, 0xec,
                        0x96, 0x1a, 0xb8, 0x4e, 0x68, 0xf5, 0x6c, 0xa1, 0x66, 0x58, 0x6e,
                        0x59, 0x42, 0xfb, 0x25, 0x94, 0xb1, 0x8a, 0x1d, 0xfd, 0xc4, 0xa8,
                        0xfd, 0xf0, 0x76, 0x34 };
```

(continues on next page)

(continued from previous page)

```
        uint8_t message256[10] = { 0xf0, 0x8f, 0x89, 0x08, 0x75, 0xe1, 0x39, 0x48,
                        0x04, 0x89 };
        uint8_t mac256[5] = { 0xb4, 0x9c, 0x22, 0x39, 0xe7 };

        AES_CMAC_CTX Ctx;
        AES_CMAC_Lib_error_t sw;
        uint8_t error = 0;
        uint8_t i;

        uint8_t result[36];
        uint8_t keyId[1];

        //---------------------------------------------------------------------------
        // First example
        //
        // AES CMAC with a 128-bit key
        //
        // We use the AES hardware but not the crypto containers.
        //
        // We compute the MAC in one step. The required MAC size is 15 bytes.
        //---------------------------------------------------------------------------

        //initialize the context
        sw = AES_CMAC_InitCtx(&Ctx, AES_HARDWARE, AES_CMAC_KEY_VALUE, AES_KEY_128,
                        key);
        if (sw != AES_CMAC_SUCCESS)
                error++;
        //compute the MAC
        sw = AES_CMAC_Compute(&Ctx, message, 37);
        if (sw != AES_CMAC_SUCCESS)
                error++;
        //get the 15 bytes MAC
        sw = AES_CMAC_GetMAC(&Ctx, result, 15);
        //check the result
        for (i = 0; i < 15; i++) {
                if (result[i] != mac[i])
                        error++;
        }

        //---------------------------------------------------------------------------
        // Second example
        //
        // AES CMAC with a 128-bit key
        //
        // We use the AES software
        //
        // We compute the MAC in several steps. 15 bytes first, then 17 , then 5 bytes
        //
        // The required MAC size is 4 bytes.
        //---------------------------------------------------------------------------

        //initialize the context
```

(continues on next page)

(continued from previous page)

```
        sw = AES_CMAC_InitCtx(&Ctx, AES_SOFTWARE, AES_CMAC_KEY_VALUE, AES_KEY_128,
                        key);
        if (sw != AES_CMAC_SUCCESS)
                error++;
        //compute the MAC on
        sw = AES_CMAC_Compute(&Ctx, message, 15);
        if (sw != AES_CMAC_SUCCESS)
                error++;
        sw = AES_CMAC_Compute(&Ctx, message + 15, 17);
        if (sw != AES_CMAC_SUCCESS)
                error++;
        sw = AES_CMAC_Compute(&Ctx, message + 32, 5);
        if (sw != AES_CMAC_SUCCESS)
                error++;
        //get the 4 bytes MAC
        sw = AES_CMAC_GetMAC(&Ctx, result, 4);
        //check the result
        for (i = 0; i < 4; i++) {
                if (result[i] != mac[i])
                        error++;
        }

        //-------------------------------------------------------------------------
        // Third example
        //
        // AES CMAC with a 128-bit key
        //
        // We use the AES hardware and the crypto container
        //
        // We compute the MAC in one step. The required MAC size is 6 bytes
        //-------------------------------------------------------------------------
        //A priori the key has been written previously in a key container.
        //Here we write the key in container 0x09
        AES_SetKeyContainer((uint32_t*) key, 0x09, AES_HW_ENC_ONLY);
        keyId[0] = 0x09;

        //initialize the context
        sw = AES_CMAC_InitCtx(&Ctx, AES_HARDWARE, AES_CMAC_KEY_ID, AES_KEY_128,
                        keyId);
        //compute the MAC
        sw = AES_CMAC_Compute(&Ctx, message, 37);
        if (sw != AES_CMAC_SUCCESS)
                error++;
        //get the 6 bytes MAC
        sw = AES_CMAC_GetMAC(&Ctx, result, 6);
        //check the result
        for (i = 0; i < 6; i++) {
                if (result[i] != mac[i])
                        error++;
        }

        //-------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
        // fourth example
        //
        // AES CMAC with a 256-bit key
        //
        // We use the AES software
        //
        // We compute the MAC in one step. The required MAC size is 5 bytes.
        //------------------------------------------------------------------------------

        //initialize the context
        sw = AES_CMAC_InitCtx(&Ctx, AES_SOFTWARE, AES_CMAC_KEY_VALUE, AES_KEY_256,
                        key256);
        if (sw != AES_CMAC_SUCCESS)
                error++;
        //compute the MAC
        sw = AES_CMAC_Compute(&Ctx, message256, 10);
        if (sw != AES_CMAC_SUCCESS)
                error++;
        //get the 5 bytes MAC
        sw = AES_CMAC_GetMAC(&Ctx, result, 5);
        //check the result
        for (i = 0; i < 5; i++) {
                if (result[i] != mac256[i])
                        error++;
        }

        if (error)
                return (1);
        else
                return (0);
}
```

4.5 AES CCM

4.5.1 Bibliography

[1] NIST SP 800-38C: Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality

NIST SP800-38C: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38c.pdf

4.5.2 Goal of the document

The goal of this chapter is to describe the functionality of the library **AES CCM**.

This chapter:

- describes the API, including the prototype, the parameters, the error codes etc…
- provides the performances of the functions.

4.5.3 The Counter with Cipher Block Chaining-Message Authentication Code(CCM)

This library implements AES CCM algorithm according to the specification NIST SP 800-38C[1].

AES CCM is an AEAD algorithm that it is to say an Authenticated Encryption with Additional Data algorithm. As the name indicates it, it allows to encrypt a message and compute a MAC on the message which is optionally appended with additional data before the message.

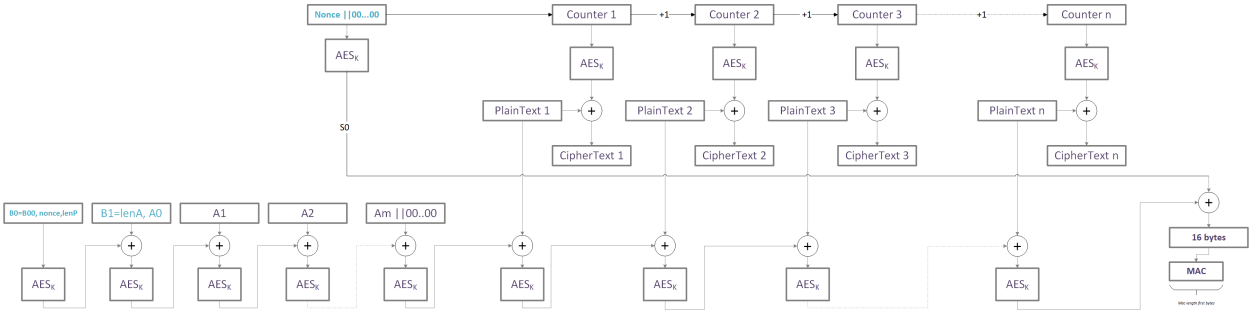
- The MAC is computed on the additional data and the following message.
- Only the message is encrypted.

Fundamentally AES CCM:

- Encrypt\decrypt data with an AES counter mode.
- MAC the data with an AES CBC MAC algorithm.

Because the encryption is performed with a counter mode, the decryption is very similar to encryption. The role of the plaintext and cipher text are just inverted.

Next figure shows the **Encryption and MAC** mechanism.



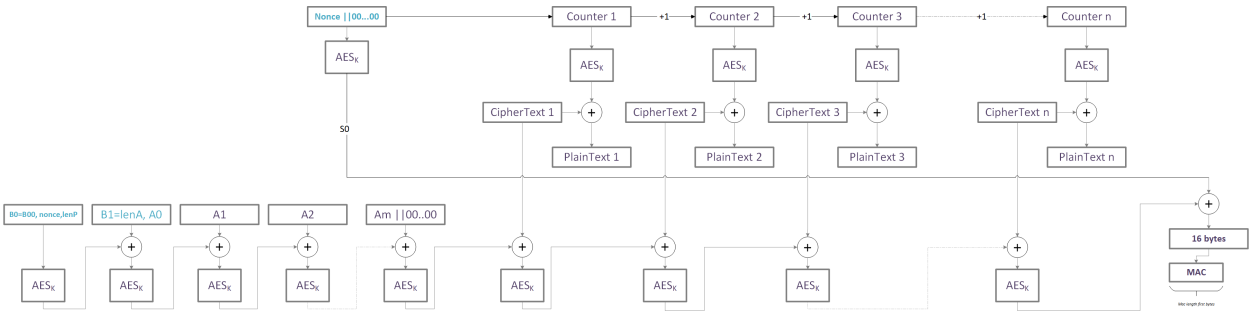
It can be seen that:

- the top of the figure corresponds to the encryption of the plaintext with a AES counter mode.
- the bottom of the figure shows the MAC generation with a AES CBC MAC.

The initialization of the counter is mainly performed with a nonce. The details are given in [1]. The initialization of the MAC involves two blocks. First block is composed of the nonce, the length of the plaintext//ciphertext. The second block contains the length of the additional data. If there is still some room in the second block, it is completed with the beginning of the additional data. Details can again be found in [1].

The MAC process is completed by XORing the encryption of the nonce(S0).

The decryption and MAC process is similar. Ciphertext and plaintext are just inverted. Next figure shows the **Decryption and MAC** mechanism.



4.5.4 Underlying AES

EM9305 embeds:

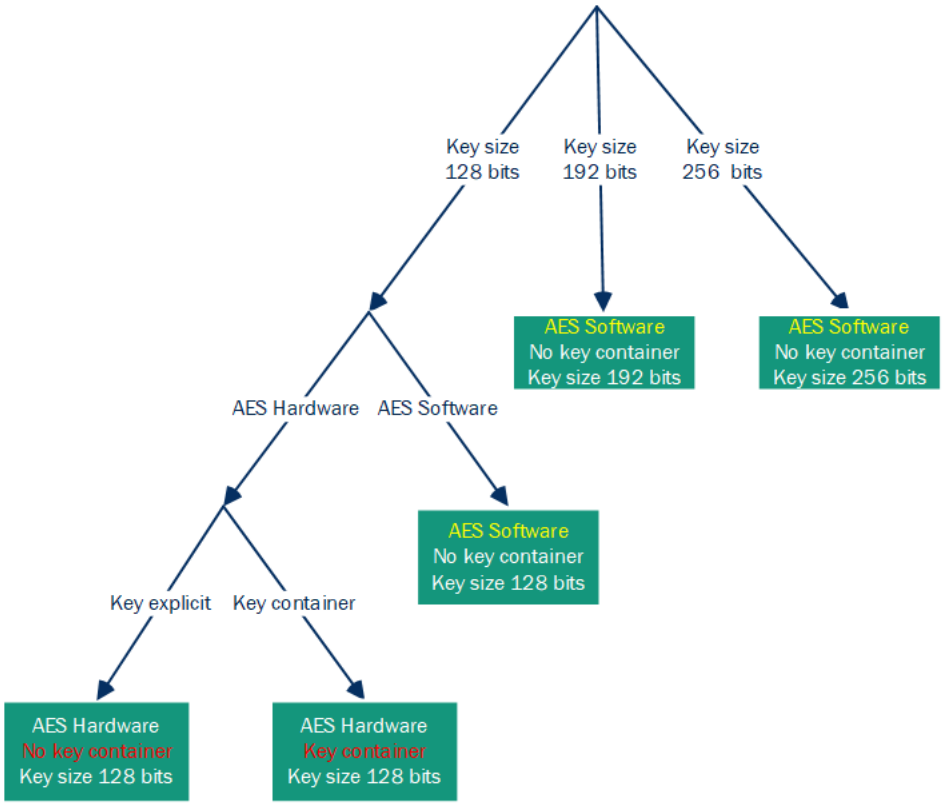
- A software AES that manages the key sizes (128, 192 and 256).
- A hardware AES that only manages 128-bit key size.

The hardware AES can be invoked:

- either with a key given explicitly
- or with a key stored in a key container. In this case, the ID of the key is provided to the AES.

AES CCM APIs interface both the AES Software and the AES hardware. When AES hardware is used, the APIs allows the use of an explicit key or the use of a key container.

Next figure shows which AES is executed, according to the various options.



4.5.5 APIs

4.5.5.1 Enumerations

4.5.5.1.1 AES Type

The underlying algorithm is chosen with the following macro: *AES type*

4.5.5.1.2 AES Key size in bits

The key size is chosen with the macro defined here: *AES Key size in bits*

4.5.5.1.3 Explicit key or key container

The choice between a key provided by value or implicitly with a key container ID is given by the following enumeration:

enum **AES_CCM_KeyType_t**

Select if the key is passed by value(explicitly) or if the key is contained in a key container.

Values:

enumerator **AES_CCM_KEY_VALUE**

Key value is provided explicitly by value.

enumerator **AES_CCM_KEY_ID**

Key value is provided by its ID. The key is in a key container.

4.5.5.1.4 Error status

The API error status are given by:

enum **AES_CCM_Lib_error_t**

Error status words for AES CCM mode.

Values:

enumerator **AES_CCM_SUCCESS**

AES CCM computation successful.

enumerator **AES_CCM_INCOMPATIBLE_PARAMETER**

Incompatible key size with key type and AES type.

enumerator **AES_CCM_INCORRECT_LENGTH**

Incorrect length- Data must be 16 bytes long.

enumerator **AES_CCM_INCORRECT_KEY_LENGTH**

Key length is incorrect.It shall be 128,192 or 256.

enumerator **AES_CCM_INCORRECT_RESULT_POINTER**

Result pointer is null.

enumerator **AES_CCM_INCORRECT_NONCE_LENGTH**

Incorrect Nonce length. It shall be in [7..13] range.

enumerator **AES_CCM_INCORRECT_MAC_LENGTH**

Incorrect MAC length. It shall be in [4..16] range.

enumerator **AES_CCM_INCONSISTENT_NONCE_AND_P_LENGTHS**

LenP is not consistent with the nonce. q=number of significant byte in lenP is such that q+lenNonce=15.

4.5.5.2 Context

A context that contains the key size, the underlying AES , the key type , the key value \ key ID, the length of the MAC, the length of nonce, the current counter, the total length of message, the total length of the additional data and intermediate data is defined as follows.

struct **AES_CCM_CTX**

AES CCM context

Public Members

AES_key_size_bit_t **keySize**

Key size.

AES_Select_t **AESType**

AES Type: Either AES_HARDWARE or AES_SOFTWARE.

AES_CCM_KeyType_t **keyType**

Key type. Either KEY_ID or KEY_VALUE

uint8_t **key**[32]

The key value or the key index

uint8_t **S0**[16]

S0 value to finalize the MAC

uint8_t **counter**[16]

Current counter

uint8_t **buffer**[16]

Intermediate buffer for encryption/decryption

uint8_t **MAC**[16]

MAC value

uint8_t **lenMAC**

Length of the desired MAC

uint8_t **lenNonce**

Length of the Nonce

uint8_t **nbByteReady**

Indicate how many bytes in the buffer are available to encrypt/decrypt the message before executing a new AES

uint8_t **nbMacReady**

Indicate how many bytes in the MAC buffer are accumulated

uint8_t **AdditionalOver**

Indicate if the treatment of additional data is over

4.5.5.3 AES_CCM_InitCtx

4.5.5.3.1 Goal of the function

This function initializes an AES CCM context with the different options: key size, underlying AES and key type. It sets the key and the initial counter value. It also initializes the calculation, computing S0 and the two first blocks of the additional data.

4.5.5.3.2 Prototype

AES_CCM_Lib_error_t **AES_CCM_InitCtx**(*AES_CCM_CTX* *ctx, *AES_Select_t* AESSelect, *AES_CCM_KeyType_t* keyType, *AES_key_size_bit_t* keySize, uint8_t *key, uint8_t lenMAC, uint8_t lenNonce, uint8_t *nonce, uint8_t totalLenP[8], uint8_t totalLenA[8])

Initialize an AES CCM context. Choose the underlying AES, the key type and key size. Set the key value. Set the nonce. Set the MAC length, the total length of the message to encrypt or decrypt. Set the total length of the additional data.

Parameters

- **ctx** – **[out]** AES CCM context to set
- **AESSelect** – **[in]** The underlying AES. Either AES_HARDWARE or AES_SOFTWARE
- **keyType** – **[in]** The type of the key: Either KEY_VALUE for explicitly key value or KEY_ID for a key in a key container

- **keySize** – **[in]** The size of the key in bits : AES_KEY_128,AES_KEY_192 or AES_KEY_256
- **key** – **[in]** The key value(16 to 32 bytes) or the key ID (first byte)
- **lenMAC** – **[in]** Length in bytes of the MAC- It must be even and in range [4..16]
- **lenNonce** – **[in]** Length in bytes of the nonce. It must be in range [7..13]
- **nonce** – **[in]** The nonce value,a buffer of length lenNonce
- **totalLenP** – **[in]** A 8-byte buffer indicating the total length of the plaintext or ciphertext to encrypt or decrypt. If q is the length of the significant size of totalLenP, q+lenNonce must be smaller or equal to 15.
- **totalLenA** – **[in]** A 8-byte buffer indicating the total length of the additional data

Return values

- **AES_CCM_SUCCESS** – Successful initialization
- **AES_CCM_INCORRECT_KEY_LENGTH** – Incorrect key length. It must be 128,192 or 256
- **AES_CCM_INCOMPATIBLE_PARAMETER** – Hardware/software selection is not compatible with the key length. Key containers are only support with 128-bit keys.
- **AES_CCM_INCORRECT_MAC_LENGTH** – Incorrect MAC length. It must be even and in range [4..16]
- **AES_CCM_INCORRECT_NONCE_LENGTH** – Incorrect nonce length. It must be in range [7..13]
- **AES_CCM_INCONSISTENT_NONCE_AND_P_LENGTHS** – Nonce length and totalLenP are not compatible. Their combined length must be smaller or equal to 15.

Returns Error code

4.5.5.3.3 Parameters

- **Ctx** : The AES CCM context to initialize
- **AESSelect**: Select the AES hardware or the AES software
- **keyType**: Select if the key is given explicitly by value or if it refers by its key container ID.
- **keySize**: Key size in bits
- **key:**
  - When the key is given explicitly: 16, 24 or 32 bytes representing the key value.
  - When the key is given by its ID: 1 byte with the key ID.
- **lenMAC** : length of the MAC, an even value between 4 and 16.
- **lenNonce**: an 8-byte array that indicates the length in bytes of the nonce.
- **Nonce**: Nonce of length indicated by lenNonce. It must be a value between 7 and 13.
- **totalLenP** : an 8-byte array that indicates the total length in bytes of the message to encrypt\decrypt. The significant length of totalLenP=q should be so that q+lenNonce=15 at maximum.
- **totalLenA** : an 8-byte array that indicates the total length in bytes of the additional data.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_ENC_ONLY(Encryption). The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.5.5.3.4 Return values

Type	Description	OK \ NOK
AES_CCM_SUCCESS	Initialization successful	OK
AES_CCM_INCORRECT_KEY_LENGTH	Incorrect key length	NOK
AES_CCM_INCOMPATIBLE_PARAMETER	Hardware AES is not compatible with this key length	NOK
AES_CCM_INCORRECT_MAC_LENGTH	Incorrect MAC length	NOK
AES_CCM_INCORRECT_NONCE_LENGTH	Incorrect Nonce length	NOK
AES_CCM_INCONSISTENT_NONCE_AND_P_LENGTHS	length of P and length of the nonce are incompatible	NOK

4.5.5.4 AES_CCM_Hash_Additional_Data

4.5.5.4.1 Goal of the function

This function accumulates data in intermediate buffer and hash it when the buffer is full. It performs the MAC AES computation.

4.5.5.4.2 Prototype

AES_CCM_Lib_error_t **AES_CCM_Hash_Additional_Data**(*AES_CCM_CTX* *ctx, uint8_t *lenA, uint8_t *A)

Deal with the additional data. Additional data are MACed but not encrypted,.

Parameters

- **ctx** – **[inout]** AES CCM context to set
- **lenA** – **[in]** The length in bytes of the additional data dealt during this function call.
- **A** – **[in]** Byte buffer of lenA that contains the additional data

Return values **AES_CCM_SUCCESS** – Additional data successfully dealt

Returns Error code

4.5.5.4.3 Parameters

- **Ctx** : The AES CCM context
- **lenA**: Array representing the length in byte of the data provided during this call. It can be any size from 0 bytes to 0xFFFFFFFFFFFFFFFF bytes.
- **A**: Buffer of lenA bytes with the additional data provided at this call.

Note:

This function can:

- be called once with all the additional data.
- be called several times consecutively, providing parts of the additional data only. In that case lenA is the size of the data provided during this call.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_ENC_ONLY(Encryption). The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.5.5.4.4 Return values

Type	Description	OK \ NOK
AES_CCM_SUCCESS	No error, data accumulated	OK

4.5.5.5 AES_CCM_EncryptAndMAC

4.5.5.5.1 Goal of the function

This function encrypts the data provided and update the MAC accordingly.

4.5.5.5.2 Prototype

AES_CCM_Lib_error_t **AES_CCM_EncryptAndMAC**(*AES_CCM_CTX* *ctx, uint8_t lenP[8], uint8_t *plainText, uint8_t *cipherText)

Encrpyt the plaintext and update the MAC.

Parameters

- **ctx** – **[inout]** AES CCM context to set
- **lenP** – **[in]** The length in bytes of the data to encrypt during this function call.
- **plainText** – **[in]** Byte buffer of lenP that contains the data to encrypt
- **cipherText** – **[out]** Byte buffer of lenP that contains the encrypted data

Return values **AES_CCM_SUCCESS** – Successful encryption

Returns Error code

4.5.5.5.3 Parameters

- **Ctx** : The AES CCM context
- **lenP**: Array representing the length in byte of the plaintext provided during this call. It can be any size from 0 bytes to 0x0000007F FFFFFF00 bytes.
- **plainText**: Buffer of lenP bytes with the plaintext to encrypt this call.
- **cipherText**: Buffer of lenP bytes receiving the encrypted message.

Note:

This function can:

- be called once with the complete plaintext.
- be called several times consecutively, providing parts of the message only. In that case lenP is the size of the data provided during this call.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_ENC_ONLY(Encryption). The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.5.5.5.4 Return values

Type	Description	OK \ NOK
AES_CCM_SUCCESS	Successful encryption	OK
AES_CCM_INCORRECT_RESULT_POINTER	Ciphertext pointer is not initialized	NOK

4.5.5.6 AES_CCM_DecryptAndMAC

4.5.5.6.1 Goal of the function

This function decrypts the data provided and update the MAC accordingly.

4.5.5.6.2 Prototype

AES_CCM_Lib_error_t **AES_CCM_DecryptAndMAC**(*AES_CCM_CTX* *ctx, uint8_t lenC[8], uint8_t *cipherText, uint8_t *plainText)

Decrpyt the cipherText and update the MAC.

Parameters

- **ctx** – **[inout]** AES CCM context to set
- **lenC** – **[in]** The length in bytes of the data to decrypt during this function call.
- **cipherText** – **[in]** Byte buffer of lenP that contains the data to decrypt
- **plainText** – **[out]** Byte buffer of lenP that contains the decrypted data

Return values `AES_CCM_SUCCESS` – Successful encryption

Returns Error code

4.5.5.6.3 Parameters

- **Ctx** : The AES CCM context
- **lenC**: Array representing the length in byte of the ciphertext provided during this call. It can be any size.
- **cipherText**: Buffer of lenC bytes with the ciphertext to decrypt this call.
- **plainText**: Buffer of lenC bytes receiving the decrypted message.

Note:

This function can:

- be called once with the complete ciphertext.
- be called several times consecutively, providing parts of the message only. In that case lenC is the size of the data provided during this call.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type **AES_ENC_ONLY(Encryption)**. As the AES is performed in encryption, a key typed **AES_DEC_ONLY(decryption)** would not work. The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.5.5.6.4 Return values

Table 3: :header: “Type”, “Description”, “OK \\ NOK” :widths: 50,25,15

AES_CCM_SUCCESS	Successful decryption	OK
AES_CCM_INCORRECT_RESULT_POINTER	plainText pointer is not initialized	NOK

4.5.5.7 AES_CCM_GetMAC

4.5.5.7.1 Goal of the function

This function completes the MAC computation and returns the final MAC.

4.5.5.7.2 Prototype

AES_CCM_Lib_error_t **AES_CCM_GetMAC**(*AES_CCM_CTX* *ctx, uint8_t *MAC)

Finalize the MAC computation.

Parameters

- **ctx** – **[inout]** AES CCM context to set
- **MAC** – **[out]** A buffer of the size lenMAC defined previously in the init function.

Return values

- **AES_CCM_SUCCESS** – Successful encryption
- **AES_CCM_INCORRECT_RESULT_POINTER** – MAC pointer not initialized

Returns Error code

4.5.5.7.3 Parameters

- **Ctx** : The AES CCM context
- **MAC**: Buffer to receive the MAC value.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type **AES_ENC_ONLY(Encryption)**. As the AES is performed in encryption, a key typed **AES_DEC_ONLY(decryption)** would not work. The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.5.5.7.4 Return values

Type	Description	OK \ NOK
AES_CCM_SUCCESS	No error, data accumulated	OK
AES_CCM_INCORRECT_RESULT_POINTER	MAC pointer is not initialized	NOK

4.5.6 Performances

4.5.6.1 Library location

The lib is located in ROM.

4.5.6.2 Code size

Size in bytes
1300 bytes

4.5.6.3 RAM

Size in bytes
No usage of global RAM except the AES_CCM_CTX

4.5.6.4 Stack

Size in bytes
Approximately 144 bytes

4.5.6.5 Execution time

The AES_CCM_InitCtx function performs two AESs. Therefore, the time depends on the configuration.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_CCM_InitCtx	AES Hardware	128	36
AES_CCM_InitCtx	AES Software	128	246
AES_CCM_InitCtx	AES Software	192	291
AES_CCM_InitCtx	AES Software	256	330

The execution time AES_CCM_Hash_Additional_Data depends on the size of the additional data that are processed during this call. Because the AES computation is performed on 16 bytes, the operation is performed only every 16 bytes. It induces that the number of AES is roughly the [NbByte16] or[NbByte16]-1. Since the MAC computation involves AES, the performances depend on configuration and key size. Next table shows the time for 16 bytes.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_CCM_Hash_Additional_Data	AES Hardware	128	18
AES_CCM_Hash_Additional_Data	AES Software	128	123
AES_CCM_Hash_Additional_Data	AES Software	192	144
AES_CCM_Hash_Additional_Data	AES Software	256	163

The performance of the function AES_CCM_EncryptAndMAC depends on the size of the plaintext, the underlying AES and also how many bytes were accumulated in the temporary buffer for the MAC computation. It also depends if all the additional data were handled in AES_CCM_Hash_Additional_Data. Globally, two AESs are performed per block of 16 bytes.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_CCM_EncryptAndMAC	AES Hardware	128	31
AES_CCM_EncryptAndMAC	AES Software	128	241
AES_CCM_EncryptAndMAC	AES Software	192	284
AES_CCM_EncryptAndMAC	AES Software	256	321

The performance of the function AES_CCM_DecryptAndMAC are the same for the encryption and mainly depend on the data size. For 16 bytes of data to decrypt we have the following figures.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_CCM_DecryptAndMAC	AES Hardware	128	31
AES_CCM_DecryptAndMAC	AES Software	128	241
AES_CCM_DecryptAndMAC	AES Software	192	284
AES_CCM_DecryptAndMAC	AES Software	256	321

Finally AES_CCM_GetMAC always involves an AES computation. It may involves a second AES if it was not performed previously in the encrypt\decrypt functions. Below are the figures for the case with one AES.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_CCM_GetMAC	AES Hardware	128	14
AES_CCM_GetMAC	AES Software	128	119
AES_CCM_GetMAC	AES Software	192	138
AES_CCM_GetMAC	AES Software	256	156

4.5.7 Dependencies

AES CCM lib depends on the AES lib.

4.5.8 Example

Next code shows basic examples:

- using the AES hardware with an explicit key
- using the AES hardware with key stored in key containers
- using the AES software
- with various size for the additional data
- with various size of plaintext \ ciphertext

```
////////////////////////////////////////////////////////////////////////////////
///
/// @file       ExampleAES_CCM.c
///
/// @project    T9305
///
/// @author     SAS
```

(continues on next page)

(continued from previous page)

```
///
/// @brief          Example of use of AES CCM library
///
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
///
////////////////////////////////////////////////////////////////////////////////
///
/// @copyright Copyright (C) 2022 EM Microelectronic
/// @cond
///
/// All rights reserved.
///
/// Redistribution and use in source and binary forms, with or without
/// modification, are permitted provided that the following conditions are met:
/// 1. Redistributions of source code must retain the above copyright notice,
/// this list of conditions and the following disclaimer.
/// 2. Redistributions in binary form must reproduce the above copyright notice,
/// this list of conditions and the following disclaimer in the documentation
/// and/or other materials provided with the distribution.
///
////////////////////////////////////////////////////////////////////////////////
///
/// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
/// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
/// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
/// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
/// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
/// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
/// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
/// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
/// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
/// POSSIBILITY OF SUCH DAMAGE.
/// @endcond
////////////////////////////////////////////////////////////////////////////////
#include <stdint.h>
#include "AES.h"
#include "AES_CCM.h"
#include "hw_aes.h"

uint8_t ExampleAES_CCM() {

        AES_CCM_CTX ctx;
        AES_CCM_Lib_error_t sw;
        uint16_t error = 0;
        uint8_t i;
        uint8_t Cipher[0x18];
        uint8_t Decipher[0x18];
        uint8_t MAC[0x10];

        //test vector with a 128-bit key
```

(continues on next page)

(continued from previous page)

```
#define TLEN 0x10
#define PLEN 0x18
#define PLEN1 0x07
#define ALEN 0x20
#define ALEN1 0x08
#define NLEN 0x0D
        uint8_t Key[AES_BYTE_SIZE_KEY128] = { 0x43, 0xc1, 0x14, 0x28, 0x77, 0xd9,
                        0xf4, 0x50, 0xe1, 0x2d, 0x7b, 0x6d, 0xb4, 0x7a, 0x85, 0xba };

        uint8_t Additional[0x20] = { 0x6a, 0x59, 0xaa, 0xca, 0xdd, 0x41, 0x6e, 0x46,
                        0x52, 0x64, 0xc1, 0x5e, 0x1a, 0x1e, 0x9b, 0xfa, 0x08, 0x46, 0x87,
                        0x49, 0x27, 0x10, 0xf9, 0xbd, 0xa8, 0x32, 0xe2, 0x57, 0x1e, 0x46,
                        0x82, 0x24 };
        uint8_t LengthA[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ALEN }; //total␣
↪length of A
        uint8_t LengthA1[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ALEN1 }; //
↪Size of the first part of A
        uint8_t LengthA2[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ALEN
                        - ALEN1 }; //Size of the second part of A

        uint8_t Nonce[0x0D] = { 0x76, 0xbe, 0xcd, 0x9d, 0x27, 0xca, 0x8a, 0x02,
                        0x62, 0x15, 0xf3, 0x27, 0x12 };

        uint8_t LengthP[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, PLEN }; //total␣
↪length of P
        uint8_t LengthP1[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, PLEN1 }; //
↪Size of the first part of P
        uint8_t LengthP2[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, PLEN
                        - PLEN1 }; //Size of the second part of P
        uint8_t Plaintext[0x18] = { 0xb5, 0x06, 0xa6, 0xba, 0x90, 0x0c, 0x11, 0x47,
                        0xc8, 0x06, 0x77, 0x53, 0x24, 0xb3, 0x6e, 0xb3, 0x76, 0xaa, 0x01,
                        0xd4, 0xc3, 0xee, 0xf6, 0xf5 };

        uint8_t ExpectedCipher[PLEN] = { 0x14, 0xb1, 0x4f, 0xe5, 0xb3, 0x17, 0x41,
                        0x13, 0x92, 0x86, 0x16, 0x38, 0xec, 0x38, 0x3a, 0xe4, 0x0b, 0xa9,
                        0x5f, 0xef, 0xe3, 0x42, 0x55, 0xdc };
        uint8_t ExpectedMAC[TLEN] = { 0x2e, 0xc0, 0x67, 0x88, 0x71, 0x14, 0xbc,
                        0x37, 0x02, 0x81, 0xde, 0x6f, 0x00, 0x83, 0x6c, 0xe4 };

        uint8_t keyId[1];

        //test vector with a 256-bit key
#define TLEN256 0x04
#define PLEN256 0x18
#define ALEN256 0x20
#define NLEN256 0x0D

        uint8_t Key256[AES_BYTE_SIZE_KEY256] = { 0x90, 0x74, 0xb1, 0xae, 0x4c, 0xa3,
                        0x34, 0x2f, 0xe5, 0xbf, 0x6f, 0x14, 0xbc, 0xf2, 0xf2, 0x79, 0x04,
                        0xf0, 0xb1, 0x51, 0x79, 0xd9, 0x5a, 0x65, 0x4f, 0x61, 0xe6, 0x99,
                        0x69, 0x2e, 0x6f, 0x71 };
        uint8_t Plaintext256[PLEN256] = { 0x23, 0x90, 0x29, 0xf1, 0x50, 0xbc, 0xcb,
```

(continues on next page)

(continued from previous page)

```
                        0xd6, 0x7e, 0xdb, 0xb6, 0x7f, 0x8a, 0xe4, 0x56, 0xb4, 0xea, 0x06,
                        0x6a, 0x4b, 0xee, 0xe0, 0x65, 0xf9 };
        uint8_t LengthA256[8] =
                        { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ALEN256 };
        uint8_t Additional256[ALEN256] = { 0x3c, 0x5f, 0x54, 0x04, 0x37, 0x0a, 0xbd,
                        0xcb, 0x1e, 0xdd, 0xe9, 0x9d, 0xe6, 0x0d, 0x06, 0x82, 0xc6, 0x00,
                        0xb0, 0x34, 0xe0, 0x63, 0xb7, 0xd3, 0x23, 0x77, 0x23, 0xda, 0x70,
                        0xab, 0x75, 0x52 };
        uint8_t Nonce256[NLEN256] = { 0x2e, 0x1e, 0x01, 0x32, 0x46, 0x85, 0x00,
                        0xd4, 0xbd, 0x47, 0x86, 0x25, 0x63 };
        uint8_t LengthP256[8] =
                        { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, PLEN256 };
        uint8_t ExpectedCipher256[PLEN256] = { 0x9c, 0x8d, 0x5d, 0xd2, 0x27, 0xfd,
                        0x9f, 0x81, 0x23, 0x76, 0x01, 0x83, 0x0a, 0xfe, 0xe4, 0xf0, 0x11,
                        0x56, 0x36, 0xc8, 0xe5, 0xd5, 0xfd, 0x74 };
        uint8_t ExpectedMAC256[TLEN256] = { 0x3c, 0xb9, 0xaf, 0xed };

        //-------------------------------------------------------------------------
        // First example
        //
        // Perform AES CCM in encryption + MAC mode
        //
        // The key is 128-bit, we use the AES hardware without key container.
        //
        // MAC length is 16 bytes. Nonce length is 13 bytes.
        //
        // The additional data are 32 bytes. The message to encrypt is 24 bytes.
        //
        // The data are encrypted and MAC in one shot
        //-------------------------------------------------------------------------

        //Initialization:
        //-using a 128 bit key
        //-not using the key container
        //using AES_HARDWARE
        //the length of the message and the length of the additional data are the total␣
↪lengths
        sw = AES_CCM_InitCtx(&ctx, AES_HARDWARE, AES_CCM_KEY_VALUE, AES_KEY_128,
                        Key, TLEN, NLEN, Nonce, LengthP, LengthA);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //Deal with the additional data in one shot
        sw = AES_CCM_Hash_Additional_Data(&ctx, LengthA, Additional);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //Encrpyt the message in one shot
        sw = AES_CCM_EncryptAndMAC(&ctx, LengthP, Plaintext, Cipher);
        if (sw != AES_CCM_SUCCESS)
                error++;
```

(continues on next page)

(continued from previous page)

```
        //get the final MAC
        sw = AES_CCM_GetMAC(&ctx, MAC);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //Check the result is as expected
        for (i = 0; i < PLEN; i++) {
                if (Cipher[i] != ExpectedCipher[i])
                        error++;
        }
        //Check the MAC is as expected
        for (i = 0; i > TLEN; i++) {
                if (MAC[i] != ExpectedMAC[i])
                        error++;
        }

        //-------------------------------------------------------------------------
        // Second example
        //
        // Perform AES CCM in decryption + MAC mode
        //
        // The key is 128-bit, we use the AES software
        //
        // MAC length is 16 bytes. Nonce length is 13 bytes.
        //
        // The additional data are 32 bytes. The message to encrypt is 24 bytes.
        //
        // Both data are encrypted and MAC in several calls.
        //-------------------------------------------------------------------------

        //Initialization:
        //-using a 128 bit key
        //-not using the key container
        //using AES_SOFTWARE
        //the length of the message and the length of the additional data are the total␣
↪lengths
        sw = AES_CCM_InitCtx(&ctx, AES_SOFTWARE, AES_CCM_KEY_VALUE, AES_KEY_128,
                        Key, TLEN, NLEN, Nonce, LengthP, LengthA);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //Introduce the first part of the additional data- LengthA1+LengthA2=LengthA
        sw = AES_CCM_Hash_Additional_Data(&ctx, LengthA1, Additional);
        if (sw != AES_CCM_SUCCESS)
                error++;
        //Introduce the second part of the additional data
        sw = AES_CCM_Hash_Additional_Data(&ctx, LengthA2, Additional + ALEN1);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //decrypt a first part of the cipherLengthP1+LengthP2=LengthP
        sw = AES_CCM_DecryptAndMAC(&ctx, LengthP1, ExpectedCipher, Decipher);
```

(continues on next page)

(continued from previous page)

```
        if (sw != AES_CCM_SUCCESS)
                error++;

        //decrypt the second part of the cipher
        sw = AES_CCM_DecryptAndMAC(&ctx, LengthP2, ExpectedCipher + PLEN1,
                        Decipher + PLEN1);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //get the final MAC
        sw = AES_CCM_GetMAC(&ctx, MAC);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //Check the result is as expected
        for (i = 0; i < PLEN; i++) {
                if (Decipher[i] != Plaintext[i])
                        error++;
        }

        //Check the MAC is as expected
        for (i = 0; i > TLEN; i++) {
                if (MAC[i] != ExpectedMAC[i])
                        error++;
        }

        //------------------------------------------------------------------------
        // Third example
        //
        // Perform AES CCM in encryption + MAC mode
        //
        // The key is 128-bit, we use the AES hardware WITH key container.
        //
        // MAC length is 16 bytes. Nonce length is 13 bytes.
        //
        // The additional data are 32 bytes. The message to encrpyt is 24 bytes.
        //
        // The data are encrypted and MAC in one shot
        //------------------------------------------------------------------------

        //A priori the key has been written previously in a key container.
        //Here we write the key in container 0x04
        AES_SetKeyContainer((uint32_t*) Key, 0x04, AES_HW_ENC_DEC);
        keyId[0] = 0x04;

        //Initialization:
        //-using a 128 bit key
        //-using the key container
        //using AES_HARDWARE
        //the length of the message and the length of the additional data are the total␣
↪lengths
        sw = AES_CCM_InitCtx(&ctx, AES_HARDWARE, AES_CCM_KEY_ID, AES_KEY_128, keyId,
```

(continues on next page)

(continued from previous page)

```
        TLEN, NLEN, Nonce, LengthP, LengthA);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //Deal with the additional data in one shot
        sw = AES_CCM_Hash_Additional_Data(&ctx, LengthA, Additional);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //Encrpyt the message in one shot
        sw = AES_CCM_EncryptAndMAC(&ctx, LengthP, Plaintext, Cipher);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //get the final MAC
        sw = AES_CCM_GetMAC(&ctx, MAC);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //Check the result is as expected
        for (i = 0; i < PLEN; i++) {
                if (Cipher[i] != ExpectedCipher[i])
                        error++;
        }
        //Check the MAC is as expected
        for (i = 0; i > TLEN; i++) {
                if (MAC[i] != ExpectedMAC[i])
                        error++;
        }

        //-------------------------------------------------------------------------
        // Fourth example
        //
        // Perform AES CCM in encryption + MAC mode
        //
        // The key is 256-bit, we use the AES software
        //
        // MAC length is 4 bytes. Nonce length is 13 bytes.
        //
        // The additional data are 32 bytes. The message to encrpyt is 24 bytes.
        //
        // The data are encrypted and MAC in one shot
        //-------------------------------------------------------------------------
↪--------------

        //Initialization:
        //-using a 256 bit key
        //using AES_SOFTWARE
        //the length of the message and the length of the additional data are the total␣
↪lengths
        sw = AES_CCM_InitCtx(&ctx, AES_SOFTWARE, AES_CCM_KEY_VALUE, AES_KEY_256,
                        Key256, TLEN256, NLEN256, Nonce256, LengthP256, LengthA256);
```

(continues on next page)

(continued from previous page)

```
        if (sw != AES_CCM_SUCCESS)
                error++;

        //Deal with the additional data in one shot
        sw = AES_CCM_Hash_Additional_Data(&ctx, LengthA256, Additional256);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //Encrpyt the message in one shot
        sw = AES_CCM_EncryptAndMAC(&ctx, LengthP256, Plaintext256, Cipher);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //get the final MAC
        sw = AES_CCM_GetMAC(&ctx, MAC);
        if (sw != AES_CCM_SUCCESS)
                error++;

        //Check the result is as expected
        for (i = 0; i < PLEN; i++) {
                if (Cipher[i] != ExpectedCipher256[i])
                        error++;
        }
        //Check the MAC is as expected
        for (i = 0; i > TLEN; i++) {
                if (MAC[i] != ExpectedMAC256[i])
                        error++;
        }

        if (error != 0)
                return (1);
        else
                return (0);
}
```

4.6 AES GCM

4.6.1 Bibliography

[1] NIST SP 800-38D: Recommendation for Block Cipher Modes of Operation: the GCM Mode for Authentication and Confidentiality

NIST SP800-38D: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf

4.6.2 Goal of the document

The goal of this chapter is to describe the functionality of the library **AES GCM**.

This chapter:

- describes the API, including the prototype, the parameters, the error codes etc…
- provides the performances of the function.

4.6.3 The Galois Counter Mode (GCM)

This library implements AES GCM algorithm according to the specification NIST SP 800-38D[1].

AES GCM is an AEAD algorithm that it is to say an Authenticated Encryption with Additional Data algorithm. As the name indicates it, it allows to encrypt a message and compute a MAC on the message which is optionally appended with additional data before the message.

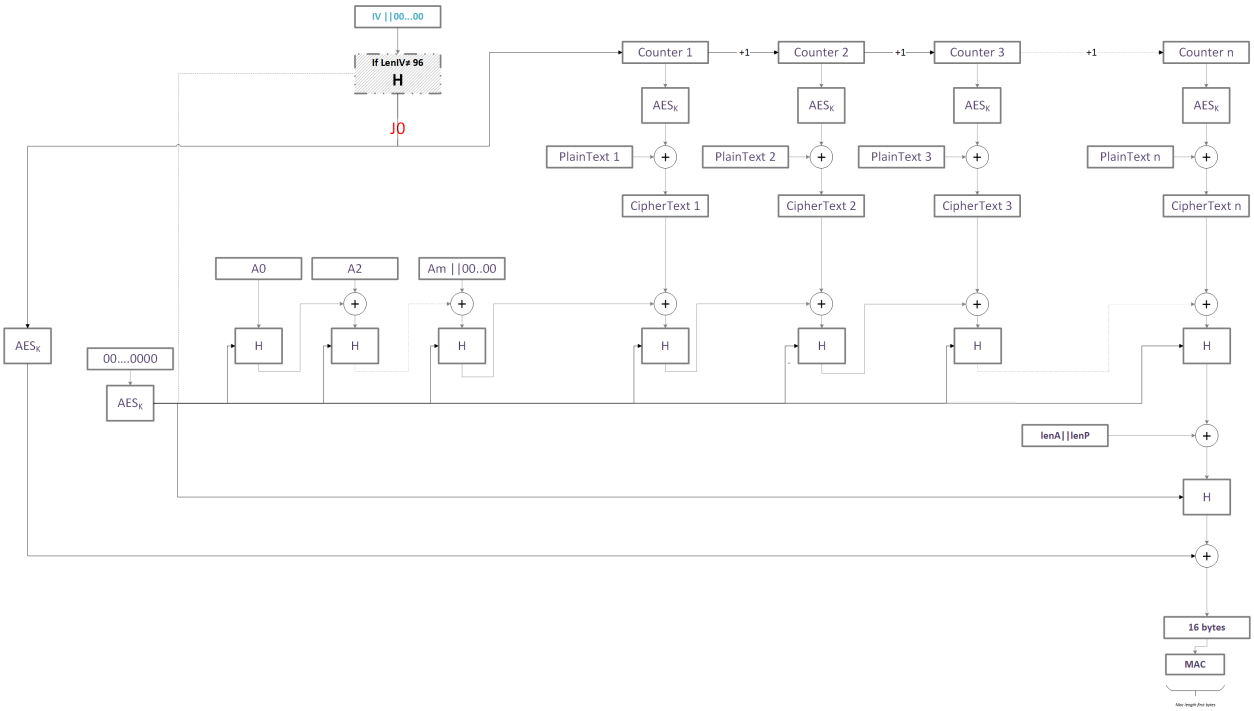
- The MAC is computed on the additional data and the following message.
- Only the message is encrypted.

Fundamentally AES GCM:

- Encrypt\decrypt data with an AES counter mode.
- MAC the data with Galois Field multiplication (H)
- Data of any size can be encrypted\decrypted and MAC.

Because the encryption is performed with a counter mode, the decryption is very similar to encryption. The plaintext and ciphertext are just inverted.

Next figure shows the **Encryption and MAC** mechanism.



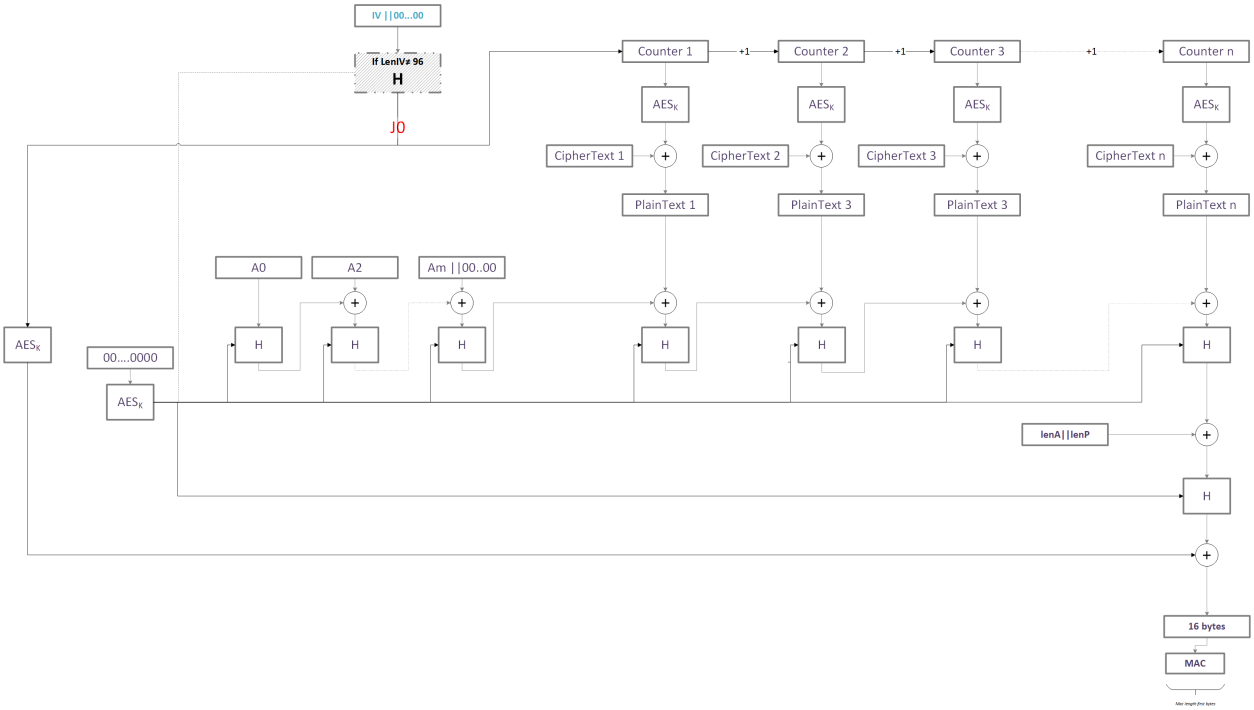
It can be seen that:

- the top of the figure corresponds to the encryption of the plaintext with an AES counter mode.
- the bottom of the figure shows the MAC generation with a Galois field multiplication (H).

When the IV is 96 bits, the initialization of the counter is mainly performed with the IV. When the IV is not 96 bits long , the IV is firstly hashed until it fits in an AES block. The grey box illustrates this case. The details are given in [1].

The Galois field multiplication involves a subkey. The subkey is generated as the encryption of the null block with the main key K.

The MAC process is completed in XORing the length of the additional data and the length of the data. The result is hashed a last time and XORed with the encryption of the IV. The decryption and MAC process is similar. The role of the ciphertext and plaintext are just inverted. Next figure shows the **Decryption and MAC** mechanism.



4.6.3.1 GMAC

GMAC is a special case of GCM mode where data is only authenticated, and not encrypted\ decrypted. It is equivalent to have additional data without any message. The library provides specific APIs to generate a GMAC.

4.6.4 Underlying AES

EM9305 embeds:

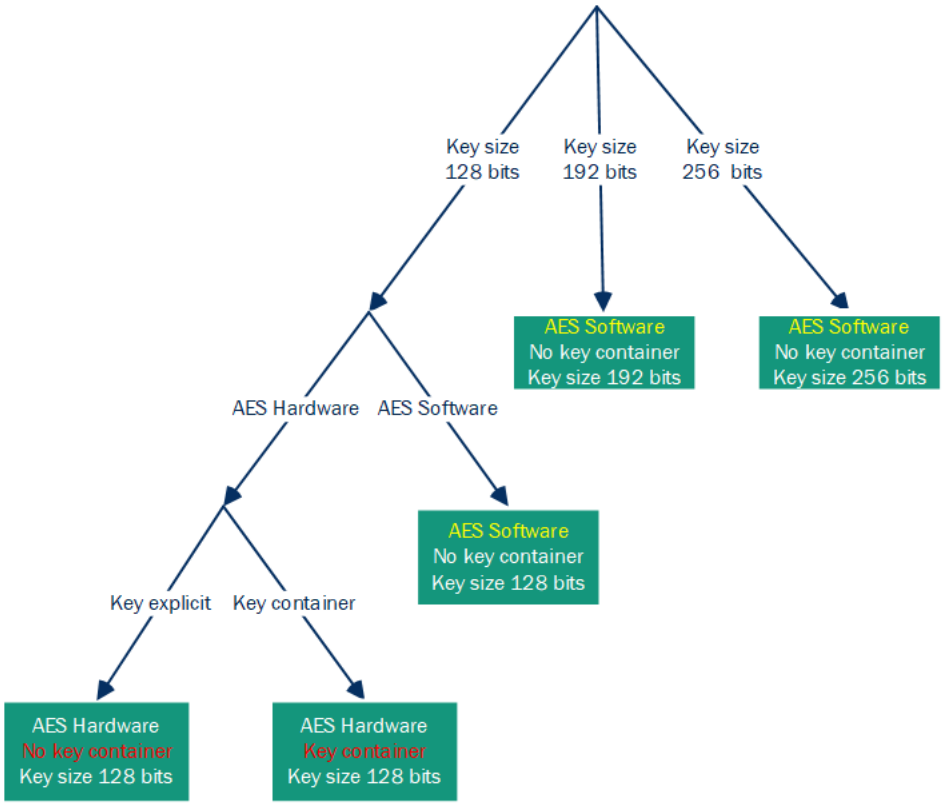
- A software AES that manages the key sizes (128, 192 and 256).
- A hardware AES that only manages 128-bit key size.

The hardware AES can be invoked:

- either with a key given explicitly
- or with a key stored in a key container. In this case, the ID of the key is provided to the AES.

AES GCM APIs interface both the AES Software and the AES hardware. When AES hardware is used, the APIs allows the use of an explicit key or the use of a key container.

Next figure shows which AES is executed, according to the various options.



4.6.5 APIs

4.6.5.1 Enumerations

4.6.5.1.1 AES Type

The underlying algorithm is chosen with the following macro: *AES type*

4.6.5.1.2 AES Key size in bits

The key size is chosen with the macro defined here: *AES Key size in bits*

4.6.5.1.3 Explicit key or key container

The choice between a key provided by value or implicitly with a key container ID is given by the following enumeration:

enum **AES_GCM_KeyType_t**

Select if the key is passed by value(explicitly) or if the key is contained in a key container.

Values:

enumerator **AES_GCM_KEY_VALUE**

Key value is provided explicitly by value.

enumerator **AES_GCM_KEY_ID**

Key value is provided by its ID. The key is in a key container.

4.6.5.1.4 Error status

The API error status are given by:

enum **AES_GCM_Lib_error_t**

Error status words for AES GCM mode.

Values:

enumerator **AES_GCM_SUCCESS**

AES GCM computation successful.

enumerator **AES_GCM_INCOMPATIBLE_PARAMETER**

Incompatible key size with key type and AES type.

enumerator **AES_GCM_INCORRECT_LENGTH**

Incorrect length- Data must be 16 bytes long.

enumerator **AES_GCM_INCORRECT_KEY_LENGTH**

Key length is incorrect.It shall be 128,192 or 256.

enumerator **AES_GCM_INCORRECT_RESULT_POINTER**

Result pointer is null.

4.6.5.2 Context

A context that contains the key size, the underlying AES , the key type , the key value \ key ID, the current counter and intermediate data is defined as follows.

struct **AES_GCM_CTX**

AES GCM context

Public Members

AES_key_size_bit_t **keySize**

Key size.

AES_Select_t **AESType**

AES Type: Either AES_HARDWARE or AES_SOFTWARE.

AES_GCM_KeyType_t **keyType**

Key type. Either KEY_ID or KEY_VALUE

uint8_t **key**[32]

The key value or the key index

uint8_t **H**[16]

The Hash subkey

uint8_t **S**[16]

Current MAC value

uint8_t **J0**[16]

J0 value

uint8_t **counter**[16]

Current counter

uint8_t **lengthA**[8]

Accumulated length of Additional data

uint8_t **lengthP**[8]

Accumulated length of plaintext or ciphertext

uint8_t **buffer**[16]

Intermediate buffer for encryption/decryption

uint8_t **nbByteReady**

Indicate how many bytes in the buffer are available to encrypt/decrypt the message before executing a new AES

uint8_t **GMAC**

Indicate if the computation is a GMAC or not

4.6.5.3 AES_GCM_InitCtx

4.6.5.3.1 Goal of the function

This function initializes an AES context with the different options: key size, underlying AES and key type. It sets the key and the initial counter value. It also initializes the calculation, computing J0.

4.6.5.3.2 Prototype

AES_GCM_Lib_error_t **AES_GCM_InitCtx**(*AES_GCM_CTX* *ctx, *AES_Select_t* AESSelect, *AES_GCM_KeyType_t* keyType, *AES_key_size_bit_t* keySize, uint8_t *Key, uint8_t lenIV[8], uint8_t *IV)

Initialize an AES GCM context. Choose the underlying AES, the key type and key size. Set the key value. Set the initial value.

Parameters

- **ctx** – **[out]** AES GCM context to set
- **AESSelect** – **[in]** The underlying AES. Either AES_HARDWARE or AES_SOFTWARE
- **keyType** – **[in]** The type of the key: Either KEY_VALUE for explicitly key value or KEY_ID for a key in a key container
- **keySize** – **[in]** The size of the key in bits : AES_KEY_128,AES_KEY_192 or AES_KEY_256
- **key** – **[in]** The key value(16 to 32 bytes) or the key ID (first byte)
- **lenIV** – **[in]** 8-byte array representing the length in bytes of the IV. E.g uint8_t lenIV[8]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0C} indicates a 12 bytes IV.
- **IV** – **[in]** Buffer of lenIV byte with the initial value

Return values

- **AES_GCM_SUCCESS** – No error occurred
- **AES_GCM_INCORRECT_KEY_LENGTH** – Incorrect key length
- **AES_GCM_INCOMPATIBLE_PARAMETER** – Hardware/software selection is not compatible with the key length

Returns Error code

4.6.5.3.3 Parameters

- **Ctx** : The AES GCM context to initialize
- **AESSelect**: Select the AES hardware or the AES software
- **keyType**: Select if the key is given explicitly by value or if it refers by its key container ID.
- **keySize**: Key size in bits
- **initialCounter** : 16-byte initial counter value
- **key:**
  - When the key is given explicitly: 16, 24 or 32 bytes representing the key value.
  - When the key is given by its ID: 1 byte with the key ID.
- **lenIV**: an 8-byte array that indicates the length in bytes of the IV.
- **IV**: Initial value of length indicated by lenIV

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_ENC_ONLY(Encryption). The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.6.5.3.4 Return values

Type	Description	OK \ NOK
AES_GCM_SUCCESS	Initialization successful	OK
AES_GCM_INCORRECT_KEY_LENGTH	Incorrect key length	NOK
AES_GCM_INCOMPATIBLE_PARAMETER	Hardware AES is not compatible with this key length	NOK

4.6.5.4 AES_GCM_Hash_Additional_Data

4.6.5.4.1 Goal of the function

This function accumulates data in intermediate buffer and hashes it when the buffer is full. It performs the Galois Field multiplication.

4.6.5.4.2 Prototype

AES_GCM_Lib_error_t **AES_GCM_Hash_Additional_Data**(*AES_GCM_CTX* *ctx, uint8_t lenA[8], uint8_t *A)

Hash the additional data that precedes the data to encrypt/decrypt.

Note: This function can be called several times consecutively with different parts of the additional data. LenA does not represent the total length of the data. It represents the length of the data provided for this call.

Note: This function can be skipped in case there is no additional data.

Parameters

- **ctx** – **[inout]** AES GCM
- **lenA** – **[in]** 8-byte array representing the length in bytes of the additional data provided at this call. E.g uint8_t lenIV[8]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x00} indicates a 256-byte additional data
- **A** – **[in]** Buffer of lenA byte with the additional data

Return values **AES_GCM_SUCCESS** – No error occurred

Returns Error code

4.6.5.4.3 Parameters

- **Ctx** : The AES GCM context
- **lenA**: Array representing the length in byte of the data provided during this call. It can be any size from 0 byte to 0xFFFFFFFFFFFFFFFF bytes.
- **A**: Buffer of lenA bytes with the additional data provided for this call.

Note:

This function can:

- be called once with all the additional data.
- be called several times consecutively, providing parts of the additional data only. In that case lenA is the size of the data provided during this call.

4.6.5.4.4 Return values

Type	Description	OK \ NOK
AES_GCM_SUCCESS	No error, data accumulated	OK

4.6.5.5 AES_GCM_EncryptAndMAC

4.6.5.5.1 Goal of the function

This function encrypts the data provided and updates the MAC accordingly.

4.6.5.5.2 Prototype

AES_GCM_Lib_error_t **AES_GCM_EncryptAndMAC**(*AES_GCM_CTX* *ctx, uint8_t lenP[8], uint8_t *plainText, uint8_t *cipherText)

Encrpyt the provided plaintext and update the MAC.

Note: This function can be called several times consecutively with different parts of the plaintext. LenP does not represent the total length of the data. It represents the length of the data provided for this call.

Parameters

- **ctx** – **[inout]** AES GCM
- **lenP** – **[in]** 8-byte array representing the length in bytes of the plaintext data provided at this call. E.g uint8_t lenP[8]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x00} indicates a 512-byte plaintext.
- **plainText** – **[in]** Buffer of lenP byte with the data to encrypt
- **cipherText** – **[out]** Buffer of lenP byte with the encrypted data

Return values **AES_GCM_SUCCESS** – No error occurred

Returns Error code

4.6.5.5.3 Parameters

- **Ctx** : The AES GCM context
- **lenP**: Array representing the length in bytes of the plaintext provided during this call. It can be any size from 0 bytes to 0x0000007F FFFFFF00 bytes.
- **plainText**: Buffer of lenP bytes with the plaintext to encrypt this call.
- **cipherText**: Buffer of lenP bytes receiving the encrypted message.

Note:

This function can:

- be called once with the complete plaintext.
- be called several times consecutively, providing parts of the message only. In that case lenP is the size of the data provided during this call.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type AES_ENC_ONLY(Encryption). The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.6.5.5.4 Return values

Table 4: :header: “Type”, “Description”, “OK \\ NOK” :widths: 50,25,15

AES_GCM_SUCCESS	Successful encryption	OK
AES_GCM_INCORRECT_RESULT_POINTER	Ciphertext pointer is not initialized	NOK
AES_GCM_INCORRECT_LENGTH	lenP is too big	NOK

4.6.5.6 AES_GCM_DecryptAndMAC

4.6.5.6.1 Goal of the function

This function decrypts the data provided and updates the MAC accordingly.

4.6.5.6.2 Prototype

AES_GCM_Lib_error_t **AES_GCM_DecryptAndMAC**(*AES_GCM_CTX* *ctx, uint8_t lenC[8], uint8_t *cipherText, uint8_t *plainText)

Decrypt the provided ciphertext and update the MAC.

Note: This function can be called several times consecutively with different parts of the ciphertext. LenC does not represent the total length of the data. It represents the length of the data provided for this call.

Parameters

- **ctx** – **[inout]** AES GCM
- **lenC** – **[in]** 8-byte array representing the length in bytes of the cipherText data provided at this call. E.g uint8_t lenC[8]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x00} indicates a 512-byte plaintext.
- **cipherText** – **[in]** Buffer of lenC byte with the data to decrypt
- **plainText** – **[out]** Buffer of lenC byte with the decrypted data

Return values

- **AES_GCM_SUCCESS** – No error occurred
- **AES_GCM_INCORRECT_RESULT_POINTER** – Result pointer not initialized
- **AES_GCM_INCORRECT_LENGTH** – The length of the message should be snaller than 2^39 -256 = 000000 7F FF FF FF 00

Returns Error code

4.6.5.6.3 Parameters

- **Ctx** : The AES GCM context
- **lenC**: Array representing the length in bytes of the ciphertext provided during this call. It can be any size from 0 byte to 0x0000007F FFFFFF00 bytes.
- **cipherText**: Buffer of lenC bytes with the ciphertext to decrypt this call.
- **plainText**: Buffer of lenC bytes receiving the decrypted message.

Note:

This function can:

- be called once with the complete ciphertext.
- be called several times consecutively, providing parts of the message only. In that case lenC is the size of the data provided during this call.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type **AES_ENC_ONLY(Encryption)**. As the AES is performed in encryption, a key typed **AES_DEC_ONLY(decryption)** would not work. The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.6.5.6.4 Return values

Type	Description	OK \ NOK
AES_GCM_SUCCESS	Successful decryption	OK
AES_GCM_INCORRECT_RESULT_POINTER	plainText pointer is not initialized	NOK
AES_GCM_INCORRECT_LENGTH	lenC is too big	NOK

4.6.5.7 AES_GCM_GMAC

4.6.5.7.1 Goal of the function

This function accumulates data in intermediate buffer and hashes it when the buffer is full. It performs the Galois Field multiplications for the MAC computation.

4.6.5.7.2 Prototype

AES_GCM_Lib_error_t **AES_GCM_GMAC**(*AES_GCM_CTX* *ctx, uint8_t lenD[8], uint8_t *data)

Perform a GMAC computation (GCM mode with no encryption)

Note: This function can be called several times consecutively with different parts of the additional data. LenD does not represent the total length of the data. It represents the length of the data provided for this call.

Parameters

- **ctx** – **[inout]** AES GCM
- **lenD** – **[in]** 8-byte array representing the length in bytes of the data provided at this call. E.g uint8_t lenD[8]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x00} indicates a 512-byte plaintext.
- **data** – **[in]** Buffer of lenD byte with the data to MAC

Return values

- **AES_GCM_SUCCESS** – No error occurred
- **AES_GCM_INCORRECT_RESULT_POINTER** – Result pointer not initialized
- **AES_GCM_INCORRECT_LENGTH** – The length of the message should be smaller than 2^39 -256 = 000000 7F FF FF FF 00

Returns Error code

4.6.5.7.3 Parameters

- **Ctx** : The AES GCM context
- **lenA**: Array representing the length in bytes of the data provided during this call. It can be any size from 0 byte to 0xFFFFFFFFFFFFFFFF bytes.
- **A**: Buffer of lenA bytes with the additional data provided at this call.

Note:

This function can:

- be called once with all the additional data.
- be called several times consecutively, providing parts of the additional data only. In that case lenA is the size of the data provided during this call.

4.6.5.7.4 Return values

Type	Description	OK \ NOK
AES_GCM_SUCCESS	No error, data accumulated	OK

4.6.5.8 AES_GCM_GetMAC

4.6.5.8.1 Goal of the function

This function completes the MAC computation and returns the final MAC.

4.6.5.8.2 Prototype

AES_GCM_Lib_error_t **AES_GCM_GetMAC**(*AES_GCM_CTX* *Ctx, uint8_t lenMAC, uint8_t *MAC)

Finalize the MAC computation once all the data have been provided.

Parameters

- **ctx** – **[inout]** AES GCM
- **lenMAC** – **[in]** Length in bytes of the MAC. It must be a value in range [4..16].
- **MAC** – **[out]** Buffer to receive the MAC. It must be at least of length LenMAC.

Return values

- **AES_GCM_SUCCESS** – No error occurred
- **AES_GCM_INCORRECT_LENGTH** – The MAC length shall be in range [4..16].
- **AES_GCM_INCORRECT_RESULT_POINTER** – MAC pointer not initialized

Returns Error code

4.6.5.8.3 Parameters

- **Ctx** : The AES GCM context
- **lenMAC**: length in bytes of the desired MAC. It must be a value between 4 and 16.
- **MAC**: Buffer to receive the MAC value.

Warning: When using a key container, the key must be either of type AES_HW_ENC_DEC (encryption and decryption) or type **AES_ENC_ONLY(Encryption)**. As the AES is performed in encryption, a key typed **AES_DEC_ONLY(decryption)** would not work. The key must not have been invalidated. In case those conditions are not respected, this function returns an error of type AES_Error_t.

4.6.5.8.4 Return values

Type	Description	OK \ NOK
AES_GCM_SUCCESS	No error, data accumulated	OK
AES_GCM_INCORRECT_LENGTH	MAC length is incorrect. It should be in [4..16]	NOK
AES_GCM_INCORRECT_RESULT_POINTER	MAC pointer is not initialized	NOK

4.6.6 Performances

4.6.6.1 Library location

The lib is located in ROM.

4.6.6.2 Code size

Size in bytes
1672

4.6.6.3 RAM

Size in bytes
No usage of global RAM except the AES_GCM_CTX

4.6.6.4 Stack

Size in bytes
176 bytes

4.6.6.5 Execution time

The AES_GCM_InitCtx function performs an AES. Therefore, the time depends on the configuration.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_GCM_InitCtx	AES Hardware	128	25
AES_GCM_InitCtx	AES Software	128	129
AES_GCM_InitCtx	AES Software	192	153
AES_GCM_InitCtx	AES Software	256	171

The execution time AES_GCM_Hash_Additional_Data depends on the size of the additional data that is processed during this call. Because the Galois multiplication is performed on data of 16 bytes, the operation is performed only every 16 bytes. It induces that the number of multiplications is roughly the [NbByte16] or[NbByte16]-1. The hash of the data does not involve AES so that the key size is not impacting the timing. Next table shows the time for 16 bytes.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_GCM_Hash_Additional_Data	AES	Any	718

The function AES_GCM_GMAC is fundamentally the same as AES_GCM_Hash_Additional_Data. The performances are identical. Next table shows the time for 16 bytes.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_GCM_GMAC	AES	Any	718

The performance of the function AES_GCM_EncryptAndMAC depends on the size of the plaintext, the underlying AES and also how many bytes were accumulated in the temporary buffer for the MAC computation. It also depends if all the additional data were handled in AES_GCM_Hash_Additional_Data. Globally, one Galois field multiplication and one AES is performed per block of 16 bytes.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_GCM_EncryptAndMAC	AES Hardware	128	917
AES_GCM_EncryptAndMAC	AES Software	128	1022
AES_GCM_EncryptAndMAC	AES Software	192	1046
AES_GCM_EncryptAndMAC	AES Software	256	1064

The performances of the function AES_GCM_DecryptAndMAC are the same for the encryption and mainly depend on the data size. For 16 bytes of data to decrypt we have the following figures:

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_GCM_DecryptAndMAC	AES Hardware	128	917
AES_GCM_DecryptAndMAC	AES Software	128	1022
AES_GCM_DecryptAndMAC	AES Software	192	1046
AES_GCM_DecryptAndMAC	AES Software	256	1064

Finally AES_GCM_GetMAC always involves a Galois field multiplication and an AES computation. It may involves a second multiplication if it was not performed previously in the encrypt\decrypt functions. Below are the figures for the case with 1 multiplication.

Function	Underlying algorithm	Key size in bits	Time in us at 48Mhz
AES_GCM_GetMAC	AES Hardware	128	908
AES_GCM_GetMAC	AES Software	128	1013
AES_GCM_GetMAC	AES Software	192	1035
AES_GCM_GetMAC	AES Software	256	1094

4.6.7 Dependencies

AES GCM lib depends on the AES lib.

4.6.8 Example

Next code shows basic examples of GCM algorithm

- using the AES hardware with an explicit key
- using the AES hardware with key stored in key containers
- using the AES software
- with various size for the additional data
- with various size of plaintext \ ciphertext

It also shows a GMAC example.

```
////////////////////////////////////////////////////////////////////////////////
///
/// @file       ExampleAES_GCM.c
///
/// @project    T9305
///
/// @author     SAS
///
/// @brief           Example of use of AES GCM library
///
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
///
////////////////////////////////////////////////////////////////////////////////
///
/// @copyright Copyright (C) 2022 EM Microelectronic
/// @cond
///
/// All rights reserved.
///
/// Redistribution and use in source and binary forms, with or without
/// modification, are permitted provided that the following conditions are met:
/// 1. Redistributions of source code must retain the above copyright notice,
/// this list of conditions and the following disclaimer.
/// 2. Redistributions in binary form must reproduce the above copyright notice,
/// this list of conditions and the following disclaimer in the documentation
/// and/or other materials provided with the distribution.
///
////////////////////////////////////////////////////////////////////////////////
///
/// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
/// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
/// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
/// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
```

(continues on next page)

(continued from previous page)

```
/// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
/// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
/// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
/// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
/// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
/// POSSIBILITY OF SUCH DAMAGE.
/// @endcond
////////////////////////////////////////////////////////////////////////////////
#include <stdint.h>
#include "AES.h"
#include "AES_GCM.h"
#include "hw_aes.h"

uint8_t ExampleAES_GCM() {

        AES_GCM_CTX ctx;
        AES_GCM_Lib_error_t sw;
        uint16_t error = 0;
        uint8_t i;
        uint8_t Cipher[0x40];
        uint8_t Decipher[0x40];
        uint8_t MAC[0x10];
        uint8_t keyId[1];

        //test vector with a 128-bit key
#define TLEN 0x0C
#define PLEN 0x3C
#define ALEN 0x14
#define IVLEN 0x0C

        uint8_t Key[AES_BYTE_SIZE_KEY128] = { 0xFE, 0xFF, 0xE9, 0x92, 0x86, 0x65,
                        0x73, 0x1C, 0x6D, 0x6A, 0x8F, 0x94, 0x67, 0x30, 0x83, 0x08 };
        uint8_t Plaintext[PLEN] = { 0xD9, 0x31, 0x32, 0x25, 0xF8, 0x84, 0x06, 0xE5,
                        0xA5, 0x59, 0x09, 0xC5, 0xAF, 0xF5, 0x26, 0x9A, 0x86, 0xA7, 0xA9,
                        0x53, 0x15, 0x34, 0xF7, 0xDA, 0x2E, 0x4C, 0x30, 0x3D, 0x8A, 0x31,
                        0x8A, 0x72, 0x1C, 0x3C, 0x0C, 0x95, 0x95, 0x68, 0x09, 0x53, 0x2F,
                        0xCF, 0x0E, 0x24, 0x49, 0xA6, 0xB5, 0x25, 0xB1, 0x6A, 0xED, 0xF5,
                        0xAA, 0x0D, 0xE6, 0x57, 0xBA, 0x63, 0x7B, 0x39 };
        uint8_t Additional[ALEN] = { 0x3A, 0xD7, 0x7B, 0xB4, 0x0D, 0x7A, 0x36, 0x60,
                        0xA8, 0x9E, 0xCA, 0xF3, 0x24, 0x66, 0xEF, 0x97, 0xF5, 0xD3, 0xD5,
                        0x85 };
        uint8_t LengthA[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ALEN };
        uint8_t LengthIV[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, IVLEN };
        uint8_t IV[IVLEN] = { 0xCA, 0xFE, 0xBA, 0xBE, 0xFA, 0xCE, 0xDB, 0xAD, 0xDE,
                        0xCA, 0xF8, 0x88 };
        uint8_t LengthP[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, PLEN };
        uint8_t ExpectedCipher[PLEN] = { 0x42, 0x83, 0x1E, 0xC2, 0x21, 0x77, 0x74,
                        0x24, 0x4B, 0x72, 0x21, 0xB7, 0x84, 0xD0, 0xD4, 0x9C, 0xE3, 0xAA,
                        0x21, 0x2F, 0x2C, 0x02, 0xA4, 0xE0, 0x35, 0xC1, 0x7E, 0x23, 0x29,
                        0xAC, 0xA1, 0x2E, 0x21, 0xD5, 0x14, 0xB2, 0x54, 0x66, 0x93, 0x1C,
                        0x7D, 0x8F, 0x6A, 0x5A, 0xAC, 0x84, 0xAA, 0x05, 0x1B, 0xA3, 0x0B,
                        0x39, 0x6A, 0x0A, 0xAC, 0x97, 0x3D, 0x58, 0xE0, 0x91 };
```

(continues on next page)

(continued from previous page)

```
        uint8_t ExpectedMAC[TLEN] = { 0xF0, 0x7C, 0x25, 0x28, 0xEE, 0xA2, 0xFC,
                        0xA1, 0x21, 0x1F, 0x90, 0x5E };

        //test vector with 192 bit key
#define TLEN192 0x10
#define PLEN192 0x40
#define ALEN192 0x40
#define IVLEN192 0x0C
        uint8_t Key192[AES_BYTE_SIZE_KEY192] = { 0xFE, 0xFF, 0xE9, 0x92, 0x86, 0x65,
                        0x73, 0x1C, 0x6D, 0x6A, 0x8F, 0x94, 0x67, 0x30, 0x83, 0x08, 0xFE,
                        0xFF, 0xE9, 0x92, 0x86, 0x65, 0x73, 0x1C };
        uint8_t Plaintext192[PLEN192] = { 0xD9, 0x31, 0x32, 0x25, 0xF8, 0x84, 0x06,
                        0xE5, 0xA5, 0x59, 0x09, 0xC5, 0xAF, 0xF5, 0x26, 0x9A, 0x86, 0xA7,
                        0xA9, 0x53, 0x15, 0x34, 0xF7, 0xDA, 0x2E, 0x4C, 0x30, 0x3D, 0x8A,
                        0x31, 0x8A, 0x72, 0x1C, 0x3C, 0x0C, 0x95, 0x95, 0x68, 0x09, 0x53,
                        0x2F, 0xCF, 0x0E, 0x24, 0x49, 0xA6, 0xB5, 0x25, 0xB1, 0x6A, 0xED,
                        0xF5, 0xAA, 0x0D, 0xE6, 0x57, 0xBA, 0x63, 0x7B, 0x39, 0x1A, 0xAF,
                        0xD2, 0x55 };
        uint8_t Additional192[ALEN192] = { 0x3A, 0xD7, 0x7B, 0xB4, 0x0D, 0x7A, 0x36,
                        0x60, 0xA8, 0x9E, 0xCA, 0xF3, 0x24, 0x66, 0xEF, 0x97, 0xF5, 0xD3,
                        0xD5, 0x85, 0x03, 0xB9, 0x69, 0x9D, 0xE7, 0x85, 0x89, 0x5A, 0x96,
                        0xFD, 0xBA, 0xAF, 0x43, 0xB1, 0xCD, 0x7F, 0x59, 0x8E, 0xCE, 0x23,
                        0x88, 0x1B, 0x00, 0xE3, 0xED, 0x03, 0x06, 0x88, 0x7B, 0x0C, 0x78,
                        0x5E, 0x27, 0xE8, 0xAD, 0x3F, 0x82, 0x23, 0x20, 0x71, 0x04, 0x72,
                        0x5D, 0xD4 };
        uint8_t LengthA192[8] =
                        { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ALEN192 };
        uint8_t LengthIV192[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        IVLEN192 };
        uint8_t IV192[IVLEN192] = { 0xCA, 0xFE, 0xBA, 0xBE, 0xFA, 0xCE, 0xDB, 0xAD,
                        0xDE, 0xCA, 0xF8, 0x88 };
        uint8_t LengthP192[8] =
                        { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, PLEN192 };
        uint8_t ExpectedCipher192[PLEN192] = { 0x39, 0x80, 0xCA, 0x0B, 0x3C, 0x00,
                        0xE8, 0x41, 0xEB, 0x06, 0xFA, 0xC4, 0x87, 0x2A, 0x27, 0x57, 0x85,
                        0x9E, 0x1C, 0xEA, 0xA6, 0xEF, 0xD9, 0x84, 0x62, 0x85, 0x93, 0xB4,
                        0x0C, 0xA1, 0xE1, 0x9C, 0x7D, 0x77, 0x3D, 0x00, 0xC1, 0x44, 0xC5,
                        0x25, 0xAC, 0x61, 0x9D, 0x18, 0xC8, 0x4A, 0x3F, 0x47, 0x18, 0xE2,
                        0x44, 0x8B, 0x2F, 0xE3, 0x24, 0xD9, 0xCC, 0xDA, 0x27, 0x10, 0xAC,
                        0xAD, 0xE2, 0x56 };
        uint8_t ExpectedMAC192[TLEN192] = { 0x3B, 0x91, 0x53, 0xB4, 0xE7, 0x31,
                        0x8A, 0x5F, 0x3B, 0xBE, 0xAC, 0x10, 0x8F, 0x8A, 0x8E, 0xDB };

        //test vector with 256 bit key
#define TLEN256 0x10
#define ALEN256 0x40
#define IVLEN256 0x0C
        uint8_t Key256[AES_BYTE_SIZE_KEY256] = { 0xFE, 0xFF, 0xE9, 0x92, 0x86, 0x65,
                        0x73, 0x1C, 0x6D, 0x6A, 0x8F, 0x94, 0x67, 0x30, 0x83, 0x08, 0xFE,
                        0xFF, 0xE9, 0x92, 0x86, 0x65, 0x73, 0x1C, 0x6D, 0x6A, 0x8F, 0x94,
                        0x67, 0x30, 0x83, 0x08 };
```

(continues on next page)

(continued from previous page)

```
        uint8_t Additional256[ALEN256] = { 0x3A, 0xD7, 0x7B, 0xB4, 0x0D, 0x7A, 0x36,
                        0x60, 0xA8, 0x9E, 0xCA, 0xF3, 0x24, 0x66, 0xEF, 0x97, 0xF5, 0xD3,
                        0xD5, 0x85, 0x03, 0xB9, 0x69, 0x9D, 0xE7, 0x85, 0x89, 0x5A, 0x96,
                        0xFD, 0xBA, 0xAF, 0x43, 0xB1, 0xCD, 0x7F, 0x59, 0x8E, 0xCE, 0x23,
                        0x88, 0x1B, 0x00, 0xE3, 0xED, 0x03, 0x06, 0x88, 0x7B, 0x0C, 0x78,
                        0x5E, 0x27, 0xE8, 0xAD, 0x3F, 0x82, 0x23, 0x20, 0x71, 0x04, 0x72,
                        0x5D, 0xD4 };
        uint8_t LengthA256[8] =
                        { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ALEN256 };
        uint8_t LengthIV256[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        IVLEN256 };
        uint8_t IV256[IVLEN256] = { 0xCA, 0xFE, 0xBA, 0xBE, 0xFA, 0xCE, 0xDB, 0xAD,
                        0xDE, 0xCA, 0xF8, 0x88 };
        uint8_t ExpectedMAC256[TLEN256] = { 0xDE, 0x34, 0xB6, 0xDC, 0xD4, 0xCE,
                        0xE2, 0xFD, 0xBE, 0xC3, 0xCE, 0xA0, 0x1A, 0xF1, 0xEE, 0x44 };

        //-------------------------------------------------------------------------
        // First example
        //
        // Perform AES GCM in encryption + MAC mode
        //
        // The key is 128-bit, we use the AES hardware without key container.
        //
        // MAC length is 12 bytes. IV length is 12 bytes.
        //
        // The additional data are 20 bytes. The message to encrypt is 60 bytes.
        //
        // The data are encrypted and MAC in one shot
        //-------------------------------------------------------------------------

        //Initialization:
        //-using a 128 bit key
        //-not using the key container
        //using AES_HARDWARE
        //the length of the message and the length of the additional data are the total␣
↪lengths
        sw = AES_GCM_InitCtx(&ctx, AES_HARDWARE, AES_GCM_KEY_VALUE, AES_KEY_128,
                        Key, LengthIV, IV);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Deal with the additional data in one shot
        sw = AES_GCM_Hash_Additional_Data(&ctx, LengthA, Additional);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Encrpyt the message in one shot
        sw = AES_GCM_EncryptAndMAC(&ctx, LengthP, Plaintext, Cipher);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //get the final MAC
```

(continues on next page)

(continued from previous page)

```
        sw = AES_GCM_GetMAC(&ctx, TLEN, MAC);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Check the result is as expected
        for (i = 0; i < PLEN; i++) {
                if (Cipher[i] != ExpectedCipher[i])
                        error++;
        }
        //Check the MAC is as expected
        for (i = 0; i > TLEN; i++) {
                if (MAC[i] != ExpectedMAC[i])
                        error++;
        }

        //------------------------------------------------------------------------
        // Second example
        //
        // Perform AES GCM in decryption + MAC mode
        //
        // The key is 128-bit, we use the AES software
        //
        // MAC length is 12 bytes. IV length is 12 bytes.
        //
        // The additional data are 20 bytes. The message to encrypt is 60 bytes.
        //
        // We deal with the additional data in two steps(5 bytes then 15 bytes)
        // and we encrypt the message in 3 steps(17, 8 , 35)
        //------------------------------------------------------------------------
        uint8_t LengthA1[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 5 }; //Size of␣
↪the first part of A=5 bytes
        uint8_t LengthA2[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ALEN - 5 }; //
↪Size of the second part of A=20-5=15 bytes
        uint8_t LengthP1[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 17 }; //Size␣
↪of the first part of A
        uint8_t LengthP2[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 8 };
        uint8_t LengthP3[8] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, PLEN - 17
                        - 8 };

        //Initialization:
        //-using a 128 bit key
        //-not using the key container
        //using AES_SOFTWARE
        //the length of the message and the length of the additional data are the total␣
↪lengths
        sw = AES_GCM_InitCtx(&ctx, AES_SOFTWARE, AES_GCM_KEY_VALUE, AES_KEY_128,
                        Key, LengthIV, IV);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Introduce the first part of the additional data- LengthA1+LengthA2=LengthA
        sw = AES_GCM_Hash_Additional_Data(&ctx, LengthA1, Additional);
```

(continues on next page)

(continued from previous page)

```
        if (sw != AES_GCM_SUCCESS)
                error++;
        //Introduce the second part of the additional data
        sw = AES_GCM_Hash_Additional_Data(&ctx, LengthA2, Additional + 5);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //decrypt a first part of the cipherLengthP1+LengthP2=LengthP
        sw = AES_GCM_DecryptAndMAC(&ctx, LengthP1, ExpectedCipher, Decipher);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //decrypt a first part of the cipher
        sw = AES_GCM_DecryptAndMAC(&ctx, LengthP2, ExpectedCipher + 17,
                        Decipher + 17);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //decrypt a first part of the cipher
        sw = AES_GCM_DecryptAndMAC(&ctx, LengthP3, ExpectedCipher + 17 + 8,
                        Decipher + 17 + 8);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //get the final MAC
        sw = AES_GCM_GetMAC(&ctx, TLEN, MAC);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Check the result is as expected
        for (i = 0; i < PLEN; i++) {
                if (Decipher[i] != Plaintext[i])
                        error++;
        }

        //Check the MAC is as expected
        for (i = 0; i > TLEN; i++) {
                if (MAC[i] != ExpectedMAC[i])
                        error++;
        }

        //------------------------------------------------------------------------
        // Third example
        //
        // Perform AES GCM in encryption + MAC mode
        //
        // The key is 128-bit, we use the AES hardware WITH key container.
        //
        /// MAC length is 12 bytes. IV length is 12 bytes.
        //
        // The additional data are 20 bytes. The message to encrypt is 60 bytes.
        //
```

(continues on next page)

(continued from previous page)

```
        // The data are encrypted and MAC in one shot
        //-------------------------------------------------------------------------

        //A priori the key has been written previously in a key container.
        //Here we write the key in container 0x05
        AES_SetKeyContainer((uint32_t*) Key, 0x05, AES_HW_ENC_DEC);
        keyId[0] = 0x05;

        //Initialization:
        //-using a 128 bit key
        //-using the key container
        //using AES_HARDWARE
        //the length of the message and the length of the additional data are the total␣
↪lengths
        sw = AES_GCM_InitCtx(&ctx, AES_HARDWARE, AES_GCM_KEY_ID, AES_KEY_128, keyId,
                        LengthIV, IV);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Deal with the additional data in one shot
        sw = AES_GCM_Hash_Additional_Data(&ctx, LengthA, Additional);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Encrpyt the message in one shot
        sw = AES_GCM_EncryptAndMAC(&ctx, LengthP, Plaintext, Cipher);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //get the final MAC
        sw = AES_GCM_GetMAC(&ctx, TLEN, MAC);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Check the result is as expected
        for (i = 0; i < PLEN; i++) {
                if (Cipher[i] != ExpectedCipher[i])
                        error++;
        }
        //Check the MAC is as expected
        for (i = 0; i > TLEN; i++) {
                if (MAC[i] != ExpectedMAC[i])
                        error++;
        }

        //-------------------------------------------------------------------------
        // Fourth example
        //
        // Perform AES GCM in encryption + MAC mode
        //
        // The key is 192-bit, we use the AES software
        //
```

(continues on next page)

(continued from previous page)

```
        // MAC length is 16 bytes. IV length is 12 bytes.
        //
        // The additional data are 48 bytes. The message to encrpyt is 48 bytes.
        //
        // The data are encrypted and MAC in one shot
        //------------------------------------------------------------------------
        //the length of the message and the length of the additional data are the total␣
↪lengths
        sw = AES_GCM_InitCtx(&ctx, AES_SOFTWARE, AES_GCM_KEY_VALUE, AES_KEY_192,
                        Key192, LengthIV192, IV192);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Deal with the additional data in one shot
        sw = AES_GCM_Hash_Additional_Data(&ctx, LengthA192, Additional192);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Encrpyt the message in one shot
        sw = AES_GCM_EncryptAndMAC(&ctx, LengthP192, Plaintext192, Cipher);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //get the final MAC
        sw = AES_GCM_GetMAC(&ctx, TLEN192, MAC);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Check the result is as expected
        for (i = 0; i < PLEN192; i++) {
                if (Cipher[i] != ExpectedCipher192[i])
                        error++;
        }
        //Check the MAC is as expected
        for (i = 0; i > TLEN192; i++) {
                if (MAC[i] != ExpectedMAC192[i])
                        error++;
        }

        //------------------------------------------------------------------------
        // Fifth example
        //
        // Perform AESGMAC
        //
        // The key is 256-bit, we use the AES software
        //
        // MAC length is 16 bytes. IV length is 12 bytes.
        //
        // The additional data are 64 bytes.
        //
        // The data are MACed in one shot. Note that we could also perform the AES_GCM_
↪GMAC function
```

(continues on next page)

(continued from previous page)

```
        //in several calls.
        //-------------------------------------------------------------------------
        sw = AES_GCM_InitCtx(&ctx, AES_SOFTWARE, AES_GCM_KEY_VALUE, AES_KEY_256,
                        Key256, LengthIV256, IV256);
        if (sw != AES_GCM_SUCCESS)
                error++;
        //Deal with the additional data in one shot
        sw = AES_GCM_GMAC(&ctx, LengthA256, Additional256);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //get the final MAC
        sw = AES_GCM_GetMAC(&ctx, TLEN256, MAC);
        if (sw != AES_GCM_SUCCESS)
                error++;

        //Check the MAC is as expected
        for (i = 0; i > TLEN256; i++) {
                if (MAC[i] != ExpectedMAC256[i])
                        error++;
        }

        if (error != 0)
                return (1);
        else
                return (0);
}
```

CHAPTER

FIVE

PRNG (PSEUDO RANDOM NUMBER GENERATION)

5.1 Bibliography

[1] NIST SP 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generator

NIST SP 800-90A: https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final

[2] NIST SP 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation

NIST SP 800-90B https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90B.pdf

[3] NIST SP 800-90C: Recommendation for Random Bit Generator (RBG) Construction

NIST SP 800-90C: https://csrc.nist.gov/CSRC/media/Publications/sp/800-90c/draft/documents/sp800_90c_second_draft.pdf

5.2 Goal of the document

The goal of this document is to describe the functionality of the **PRNG** library.

This chapter describes:

- the principle of PRNG, especially the combination of the hardware seed and the software post-treatment.
- all the supported functions offered by the library
- the prototypes of the functions, the parameters, the error code, etc..
- the performances of the functions.

5.3 Reminder on the random number generation

5.3.1 Need of high quality random numbers

Random numbers generation is a cornerstone primitive for all the cryptographic protocols. Among other usages, random numbers are needed for:

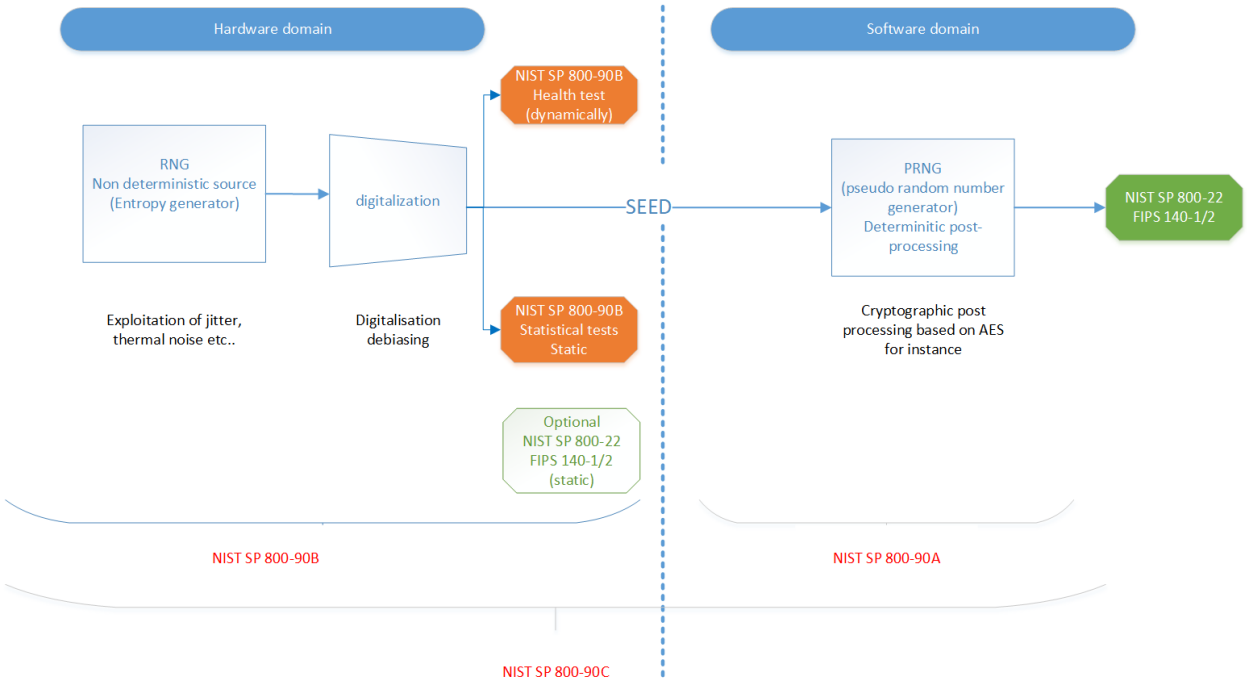
- Secret key generation
- Cryptographic challenges for authentication
- Salt and nonces generation
- Random initial values

- Etc…

For that reason, the PRNG.lib aims at providing high quality, i.e. cryptographic quality random numbers.

5.3.2 General approach

The general approach of EM9305 is to use a mix of hardware random numbers generation and a post processing performed in software. Next figure shows the general model of the PRNG.



2 phases can be distinguished:

- A **first phase** consists in generating random numbers with the **hardware**. This phase exploits physical noise(jitter, thermal noise, etc…). After digitalization, the random numbers generated constitute a seed. This phase is not deterministic.
- The **second phase** consists in **post-processing** the seed, with a software post-treatment based on AES.

We chose to be compliant with the NIST standards[1,2,3]. The overall model is compliant with the draft version of NIST SP800-90C[3]. The post-treatment is compliant with NIST SP800-90A[1]. The test of the first phase is compliant with NIST SP800-90B[2].

Regarding the hardware, four distinct hardware RNG cores are integrated in EM9305:

- Core0
- Core1
- Core2
- Core3.

The RNG cores differ by the their oscillators sizes. Core0 is the fastest core and Core3 the slowest one. When initializing the context, one can select the core that is used to seed the RNG (see SetPRNGCtx API)

For the post-treatment we use the DRBG_CTR methodology defined in [1] with AES-128 as underlying algorithm.

5.3.3 Health tests

Optionally the user can choose to run health tests on the hardware seed that is generated during InitPRNG execution. Those tests are defined in [2] and are performed on-line.

The health tests aim at detecting critical defaults of the random sequence generated. Typically, it would detect a stuck RNG.

The health tests shall be distinguished from:

- The **production tests:** During production, the logic of each part is tested. If the RNG does not work correctly, it means that the chip is defective and is therefore scraped.
- The **RNG qualification tests:** For those tests, millions of random numbers are picked, in all conditions : low, room, high temperatures, all corners, all voltage conditions etc.. The objective is to characterize the design of the RNG and its stability in various environment. It also help determining the min-entropy of the hardware cores.

Regarding health tests, two on-the-fly tests are implemented according to the NIST SP 800-90B [2] specification.

- **Repetitive health test:** this test verifies that there is no long run of the same byte value. If the same value appears consecutively, it might mean the RNG is stuck.
- **Adaptive health test:** this test verifies that a byte value does not appear too often within the 512-byte sequence. If it is the case, it might mean that the RNG is looping on itself, or at least its entropy is low.

Health tests are statistical tests. Therefore, health tests can generate **false-positive** even in the RNG works correctly. For instance, it might happen that the same byte appears 5 times consecutively in a stream of 512 bytes. So health tests may occasionally generate errors. This is inherent to the health tests and the nature of a random generator.

To circumvent this pitfall, two measures are taken:

- Both health tests have been parametrized to have a false-positive rate of *1 over 1 million*. There is a tradeoff when choosing the tests parameters. Indeed too stringent tests would fail with higher probability. Tests that are more permissive would not detect any default and therefore would be useless.
- In case of failure of the health tests, InitPRNG picks a new sequence of random numbers and tests it again. Altogether, 3 trials are performed. If after 3 trials, the sequence of random numbers still fails the health tests, it certainly means there is a major breakdown of the RNG. In this case, InitPRNG reports an error status.

5.3.4 Characterization and validation

The hardware seed is characterized with 3 different statistical tests:

- FIPS 140-1 and FIPS 140-2 ad-hoc statistical test suites
- NIST SP 800-90B [2] statistical tests suite
- NIST SP 800-22 statistical tests suite

The final outcome after the post-processing is qualified against:

- NIST SP 800-22 statistical test suite.

5.4 APIs

5.4.1 Context

A **PRNG_CTX** context structure is defined as the combination of options and working buffers used for the NIST SP 800-90A post-processing. The structure is as follows.

struct **PRNG_CTX**

Structure of PRNG context

Public Members

uint8_t **ConfigAES**

Choose AES SOFTWARE or AES HARDWARE.

uint8_t **ConfigRngCore**

Choose the RNG core.

uint8_t **ConfigHealthTests**

Enable or disable health tests.

uint8_t **rfu**

Padding.

uint32_t **ConfigPRNG**

Choose the number of bits required to have 256 bits of entropy (256 to 640)

PRNG_NIST_CTX **NistCTX**

PRNG NIST context.

The NIST working buffer is structured as 3 working buffers:

- a key
- a value V
- a counter

The exact structure is given by the PRNG_NIST_CTX.

struct **PRNG_NIST_CTX**

Structure of PRNG NIST context

Public Members

uint32_t **Key**[4]

PRNG context key value.

uint32_t **V**[4]

PRNG context internal state.

uint32_t **ResetCounter**

PRNG's number of uses since last reseeding or reset.

5.4.2 APIs overview

Basically:

- The function **SetPRNGCtx** allows to set the options of the context.
- The function **InitPRNG** aims at initializing the NIST context.
- The function **GetPRNG** exploits the context and generates strong random numbers.

5.4.3 Options

5.4.3.1 Choice of the hardware core

The choice of the hardware core is performed with PRNG_Core_t enumeration:

enum **PRNG_Core_t**

Selection of the Hardware RNG core

Values:

enumerator **HW_DEFAULT_RNG_CORE**

RNG default core is core 2. After characterization, core 2 shows better random quality.

enumerator **CORE_0**

Core 0.

enumerator **CORE_1**

Core 1.

enumerator **CORE_2**

Core 2.

enumerator **CORE_3**

Core 3.

5.4.3.2 Enabling or disabling the health tests

The enabling or disabling of the health tests is performed with the PRNG_Health_t enumeration:

enum **PRNG_Health_t**

Health tests performed on the hardware RNG sequence

Values:

enumerator **DISABLE_HEALTH_TESTS**

Health tests disabled.

enumerator **ENABLE_HEALTH_TESTS**

Health tests enabled.

enumerator **HEALTH_TESTS_DEFAULT_CONFIG**

Health tests default configuration.

5.4.3.3 Choice of the underlying AES

The choice of the underlying AES is performed with PRNG_AES_t:

enum **PRNG_AES_t**

Underlying AES for the post processing.

Values:

enumerator **AES_PRNG_HARDWARE**

AES Hardware.

enumerator **AES_PRNG_SOFTWARE**

AES Software.

enumerator **HW_PRNG_DEFAULT_AES_CONFIG**

Default AES is the AES hardware.

5.4.4 Error status

The status of the functions is of type PRNG_Lib_error_t:

enum **PRNG_Lib_error_t**

Error status for PRNG lib

Values:

enumerator **SW_PRNG_OK**

AES computation successful.

enumerator **SW_PRNG_HEALTH_REPETITIVE_ERROR**

Repetitive health test failed.

enumerator **SW_PRNG_HEALTH_ADAPTIVE_ERROR**

Adaptive health test failed.

enumerator **SW_PRNG_NOK**

PRNG initialization not OK.

enumerator **SW_PRNG_INCORRECT_RESULT_POINTER**

Result pointer is not initialized.

enumerator **SW_PRNG_NOT_INITIALIZED**

PRNG was not initialized. Execute InitPRNG first.

5.4.5 SetPRNGCtx

5.4.5.1 Goal of the function

The function sets the context options. This function only sets the parameters of the context. It does not generate any random number.

5.4.5.2 Function

PRNG_Lib_error_t **SetPRNGCtx**(*PRNG_CTX* *Ctx, *PRNG_AES_t* AESSelect, *PRNG_Core_t* CoreSelect, uint32_t PRNGConfig, *PRNG_Health_t* HealthTests)

Set the configuration parameters of a PRNG context.

Note: SetPRNGCtx only defines the parameters. It does not pick any random hardware number and does not initialize the NIST SP800-90A context.

Parameters

- **Ctx** – **[out]** Context of the PRNG
- **AESSelect** – **[in]** Select the underlying AES (AES_HARDWARE or AES_SOFTWARE)
- **CoreSelect** – **[in]** Select the underlying hardware RNG core (from 0 to 3)
- **PRNGConfig** – **[in]** Choose the number of bits required to have 256 bits of entropy (256 to 640)
- **HealthTests** – **[in]** Determine if the health tests will be enabled or not

Return values **SW_PRNG_OK** – PRNG setting successful

Returns Error status

5.4.5.3 Parameters

The context can be configured with 4 options:

1. **ConfigRngCore:** The RNG hardware core can be selected. EM9305 embeds four different RNG cores: Core0, Core1, Core2 and Core3. After characterization of EM9305, Core2 appears to be the best option. The core should be selected among the options defined here: *Choice of the hardware core*
2. **ConfigAES:** The underlying AES can be selected:

- Either the AES-128 Software is selected
- Or the AES-128 Hardware is selected

For best performances, AES hardware will be preferably chosen. Nonetheless, one may prefer the AES software to reduce the peak power consumption. Options are defined here *Choice of the underlying AES*

3. **ConfigHealthTests:** The health tests can be enabled or disabled. To be compliant with NIST SP800-90B, health tests shall be performed. Options are defined here: *Enabling or disabling the health tests*

Note:

- When the health tests are enabled, 512 bytes hardware random numbers must be picked during the initialization.
- When the health tests are disabled, the number of hardware random bytes to pick depends on the entropy of the hardware random numbers. 32 to 80 bytes are required.

Note: Health tests check that there is no catastrophic default in the hardware generation of the random numbers.

---

4. **ConfigPRNG:** The entropy per bit of the hardware random generator. Depending on the entropy of the hardware generator, one needs to pick more or less hardware random numbers to set up the context.

- If the entropy is perfect (=1), 32 hardware random bytes are sufficient to initialize the context.
- If the entropy is not perfect(<1), more than 32 hardware random bytes are necessary.

ConfigPRNG should be set to the value: 256/(Entropy per bit).

Note: The entropy is determined during the hardware characterization. It is assumed that the entropy is not worst case 0.4. When the health tests are enabled, the hardware random numbers are taken among the 512 bytes generated for the health tests. Depending on the entropy, the number of AESs which are performed during the initialization is more or less important. **We recommend to use the default value HW_DEFAULT_ENTROPY_PER_BIT=0.8**

5.4.5.4 Return values

Type	Description	OK \ NOK
SW_PRNG_OK	Setting was successful	OK

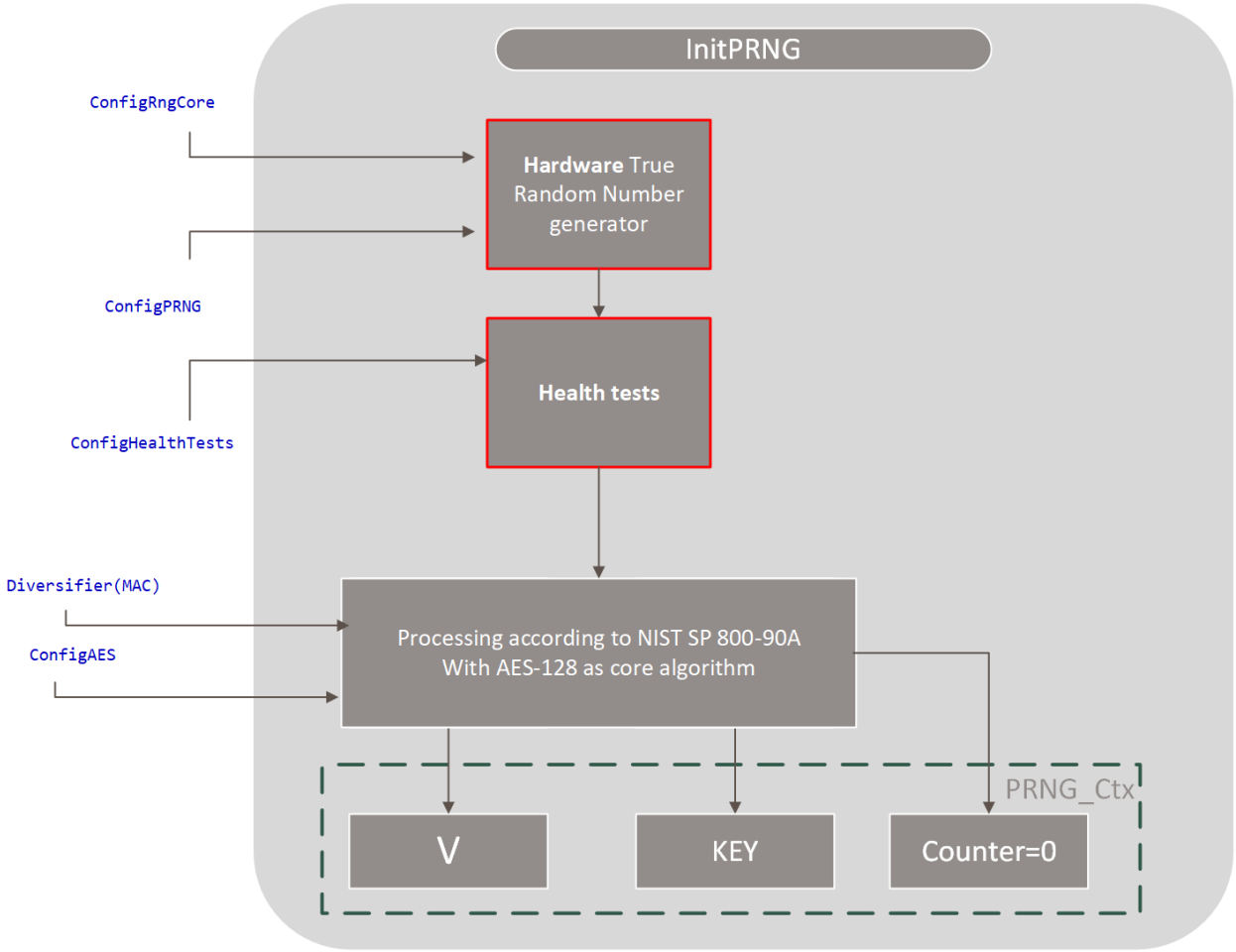
5.4.6 InitPRNG

5.4.6.1 Goal of the function

The function sets the context PRNG_NIST_CTX. It consists in 3 phases:

1. Generation of random values with the hardware random generator.
2. Optionally, test the quality of the hardware random numbers (i.e. Health tests)
3. Post process the hardware random numbers with AES to generate the initial context (V,Key). The counter is also set to null value during this phase.

Next figure illustrates the context initialization phase in function of the parameter.



5.4.6.2 Function

PRNG_Lib_error_t **InitPRNG**(*PRNG_CTX* *Ctx, uint8_t *MAC)

Initializes the PRNG. Pick hardware random numbers to initialize the NIST SP800-90A context. Perform the health tests if required.

Note: The PRNG must be (re)initialized when the PRNG context has been lost.

Parameters

- **Ctx** – **[in]** Context of the PRNG
- **MAC** – **[in]** A specific string of the device. MAC should be 6 bytes long

Return values

- **SW_PRNG_OK** – Successful initialization
- **SW_PRNG_HEALTH_REPETITIVE_ERROR** – Repetitive health test failed
- **SW_PRNG_HEALTH_ADAPTIVE_ERROR** – Adaptive health test failed
- **SW_PRNG_NOK** – Something went wrong with underlying AES

Returns Error status

5.4.6.3 Parameters

- **Ctx** A PRNG context
- **MAC** A 6-byte value. **MAC** is used as a diversifier to ensure that even if the hardware is stuck, two devices would not produce the same random numbers. The BLE MAC address is appropriate for the **MAC** parameter but any other device specific value is also suitable.

5.4.6.4 Return values

Type	Description	OK \ NOK
SW_PRNG_OK	Initialization was successful	OK
SW_PRNG_HEALTH_REPETITIVE_ERROR	Repetitive health test detected an issue. Too many successive hardware random numbers were identical.	NOK
SW_PRNG_HEALTH_ADAPTIVE_ERROR	Adaptive health test detected an issue. The distribution of the hardware random is not correct.	NOK
SW_PRNG_NOK	Something went wrong in initialization.	NOK

Warning: This function must be called at least once.

However, once the context has been initialized, there is no need to call this function anymore until the **next reset**.

After a reset, the context is lost. Therefore, it must be reinitialized.

5.4.6.5 Performances

The execution time of the function InitPRNG depends on different factors. It depends on the parameters and on the chip itself.

For the parameters:

- The performance depends on the entropy parameter. The higher is the entropy, the less hardware random are picked.
- The performance depends whether the health tests are enabled or not. If the health tests are enabled, at least 512 bytes of hardware random numbers are picked, while a maximum of 80 bytes are picked when health tests are disabled.
- In case the health tests fail, the process starts again. As there are 3 trials, in some seldom cases, the process can be three times slower than the typical case.
- The performance depends on the number of AES that are performs. The number of AES depends on the entropy parameter.
- The performance is obviously better when the AES hardware is used instead of the AES software.
- The performance depends on the selected hardware core. Core 0 is the quickest core. Core 3 is the slowest core. Core 1 is relatively quick and Core 2 is average.

In addition to the parameters, **the generation of a random hardware byte is not constant**. This is inherent to the RNG architecture. Actually, the entropy is extracted from the time the core needs to generate a byte. So the highest is standard deviation of the time from one pick to the other, the best is the entropy of the core. Typically, for a given chip, each core follows a Normal distribution.

Nonetheless, as the standard deviation is high, **a timeout** is implemented to limit the total time of InitPRNG process.

Finally, some chips are slower than the other ones. If it makes almost no difference for Core0, which is anyway very quick, the difference of timing for Core2 can be significant..

Next table shows the performance for each core to pick one hardware random byte.

Core	Min in us	Mean in us	Max in us
Core0	0.67	0.67	23
Core1	0.67	1.24	23
Core2	0.67	13	23
Core3	23	23	23

We can notice that:

- Core0 is always fast.
- Core3 is always slow.
- 23us corresponds to the timeout value.
- Core 2 is the core with biggest standard deviation.

Next table shows the number of hardware random numbers that are picked and the number of AESs executed for each configuration.

ID	Health tests enabled	Entropy of the HW RNG	AES	Number of Hardware random numbers to pick	Number of AES to perform
1	NO	1	HW	32	2
2	NO	0.8	HW	40	14
3	NO	0.4	HW	80	32
4	NO	1	SW	32	2
5	NO	0.8	SW	40	14
6	NO	0.4	SW	80	32
7	YES	1	HW	512	2
8	**YES**	**0.8**	**HW**	**512**	**14**
9	YES	0.4	HW	512	32
10	YES	1	SW	512	2
11	YES	0.8	SW	512	14
12	YES	0.4	SW	512	32

The **configuration 8** is the preferred configuration.

For the next time estimates, we consider that:

- each health test needs 125us
- each AES HW needs 30us
- each AES SW needs 250us

Putting all together for core 0 (with 1 health tests iteration):

ID	Health tests enabled	Entropy of the HW RNG	AES	Min time in us	Mean time in us	Max time in us
1	NO	1	HW	82	82	796
2	NO	0.8	HW	447	447	1340
3	NO	0.4	HW	1014	1014	2800
4	NO	1	SW	522	522	1236
5	NO	0.8	SW	3527	3527	4420
6	NO	0.4	SW	8054	8054	9840
7	YES	1	HW	529	529	11961
8	**YES**	**0.8**	**HW**	**889**	**889**	**12321**
9	YES	0.4	HW	1429	1429	12861
10	YES	1	SW	969	969	12401
11	YES	0.8	SW	3669	3669	15401
12	YES	0.4	SW	8469	8469	19901

Putting all together for core 1 (with 1 health tests iteration):

ID	Health tests enabled	Entropy of the HW RNG	AES	Min time in us	Mean time in us	Max time in us
1	NO	1	HW	82	100	796
2	NO	0.8	HW	447	470	1340
3	NO	0.4	HW	1014	1060	2800
4	NO	1	SW	522	540	1236
5	NO	0.8	SW	3527	3550	4420
6	NO	0.4	SW	8054	8100	9840
7	YES	1	HW	529	820	11961
8	**YES**	**0.8**	**HW**	**889**	**1180**	**12321**
9	YES	0.4	HW	1429	1720	12861
10	YES	1	SW	969	1269	12401
11	YES	0.8	SW	3669	4260	15401
12	YES	0.4	SW	8469	8760	19901

Putting all together for core 2 (with 1 health tests iteration):

ID	Health tests enabled	Entropy of the HW RNG	AES	Min time in us	Mean time in us	Max time in us
1	NO	1	HW	82	476	796
2	NO	0.8	HW	447	940	1340
3	NO	0.4	HW	1014	2000	2800
4	NO	1	SW	522	916	1236
5	NO	0.8	SW	3527	4020	4420
6	NO	0.4	SW	8054	9040	9840
7	YES	1	HW	529	6841	11961
8	**YES**	**0.8**	**HW**	**889**	**7201**	**12321**
9	YES	0.4	HW	1429	7741	12861
10	YES	1	SW	969	7281	12401
11	YES	0.8	SW	3669	10281	15401
12	YES	0.4	SW	8469	14781	19901

Putting all together for core 3 (with 1 health tests iteration):

ID	Health tests enabled	Entropy of the HW RNG	AES	Min time in us	Mean time in us	Max time in us
1	NO	1	HW	796	796	796
2	NO	0.8	HW	1340	1340	1340
3	NO	0.4	HW	2800	2800	2800
4	NO	1	SW	1236	1236	1236
5	NO	0.8	SW	4420	4420	4420
6	NO	0.4	SW	9840	9840	9840
7	YES	1	HW	11961	11961	11961
8	**YES**	**0.8**	**HW**	**12321**	**12321**	**12321**
9	YES	0.4	HW	12861	12861	12861
10	YES	1	SW	12401	12401	12401
11	YES	0.8	SW	15401	15401	15401
12	YES	0.4	SW	19901	19901	19901

If we look at the preferred configurations (Core 2, AES hardware, Health tests enabled and entropy equal to 0.8), we have the following times for the case where the health tests pass successfully the first, second time or if they fail.

Number of Health test iterations	Min time in us	Mean time in us	Max time in us
1	889	7201	12321
2	1778	14402	24642
3	2267	21603	**36963**

Note:

- The max time for the **preferred configuration** time is around **37ms** when the 2 first iterations of health tests fail. Nonetheless, it is reminded that the probability to fail is low, so most executions will be bound with 12.3 ms. The process will very occasionally take 25ms.
- Obviously, the absolute max time is for the configuration 12 with AES software and when the health tests fail 3 times. In that case we can reach a max time of 60ms. Nonetheless, this is not a realistic case as the real entropy is definitely higher than 0.4, AES hardware is recommended and the health tests would fail very scarcely.

5.4.7 GetPRNG

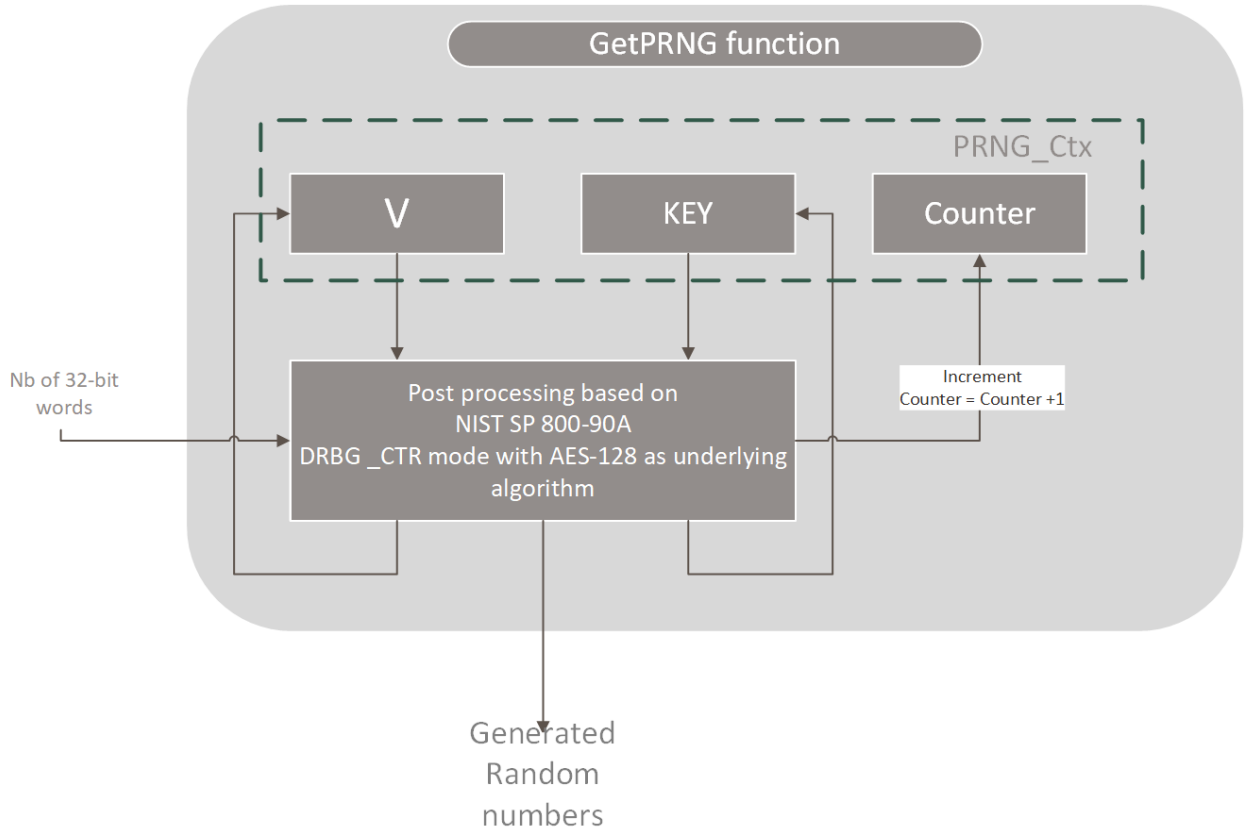
5.4.7.1 Goal of the function

This function generates random values. Internally, it modifies the context according to the specification NIST SP 800-90 A and returns random values issued from a post-treatment on the PRNG context.

This function can be called consecutively 0xFFFFFFFF times without resetting the context.

This function does not pick any random values coming from the true hardware random generator. It is a purely deterministic operation.

The process is illustrated in next figure:



5.4.7.2 Function

PRNG_Lib_error_t **GetPRNG**(*PRNG_CTX* *Ctx, uint32_t NumberOfWords, uint32_t *Random)

Generates random values with the PRNG.

Note: The PRNG must be (re)initialized when the PRNG context has been lost.

Parameters

- **Ctx** – **[in]** PRNG Ctx
- **NumberOfWords** – **[in]** Number of 32-bit words to generate.
- **Random** – **[out]** A buffer of NumberOfWords x 32 bits that receives the random values.

Return values

- **SW_PRNG_OK** – Successful generation
- **SW_PRNG_INCORRECT_RESULT_POINTER** – Result buffer not initialized.
- **SW_PRNG_NOT_INITIALIZED** – The PRNG was not initialized. Execute InitPRNG first
- **SW_PRNG_NOK** – Something went wrong with underlying AE

Returns Error status

5.4.7.3 Parameters

- **Ctx**: A context PRNG
- **NumberOfWords**: Number of desired 32-bit long random words.
- **Random**: Pointer on the result buffer.

5.4.7.4 Remark

Note: Internally, the post treatment generates 16 bytes of random numbers per loop iteration. It results that the optimal **NumberOfWords** required is a multiple of 4. For instance, it would be more efficient to call GetPRNG with NumberOfWords=4, rather than 4 times GetPRNG with NumberOfWords=1.

Also there is a penalty of 2 AES executions, at each call of the GetPRNG function. Therefore it would be more efficient to require several words in one time, rather than calling the function multiple times for the same amount of random numbers.

5.4.7.5 Return values

Type	Description	OK \ NOK
SW_PRNG_OK	Initialization was successful	OK
SW_PRNG_INCORRECT_RESULT_POINTER	Null pointer.	NOK
SW_PRNG_NOK	Something went wrong in the post-processing.	NOK

5.4.7.6 Performances

The performances of GetPRNG depend on the underlying AES and the number of desired words.

Number of 32-bit words	Number of bits	Number of AESs performed	Time in us with AES HW	Time in us with AES SW
2	64	3	86	750
4	128	3	86	750
8	256	4	115	1000
12	384	5	143	1250
16	512	6	172	1500
20	640	7	201	1750
32	1024	10	287	2500

5.5 General Performances

5.5.1 Library location

The lib is located in ROM.

5.5.2 Code size

Size in bytes
2172 bytes in ROM

5.5.3 RAM

Size in bytes
44 bytes per context

5.5.4 Stack

Size in bytes
TBD

5.5.5 Dependencies

PRNG depends on the AES library.

5.6 Important remark

Warning: PRNG_CTX shall be located in persistent RAM. Otherwise, the PRNG context will be lost whenever the part goes to sleep mode.

If the context is in persistent RAM, it is enough to invoke InitPRNG only after a reset.

5.7 Example

Next code provides a basic example of use.

```
//-------------------------------------------------------------------------------------
//
// FILE : Example_PRNG.c
//
// Goal : Basic example of PRNG use
//-------------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
#include <stdint.h>
#include "AES.h"
#include "PRNG.h"

//=============================================================================
//                                                  Example
//=============================================================================

uint8_t Example_PRNG(void) {
        uint32_t Error = 0;
        PRNG_Lib_error_t SW;
        uint32_t Random1[2];
        uint32_t Random2[2];

        //Create a PRNG context- WARNING it should be located in persistent RAM
        PRNG_CTX Ctx;

        //Choose the options for this PRNG context
        SetPRNGCtx(&Ctx, HW_PRNG_DEFAULT_AES_CONFIG, HW_DEFAULT_RNG_CORE,
        HW_PRNG_DEFAULT_CONFIG, HEALTH_TESTS_DEFAULT_CONFIG);

        //Alternatively, we could set the context directly without the help of␣
↪setPRNGCtx function
        //Ctx.ConfigRngCore = HW_DEFAULT_RNG_CORE;
        //Ctx.ConfigPRNG = HW_PRNG_DEFAULT_CONFIG;
        //Ctx.ConfigAES = HW_PRNG_DEFAULT_AES_CONFIG;
        //Ctx.ConfigHealthTests = HEALTH_TESTS_DEFAULT_CONFIG;

        //Test MAC address
        static const uint8_t MAC1[6] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 };

        //The PRNG MUST to be initialized once per session, i.e . when the context is␣
↪lost (RAM erased, reset...)
        SW = InitPRNG(&Ctx, (uint8_t*) MAC1);
        if (SW != SW_PRNG_OK)
                Error++;

        //Let's generate some random data
        SW = GetPRNG(&Ctx, 2, Random1);
        if (SW != SW_PRNG_OK)
                Error++;

        SW = GetPRNG(&Ctx, 2, Random2);
        if (SW != SW_PRNG_OK)
                Error++;

        //let's check if they are different
        if ((Random1[0]==Random2[0]) &&(Random1[1] == Random2[1]))
                Error++;

        return (Error);
```

(continues on next page)

(continued from previous page)

```
}
```

CHAPTER

SIX

ECC FUNCTIONS

6.1 ECC P-256

6.1.1 Introduction

The underlying curve used for all the ECC-based functions is **P-256**. This curve is also referred as **secp256r1**.

P-256 is the curve used for BLE pairing.

The curve is fully defined in SEC2 standard.

[1] SEC2: Recommended Elliptic Curve Domain Parameters

SEC2: https://www.secg.org/SEC2-Ver-1.0.pdf

6.1.2 Curve Parameters

The curve has the following parameters.

The prime factor *p* is:

```
p= FFFFFFFF 00000001 00000000 00000000 00000000 FFFFFFFF FFFFFFFF FFFFFFFF
```

i.e:

$p=2^{224} (2^{32}-1) + 2^{192} +2^{96} -1$

The curve parameters *a* and *b* are:

```
a=FFFFFFFF 00000001 00000000 00000000 00000000 FFFFFFFF FFFFFFFF FFFFFFFC
```

that is to say `a=-3 mod p.`

```
b =5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E 27D2604B
```

The generator *point G* is:

```
G = 6B17D1F2 E12C4247 F8BCE6E5 63A440F2 77037D81 2DEB33A0 F4A13945 D898C296,
     4FE342E2 FE1A7F9B 8EE7EB4A 7C0F9E16 2BCE3357 6B315ECE CBB64068 37BF51F5
```

The *order of G* is

```
n = FFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC632551
```

The *cofactor* is 1. It means that the group formed by the points on the curve has no subgroup.

6.1.3 Data size

A **scalar** (for instance a private key) on P-256 is 256-bit long.

A **point** (for instance a public key) on P-256 is composed of two coordinates *x* and *y*. Each coordinate is 256-bit long.

6.2 ECC Error Enumeration

The list of error status of the APIs are given by the enumeration ECC_Error_t.

enum **ECC_Error_t**

Error status for ECC.

Values:

enumerator **SW_ECC_OK**

Operation succeeded.

enumerator **SW_ECDSA_SIGNATURE_VALID**

ECDSA signature is valid.

enumerator **SW_SCHNOOR_SIGNATURE_VALID**

Schnoor signature is valid.

enumerator **SW_ECDSA_INVALID_SIGNATURE**

The ECDSA signature is not valid.

enumerator **SW_SCHNOOR_SIGNATURE_INVALID**

The Schnoor signature is not valid.

enumerator **SW_ECC_INVALID_PUBLIC_POINT**

Public point is either not on the curve or is the infinity point.

enumerator **SW_ECC_NULL_PRIVATE_KEY**

The private key is null. Computation failed.

enumerator **SW_ECC_REDUCED_PRIVATE_KEY**

The private key is bigger than N. Computation was performed but the private key was reduced modulo N.

enumerator **SW_ECC_SIGNATURE_COORDINATE_X_INVALID**

The coordinate X(r) of the signature is not valid (either bigger than N or null)

enumerator **SW_ECC_SIGNATURE_COORDINATE_Y_INVALID**

The coordinate Y(s) of the signature is not valid (either bigger than N or null)

enumerator **SW_ECC_INVALID_COMMITMENT**

ECC Schnoor commitment is not valid. It is either not on the curve or is the infinity point.

enumerator **SW_ECC_RANDOM_ERROR**

Error in the random generation.

enumerator **SW_ECC_INCORRECT_RESULT_POINTER**

Non initialized result pointer.

6.3 ECC Key management

6.3.1 Goal of the document

The goal of this document is to describe the **key management functions of the ECC.lib**

This document:

- describes all the supported functions, including the prototypes, the parameters, etc…
- provides the performances of the functions.

6.3.2 ECC Key management utilities

The **ECC.lib** provides 3 functions to manage the ECC keys on P-256.

- a function to **generate a private key**.
- a function to **generate a public key from a private key**.
- a function to **verify if a public key is valid or not**.

6.3.3 APIs

6.3.3.1 Enumerations

The error status are given here: *ECC Error Enumeration*

6.3.3.2 ECC_GeneratePrivateKey

6.3.3.2.1 Goal of the function

This function generates a private key for P-256 curve. The private key is a 32-byte long random value. The key is generated securely with the PRNG.lib.

6.3.3.2.2 Function

ECC_Error_t **ECC_GeneratePrivateKey**(uint32_t *PrivateKey)

Generates an ECC private key.

Parameters **PrivateKey** – **[out]** The generated private key, a scalar of the size of the curve (32 -bytes)

Return values

- **SW_ECC_OK** – Successful private key generation
- **SW_ECC_RANDOM_ERROR** – Error in random generation

Returns Error status

6.3.3.2.3 Parameters

- **PrivateKey** : The generated private key.

6.3.3.2.4 Return values

Type	Description	OK \ NOK
SW_ECC_OK	The private key generation was successful	OK
SW_ECC_RANDOM_ERROR	The private key generation failed. Something wrong happened with the random generation	NOK

6.3.3.3 ECC_ComputePubKey

6.3.3.3.1 Goal of the function

The function generates a public key from the private key.

6.3.3.3.2 Function

ECC_Error_t **ECC_ComputePubKey**(uint32_t *PrivateKey, POINT *PubKey)

Computes the ECC public key associated to a private key.

Parameters

- **PrivateKey** – **[in]** ECC private key (a scalar)
- **PubKey** – **[out]** ECC generated public key (a point in affine coordinates)

Return values

- **SW_ECC_OK** – No error occurred
- **SW_ECC_REDUCED_PRIVATE_KEY** – Computation was performed but the private key was reduced modulo N
- **SW_ECC_NULL_PRIVATE_KEY** – The private key is null. No computation performed.

Returns Error status

6.3.3.3.3 Parameters

- **PrivateKey** : A 32-byte private key.
- **PubKey**: The public key corresponding to the given private key (2*32 bytes).

6.3.3.3.4 Return values

Type	Description	OK \ NOK
SW_ECC_OK	The public key is successfully generated	OK
SW_ECC_REDUCED_PRIVATE_KEY	The public key was generated but as the private key was greater than the modulus N, the private key was reduced modulo N	OK
SW_ECC_NULL_PRIVATE_KEY	The private key is null modulo N. The public key is not generated	NOK

6.3.3.4 ECC_isPublicKeyValid

6.3.3.4.1 Goal of the function

The function verifies if a public key is valid. It checks if the point is on the curve or if the point is the infinity point.

6.3.3.4.2 Function

ECC_Error_t **ECC_isPublicKeyValid**(POINT *PubKey)

Verify if the public key is on the curve.

Parameters **PubKey** – **[in]** Public key to test

Return values

- **SW_ECC_OK** – Point is valid
- **SW_ECC_INVALID_PUBLIC_POINT** – Invalid public point

Returns Error status

6.3.3.4.3 Parameters

- **PubKey**: The public key to check.

6.3.3.4.4 Return values

Type	Description	OK \ NOK
SW_ECC_OK	The public key is valid	OK
SW_ECC_INVALID_PUBLIC_POINT	The public key is not valid	NOK

6.3.4 General Performances

6.3.4.1 Library location

The lib is located in ROM. The key management functions are included in the **ECC.lib.**

6.3.4.2 Code size

Size in bytes
5284 bytes in ROM for the complete ECC lib

6.3.4.3 RAM

Size in bytes
No global RAM

6.3.4.4 Stack

Size in bytes
Approximately 784 bytes

6.3.4.5 Performances

Function	Nb of cycles	Time in ms at 48Mhz
ECC_GeneratePrivateKey	See Remark	0.2 \ 1.5
ECC_ComputePubKey	5108137	106
ECC_isPublicKeyValid	5389	0.12

Note: The time of the private key generation depends on the PRNG setting of cryptographic random numbers generation. It also depends on the choice of the underlying AES (AES hardware or software) for the PRNG. Finally, if the PRNG was never set up since reset, execution time will be higher to allow the seeding of the PRNG. Note that

by default the PRNG uses the AES hardware. 0.2 us corresponds to a PRNG already set up with the AES hardware. Similarly 1.5ms corresponds to the AES software.

6.3.4.6 Dependencies

The ECC lib depends on :

- PRNG.lib for the random number generation
- AES.lib which is used by PRNG.lib

6.3.5 Example

Next code shows basic examples of:

- private key generation
- public key computation
- public key verification

```
////////////////////////////////////////////////////////////////////////////////
///
/// File:       ExampleECCKeyManagement.c
///
/// Goal:                   Example of use of ECC key functions
///
////////////////////////////////////////////////////////////////////////////////
#include <stdint.h>
#include "ECCTypedef.h"
#include "ECCStatus.h"
#include "ECCkeyManagement.h"
#include "ECCVerifyPubKey.h"

//==============================================================================
//                                                  Example
//==============================================================================

uint8_t Example_ECC_KeyManagement(void) {
        ECC_Error_t sw;
        uint8_t i;
        uint32_t Error = 0;
        POINT_P256 PubKey;
        uint32_t Private[CURVE_SIZE_P256];

        static const uint32_t Private_F[CURVE_SIZE_P256] = { 0x3f49f6d4, 0xa3c55f38,
                        0x74c9b3e3, 0xd2103f50, 0x4aff607b, 0xeb40b799, 0x5899b8a6,
                        0xcd3c1abd };
        static const POINT_P256 ExpectedPublic = { { 0x20b003d2, 0xf297be2c,
                        0x5e2c83a7, 0xe9f9a5b9, 0xeff49111, 0xacf4fddb, 0xcc030148,
                        0x0e359de6 }, { 0xdc809c49, 0x652aeb6d, 0x63329abf, 0x5a52155c,
                                        0x766345c2, 0x8fed3024, 0x741c8ed0, 0x1589d28b }␣
↪};
```

(continues on next page)

(continued from previous page)

```
        //Example to generate a private key
        //---------------------------------
        //A 256 bit(32 bytes) random fills "Private"
        sw = ECC_GeneratePrivateKey(Private);
        if (sw != SW_ECC_OK)
                Error++;

        //Compute a public key from the private key
        //-----------------------------------------
        //for this example , we take a fix private key, so that we can compare
        //the computed public key
        sw = ECC_ComputePubKey((uint32_t*) Private_F, (POINT*) &PubKey);
        if (sw != SW_ECC_OK)
                Error++;
        //check the result is as expected
        for (i = 0; i < CURVE_SIZE_P256; i++) {
                if (PubKey.x[i] != ExpectedPublic.x[i])
                        Error++;
                if (PubKey.y[i] != ExpectedPublic.y[i])
                        Error++;
        }

        //Check the public key is valid
        //-----------------------------
        sw = ECC_isPublicKeyValid((POINT*) &PubKey);
        if (sw != SW_ECC_OK)
                Error++;
        //Check that a invalid key is rejected
        //------------------------------------
        // for our example we corrupt the public key
        PubKey.x[0] = 0x00000000;
        //now check that the key is rejected
        sw = ECC_isPublicKeyValid((POINT*) &PubKey);
        if (sw != SW_ECC_INVALID_PUBLIC_POINT)
                Error++;

        //return the test status
        if (Error)
                return (1);
        else
                return (0);
}
```

6.4 ECDH - Diffie Hellman key agreement

6.4.1 Bibliography

[1] NIST 800-56A: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography

NIST 800-56A: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf

6.4.2 Goal of the document

The goal of this document is to describe the ECC Diffie Hellman function of the **ECC.lib**.

This document:

- describes the supported functions,
- describes the parameters, etc…
- provides the performances of the functions

6.4.3 Reminder on the ECDH protocol

ECDH is a key agreement protocol based on elliptic curves. In the context of EM9305, we use ECDH for the devices pairing. Each device owns a pair of keys, composed of a private key and a public key.

Each device sends its public key to the other device.

Then, they perform an ECC computation, in order to generate on both side a common secret key SK. After the execution of ECDH protocol, the secret key SK is the same on both sides.

Bluetooth specification mandates the use of the curves **P-256**. It results that the ECDH protocol is performed on P-256 curve. For details on **P-256** see *ECC P-256*.

Next figure illustrates ECDH protocol.

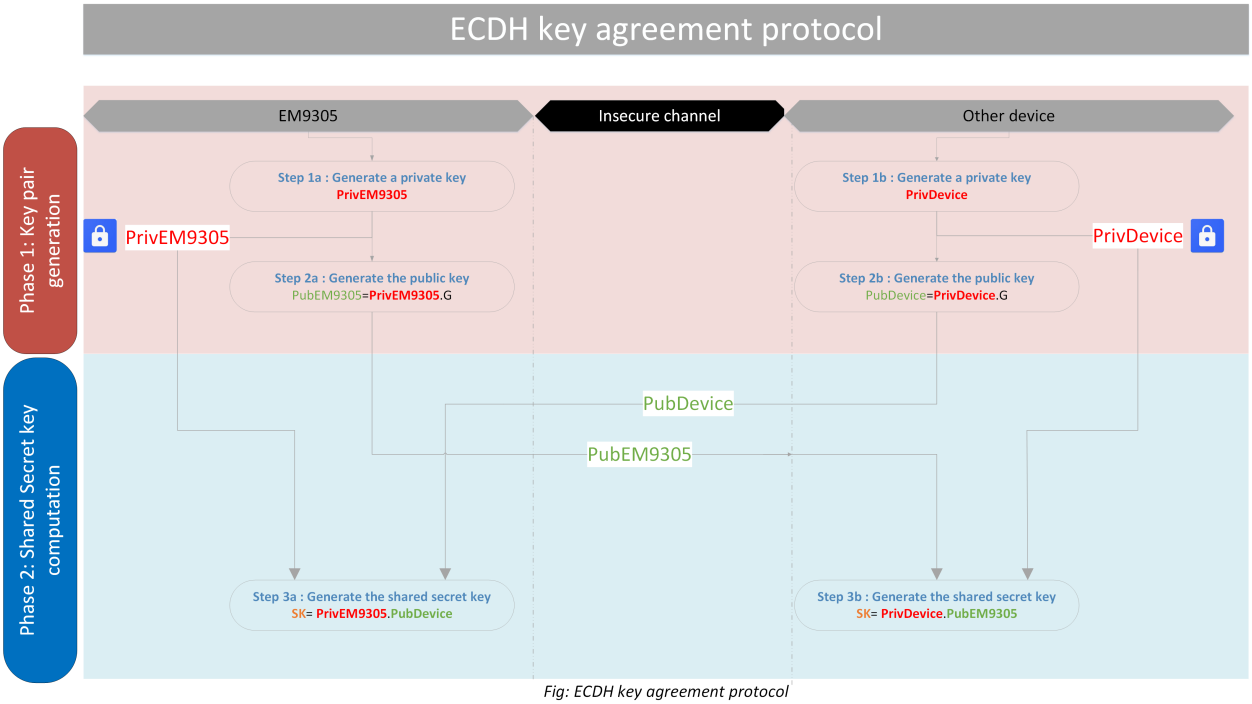


Fig: ECDH key agreement protocol
The figure illustrates the two phases for two devices to agree on a shared secret key SK.
First phase consists in generating on each side a key pair(private and public key)
In the second phase, each entity sends its public key to the other entity and
derivate from the recieved public key and its own private key, the common key SK.

The protocol is performed in 2 phases:

- **Phase 1: the key pairs generation** (in brown in the figure)
  - Each party generates a private key. Each party derives its public key from the private key.
  - Each party stores its key pair. The private key is stored securely.
  - This phase can be performed only once.
- **Phase 2: the secret key agreement** (in blue in the figure)
  - Each party sends its public key to the other party.
  - Each party generates the shared key SK from the received public key and its own private key.

6.4.4 APIs

The library includes an API that generates the shared secret key. For the phase 1, *ECC Key management* page describes the functions to generate a key pair.

6.4.4.1 Enumerations

The error status are given here: *ECC Error Enumeration*.

6.4.4.2 ECC_ComputeSharedKey

6.4.4.2.1 Goal of the function

The function generates a shared key SK from a public key and a private key.

6.4.4.2.2 Function

ECC_Error_t **ECC_ComputeSharedKey**(POINT *PubKeyA, uint32_t *PrivateKeyB, uint32_t *SK)

Computes the shared key between 2 devices.

Parameters

- **PubKeyA** – **[in]** Public key of device A. It is an affine point received from device A.
- **PrivateKeyB** – **[in]** Private key of device B. It is a scalar.
- **SK** – **[out]** The shared key, i.e a scalar of the size of the curve.

Return values

- **SW_ECC_OK** – Successful computation
- **SW_ECC_INVALID_PUBLIC_POINT** – Invalid public point
- **SW_ECC_INCORRECT_RESULT_POINTER** – Invalid result buffer(SK)

Returns Error status

6.4.4.2.3 Parameters

- **PubKey** : Received public key (an ECC point)
- **SecretKey** : a 256-bit private key
- **SK**: the generated shared key (256 bits)

6.4.4.2.4 Return values

Type	Description	OK \ NOK
SW_ECC_OK	The shared key was successfully generated	OK
SW_ECC_INVALID_PUBLIC_POINT	The public key is not on the curve	NOK

6.4.5 General Performances

6.4.5.1 Library location

The lib is located in ROM. ECDH is included in the **ECC.lib**.

6.4.5.2 Code size

Size in bytes
5284 bytes in ROM for the complete ECC lib

6.4.5.3 RAM

Size in bytes
No global RAM

6.4.5.4 Stack

Size in bytes
Approximately 944 bytes

6.4.5.5 Performances

Function	Number of cycles	Time in ms at 48Mhz
ECC_ComputeSharedKey	5235170	109

6.4.5.6 Dependencies

The ECC lib depends on :

- PRNG.lib for the random number generation
- AES.lib which is used by PRNG.lib

For security reasons, some operations are performed with random numbers. It explains the dependency with the RNG.

6.4.6 Example

Next C code example shows how to compute the shared key on the two communicating devices.

```
////////////////////////////////////////////////////////////////////////////////
///
/// FILE       ExampleECDH.c
///
/// GOAL           Example of DH computation
///
////////////////////////////////////////////////////////////////////////////////
#include <stdint.h>
#include "ECCTypedef.h"
#include "ECCStatus.h"
#include "ECDH.h"

//============================================================================
//                                                  Example
//============================================================================

uint8_t Example_ECDH(void) {
        ECC_Error_t SW;
        uint8_t i;
        uint32_t error = 0;
        uint32_t SK[CURVE_SIZE_P256];

        //we assume that entity A has the following key pair
        static const uint32_t PrivateA[CURVE_SIZE_P256] = { 0x3f49f6d4, 0xa3c55f38,
                        0x74c9b3e3, 0xd2103f50, 0x4aff607b, 0xeb40b799, 0x5899b8a6,
                        0xcd3c1abd };
        static const POINT_P256 PubA = { { 0x20b003d2, 0xf297be2c, 0x5e2c83a7,
                        0xe9f9a5b9, 0xeff49111, 0xacf4fddb, 0xcc030148, 0x0e359de6 }, {
                                        0xdc809c49, 0x652aeb6d, 0x63329abf, 0x5a52155c,␣
↪0x766345c2,
                                        0x8fed3024, 0x741c8ed0, 0x1589d28b } };

        //we assume that entity B has the following key pair
        static const uint32_t PrivateB[CURVE_SIZE_P256] = { 0x55188b3d, 0x32f6bb9a,
                        0x900afcfb, 0xeed4e72a, 0x59cb9ac2, 0xf19d7cfb, 0x6b4fdd49,
                        0xf47fc5fd };

        static const POINT_P256 PubB = { { 0x1ea1f0f0, 0x1faf1d96, 0x09592284,
                        0xf19e4c00, 0x47b58afd, 0x8615a69f, 0x559077b2, 0x2faaa190 }, {
                                        0x4c55f33e, 0x429dad37, 0x7356703a, 0x9ab85160,␣
↪0x472d1130,
                                        0xe28e3676, 0x5f89aff9, 0x15b1214a } };

        //DH protocol between the entity A and B should be as follows:
        static const uint32_t ExpectedDHKey[CURVE_SIZE_P256] = { 0xec0234a3,
                        0x57c8ad05, 0x341010a6, 0x0a397d9b, 0x99796b13, 0xb4f866f1,
                        0x868d34f3, 0x73bfa698 };

        //Function ECC_ComputeSharedKey:
```

(continues on next page)

(continued from previous page)

```
        //compute secret key between two entities A and B
        //-----------------------------------------------
        //check the DH key with PubA and PrivB
        //(computation performed by entity B)
        SW = ECC_ComputeSharedKey((POINT*) &PubA, (uint32_t*) PrivateB, SK);
        if (SW != SW_ECC_OK)
                error++;
        for (i = 0; i < CURVE_SIZE_P256; i++) {
                if (SK[i] != ExpectedDHKey[i])
                        error++;
        }

        //check the DH key with PubB and PrivA
        //(computation performed by entity A)
        SW = ECC_ComputeSharedKey((POINT*) &PubB, (uint32_t*) PrivateA, SK);
        if (SW != SW_ECC_OK)
                error++;
        for (i = 0; i < CURVE_SIZE_P256; i++) {
                if (SK[i] != ExpectedDHKey[i])
                        error++;
        }

        //return the test status
        if (error)
                return (1);
        else
                return (0);
}
```

6.5 ECDSA

6.5.1 Bibliography

[1] FIPS 186-4: Digital Signature Standard

FIPS 186-4: https://csrc.nist.gov/publications/detail/fips/186/4/final

6.5.2 Goal of the document

The goal of this document is to describe the functionality of the library **ECDSA**.

This document:

- describes the supported functions,
- describes the parameters, etc…
- provides the performances of the functions.

6.5.3 ECDSA protocol

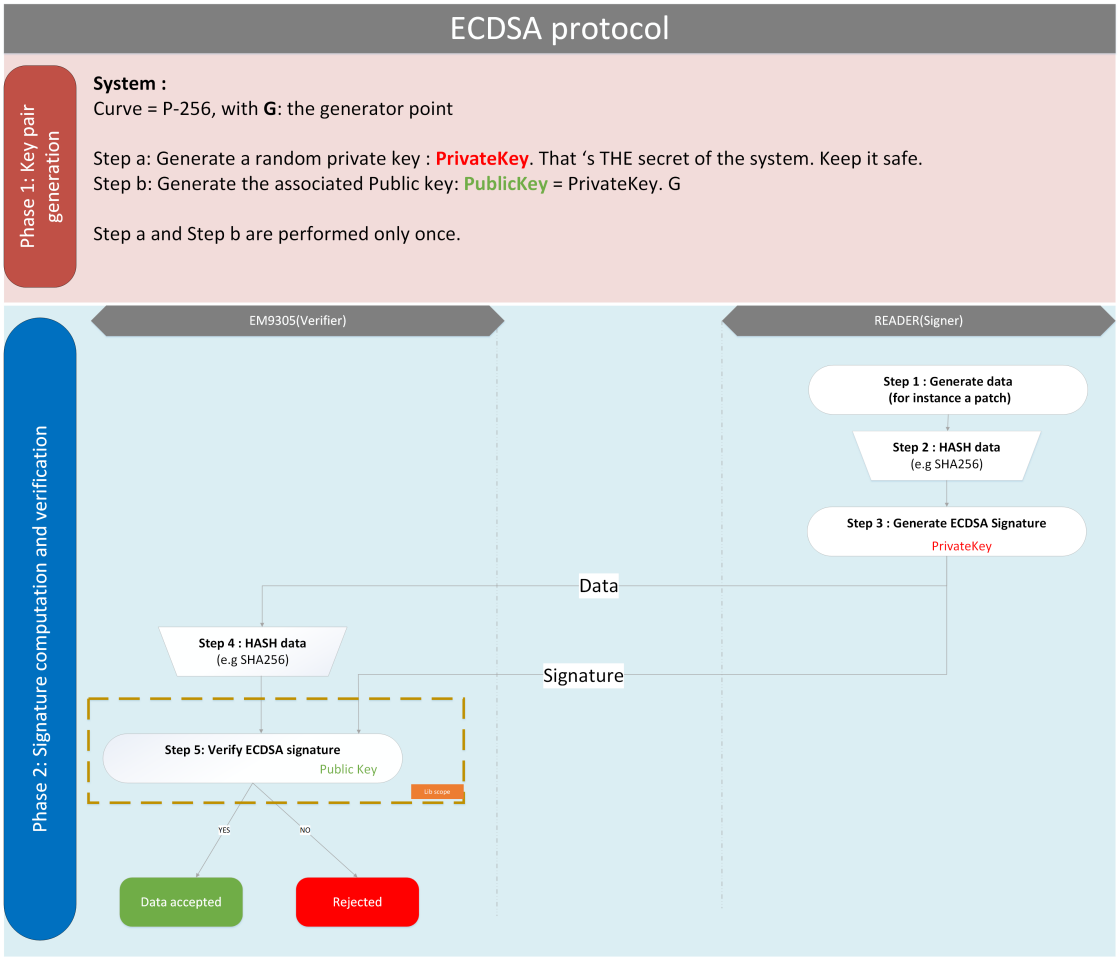
ECDSA protocol is a signature protocol based on elliptic curves.

In the context of EM9305, ECDSA is used to sign data and ensure the authenticity and the origin of the data.

External entity signs the data with the private key, while EM9305 verifies the signature with the public key.

If the signature is valid, the data is approved. If the signature is not valid, the data is rejected.

Next figure illustrates the protocol.



ECDSA relies on:

- **an underlying curve**: the underlying curve is **P-256**.
- **a hash algorithm**: the API user can use any hash function (SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 or any other). Nonetheless, it is recommended to use SHA-256 as the level of security is consistent with P-256 and a 32-byte long digest would not require truncation or padding.

For further details on **P-256**, see the chapter *ECC P-256*. For further details on the hash functions,:ref:*SHA1_Label*, *SHA224*, *SHA256*, *SHA384*, *SHA512*

The figure illustrates the 2 phases of the ECDSA signature process:

- **Phase 1: the key pairs generation** (in brown in the figure)

  A key pair of the system is generated:

  - Step a: a private key is generated.
  - Step b: the public key is generated from the private key.

  The public key is provided to the signature verifier. In our example, the key is stored in EM9305.

  The private key is provided to the signer.

- **Phase 2: The signature generation and the signature verification** (in blue in the figure)

  It is composed of several steps:

  - Step 1: the signer generates data.
  - Step 2: the signer hashes the previous data.
  - Step 3: the signer signs the hash of the data using the system private key. The signer sends the data and the signature to the verifier.
  - Step 4: the verifier hashes the received data.
  - Step 5: the verifier verifies the consistency of the signature and the hash.

6.5.4 APIs

The library only includes an API to **verify the signature**.

Note: The library does not embed any function to generate a signature.

6.5.4.1 Enumerations

The error status are given here: *ECC Error Enumeration*

6.5.4.2 ECDSA_Verify

6.5.4.2.1 Goal of the function

The function performs an ECDSA verification.

Note: The hash computation is not performed in this API. It only performs the operations on the P-256 curve.

6.5.4.2.2 Function

ECC_Error_t **ECDSA_Verify**(PointCertificate *Signature, POINT *PubKey, uint8_t *Message)

Performs an ECDSA verification.

Parameters

- **Signature** – **[in]** Signature of the message (a point)
- **PubKey** – **[in]** Public key of the system (a point)
- **Message** – **[in]** Message that was signed (32 -bytes)

Return values

- **SW_ECDSA_SIGNATURE_VALID** – Signature is valid
- **SW_ECDSA_INVALID_SIGNATURE** – Signature is invalid
- **SW_ECC_INVALID_PUBLIC_POINT** – Invalid public point
- **SW_ECC_SIGNATURE_COORDINATE_X_INVALID** – Invalid coordinate X
- **SW_ECC_SIGNATURE_COORDINATE_Y_INVALID** – Invalid coordinate Y

Returns Error status

6.5.4.2.3 Parameters

- **Signature** : Signature of the message (a point)
- **PubKey** : The public key of the system. A point that shall be on the curve P-256.
- **Message** : The message that was signed. The message shall be 32-byte long. This is typically the result of the hash function.

6.5.4.2.4 Return values

Type	Description	OK \ NOK
SW_ECDSA_SIGNATURE_VALID	Verification operation is successful and the signature is valid	OK
SW_ECDSA_INVALID_SIGNATURE	Verification operation was performed but the signature is not valid	NOK
SW_ECC_INVALID_PUBLIC_POINT	The public key is not on the curve	NOK
SW_ECC_SIGNATURE_COORDINATE_X_INVALID	The coordinate X of the given signature is not valid. Either it is null or greater than the modulus	NOK
SW_ECC_SIGNATURE_COORDINATE_Y_INVALID	The coordinate Y of the given signature is not valid. Either it is null or greater than the modulus	NOK

6.5.5 General Performances

6.5.5.1 Library location

The lib is located in ROM.

6.5.5.2 Code size

Size in bytes
5284 bytes in ROM for the complete ECC lib

6.5.5.3 RAM

Size in bytes
No global RAM

6.5.5.4 Stack

Size in bytes
Approximately 864 bytes

6.5.5.5 Performances

Function	Number of cycles	Time in ms at 48Mhz
ECDSA_Verify	6378725	132

6.5.5.6 Dependencies

The ECC lib depends on :

- PRNG.lib for the random number generation
- AES.lib which is used by PRNG.lib

For security reasons, some operations are performed with random numbers. It explains the dependency with the RNG.

6.5.6 Example

Next C code example shows how to verify a signature and the returned status word when:

- the signature is valid
- the signature is not valid

```
/*-------------------------------------------------
 FILE : ExampleECDSAVerify.c
 -------------------------------------------------*/

#include <stdint.h>
#include "ECCTypedef.h"
#include "ECCStatus.h"
#include "ECDSA_Verify.h"
#include <ExampleECCUtilities.h>
```

(continues on next page)

(continued from previous page)

```
//test structure
typedef struct {
    char * Message;
        char * Kpubx;
    char * Kpuby;
    char * SignX;
    char * SignY;
} TEST_ECDSA_P256;


//First example: the signature is correct
TEST_ECDSA_P256 ExampleOK=
{
                "44acf6b7e36c1342c2c5897204fe09504e1e2efb1a900377dbc4e7a6a133ec56",
                "1ccbe91c075fc7f4f033bfa248db8fccd3565de94bbfb12f3c59ff46c271bf83",
                "ce4014c68811f9a21a1fdb2c0e6113e06db7ca93b7404e78dc7ccd5ca89a4ca9",
                "f3ac8061b514795b8843e3d6629527ed2afd6b1f6a555a7acabb5e6f79c8c2ac",
                "8bf77819ca05a6b2786c76262bf7371cef97b218e96f175a3ccdda2acc058903"
};

//Second example: the signature is NOT consistent with the message
TEST_ECDSA_P256 ExampleNOK=
{
                "01acf6b7e36c1342c2c5897204fe09504e1e2efb1a900377dbc4e7a6a133ec56",
                "1ccbe91c075fc7f4f033bfa248db8fccd3565de94bbfb12f3c59ff46c271bf83",
                "ce4014c68811f9a21a1fdb2c0e6113e06db7ca93b7404e78dc7ccd5ca89a4ca9",
                "f3ac8061b514795b8843e3d6629527ed2afd6b1f6a555a7acabb5e6f79c8c2ac",
                "8bf77819ca05a6b2786c76262bf7371cef97b218e96f175a3ccdda2acc058903"
};



uint8_t Example_ECDSAVerify(void)
{

        uint8_t Message[CURVE_SIZE_P256*4];  //Message is 32 bytes for P256
        PointCertificate Signature;
        POINT_P256 Kpub;
        ECC_Error_t Status;
        uint8_t Error=0;


        //EXAMPLE 1: Signature is correct
        //------------------------------
        //convert the data in a byte array
        ConvertStringToU8(ExampleOK.Message,Message,CURVE_SIZE_P256*4);

        //Create the public key in a point
        ConvertStringToU32(ExampleOK.Kpubx, &Kpub.x[0],CURVE_SIZE_P256);
        ConvertStringToU32(ExampleOK.Kpuby, &Kpub.y[0],CURVE_SIZE_P256);
```

(continues on next page)

(continued from previous page)

```
        //Create the signature
        ConvertStringToU32(ExampleOK.SignX, &Signature.x[0],CURVE_SIZE_P256);
        ConvertStringToU32(ExampleOK.SignY, &Signature.y[0],CURVE_SIZE_P256);

        //Go for verification
        Status= ECDSA_Verify((PointCertificate*)&Signature,(POINT*)&Kpub,(uint8_
↪t*)Message);
        //We expect a SW OK since the signature is correct
        if(Status!=SW_ECDSA_SIGNATURE_VALID)
                Error++;

        //EXAMPLE 2: Signature is NOT correct
        //-----------------------------
        //convert the data in a byte array
        ConvertStringToU8(ExampleNOK.Message,Message,CURVE_SIZE_P256*4);

        //Create the public key in a point
        ConvertStringToU32(ExampleNOK.Kpubx, &Kpub.x[0],CURVE_SIZE_P256);
        ConvertStringToU32(ExampleNOK.Kpuby, &Kpub.y[0],CURVE_SIZE_P256);

        //Create the signature
        ConvertStringToU32(ExampleNOK.SignX, &Signature.x[0],CURVE_SIZE_P256);
        ConvertStringToU32(ExampleNOK.SignY, &Signature.y[0],CURVE_SIZE_P256);

        //Go for verification
        Status= ECDSA_Verify((PointCertificate*)&Signature,(POINT*)&Kpub,(uint8_
↪t*)Message);
        //We expect a SW NOK since the signature is NOT correct
        if(Status!=SW_ECDSA_INVALID_SIGNATURE)
                Error++;

        return(Error);
}
```

6.6 ECC Schnorr's Authentication protocol

6.6.1 Bibliography

[1] RFC 8235: Schnorr Non-interactive Zero-Knowledge Proof

RFC 8235: https://tools.ietf.org/html/rfc8235#section-3

6.6.2 Goal of the document

The goal of this document is to describe the functionality of the ECC function **Schnorr**.

This chapter:

- introduce the authentication protocol Schnorr.
- describes all the APIs supported by EM9305 including * the prototypes of the functions * the parameters * the error status
- provides the performances of the functions.

6.6.3 Schnorr's Authentication protocol

6.6.3.1 Protocol description

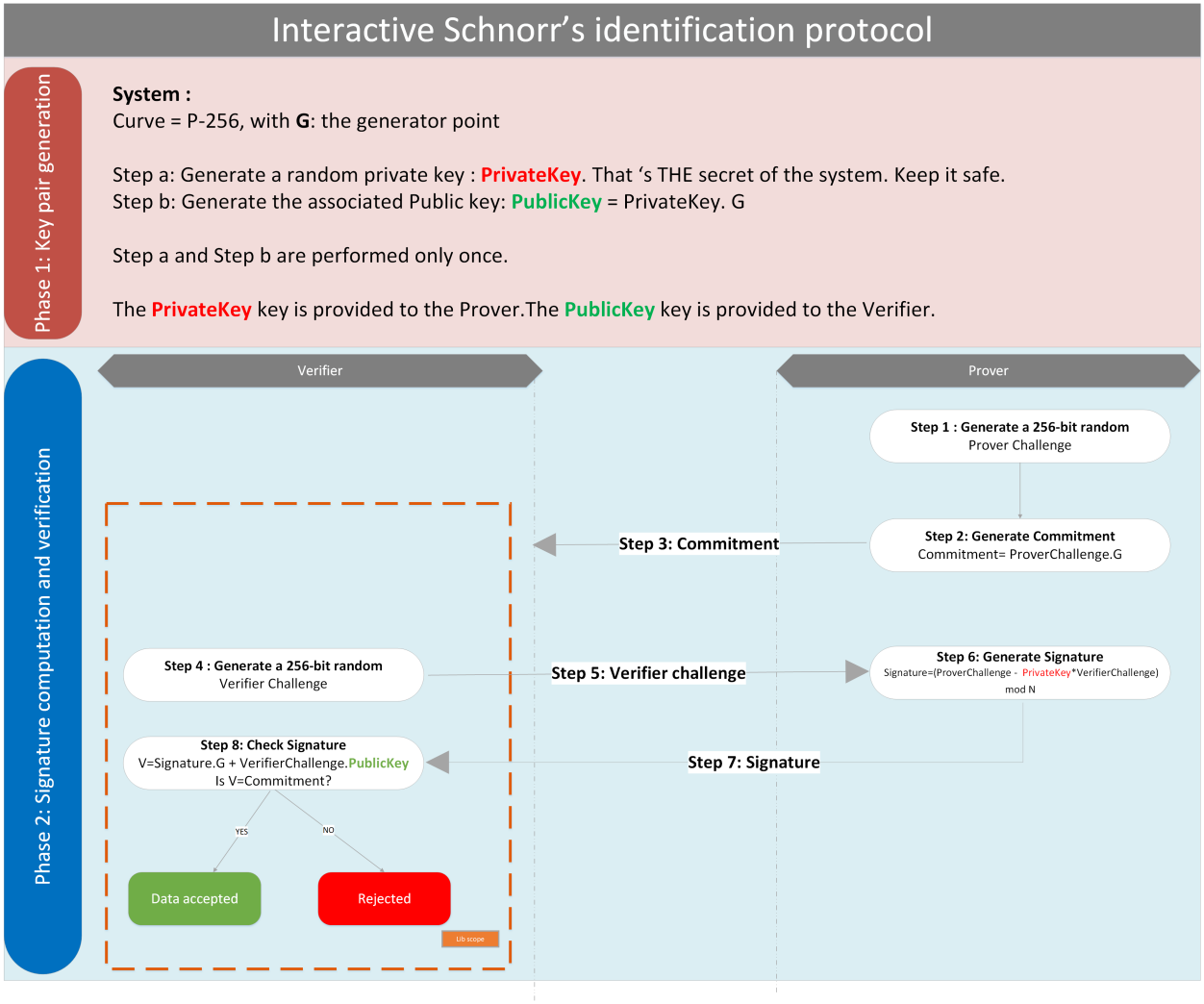
Schnorr's protocol is an identification protocol based on public key primitives. It aims **at authenticating unilaterally** a party to another party.

Note: In the context of EM9305, Schnorr's protocol is used to enter in configuration mode. The device EM9305 authenticates the external reader/tester. Therefore, the external reader is the **"Prover"** and the part EM9305 is the **"Verifier"**.

Schnorr's protocol is based on the *discret logarithm problem*. The library implements Schnorr's protocol based on ECC. As BLE products already embeds P-256, the Schnorr's protocol is also based on the underlying curve P-256. The curve details are described here: *ECC P-256*.

The implemented protocol is an **interactive Schnorr's identification protocol** and is partially described in the document [1].

Next picture illustrates the protocol.



The protocol is performed in 2 phases.

A. Phase 1: Generation of the system key set

   a. An ECC private key (*PrivateKey*) is generated.

   b. The public key (*PublicKey*) is computed from the private key. *PublicKey=PrivateKey.G*

The Private key is distributed to the **prover**, while the public key is provided to the **verifier**.

Note: The key pair (PrivateKey-PublicKey) composes the keys of the system. This operation is performed only once.

B. Phase 2: Authentication

When the **prover** wants to authenticate to the **verifier**, the following steps are performed:

1. The prover generates a random challenge: *ProverChallenge*
2. The prover computes the commitment corresponding to the prover challenge: *Commitment= ProverChallenge.G*
3. The prover sends the *commitment* to the verifier
4. The verifier generates a challenge:*VerifierChallenge*

5. The verifier sends the *VerifierChallenge* to the prover
6. The prover generates the signature: *Signature=(ProverChallenge - PrivateKey*VerifierChallenge) mod N*
7. The prover sends the *Signature* to the verifier
8. The verifier verifies the signature. It computes *V=Signature.G + VerifierChallenge.PublicKey* and compares V with the *Commitment*. If *V=Commitment*, the prover is successfully authenticated. Otherwise, the prover is rejected.

Note: The operator "." in step 2 and 8 is an ECC point multiplication. The "+" ,the "-" and the "*" of steps 6 and 8 are modular operation on scalars.

6.6.3.2 Why does it work?

Developing the formulae and replacing each element we show that:

V = Signature.G + VerifierChallenge.PublicKey

*V = [(ProverChallenge - PrivateKey*VerifierChallenge)]. G + VerifierChallenge .(PrivateKey.G)*

*V = (ProverChallenge. G) - [(PrivateKey *VerifierChallenge)mod N.G] +[VerifierChallenge *PrivateKey)modN.G]*

V = (ProverChallenge. G)

V = Commitment

Therefore, when the prover has knowledge of the PrivateKey , the V value computed by the verifier matches the recieved commitment.

6.6.4 APIs

The library includes useful APIs for the **verifier** that is to say:

- a function that generates the verifier challenge
- a function that verifies the prover signature

Note: The embedded library does not include functions executed by the prover.

6.6.4.1 Enumerations

The error status are given here: *ECC Error Enumeration*

6.6.4.2 ECCSchnoor_GenerateVerifierChallenge

6.6.4.2.1 Goal of the function

The function generates a 256-bit random challenge.

6.6.4.2.2 Function

ECC_Error_t **ECCSchnoor_GenerateVerifierChallenge**(uint32_t *VerifierChallenge)

Generate a random challenge.

Parameters **VerifierChallenge** – **[out]** a random challenge of the size of the curve (32 bytes)

Return values

- **SW_ECC_OK** – Successful private key generation
- **SW_ECC_RANDOM_ERROR** – Error in random generation

Returns Error status

6.6.4.2.3 Parameters

- **VerifierChallenge** : Result buffer of 256-bit(32 bytes)

6.6.4.2.4 Return values

Type	Description	OK \ NOK
SW_ECC_OK	Verifier random challenge successfully generated	OK
SW_ECC_RANDOM_ERROR	Error in the random generation	NOK

6.6.4.3 ECCSchnoor_Verify

6.6.4.3.1 Goal of the function

The function verifies the consistency of the signature and the commitment received from the prover.

6.6.4.3.2 Function

ECC_Error_t **ECCSchnoor_Verify**(POINT *PublicKey, POINT *Commitment, uint32_t *VerifierChallenge, uint32_t *Signature)

Verify the external signature.

Parameters

- **PublicKey** – **[in]** The public key of the device. A point on P-256
- **Commitment** – **[in]** The commitment sent by the external entity. A point on p-256

- **VerifierChallenge** – **[in]** The 32-byte challenge generated by the device
- **Signature** – **[in]** The 32-byte signature sent by the external entity

Return values

- **SW_SCHNOOR_SIGNATURE_VALID** – Signature is valid
- **SW_SCHNOOR_SIGNATURE_INVALID** – Signature is not valid
- **SW_ECC_INVALID_COMMITMENT** – Invalid commitment
- **SW_ECC_INVALID_PUBLIC_POINT** – Invalid public key

Returns Error status- Validiy of the signature

6.6.4.3.3 Parameters

- **PublicKey** : Public key of the system
- **Commitment** : The commitment received from the prover
- **VerifierChallenge** : The challenge generated during the ECCSchnorr_GenerateVerifierChallenge command
- **Signature** : The signature submitted by the prover

6.6.4.3.4 Return values

Type	Description	OK \ NOK
SW_SCHNOOR_SIGNATURE_VALID	Signature is valid. The prover is authenticated	OK
SW_SCHNOOR_SIGNATURE_INVALID	Signature is invalid. The prover is rejected	NOK
SW_ECC_INVALID_COMMITMENT	The commitment is invalid	NOK
SW_ECC_INVALID_PUBLIC_POINT	The public key is invalid	NOK

6.6.5 General Performances

6.6.5.1 Library location

The lib is located in ROM. Schnorr's authentication protocol is included in the **ECC.lib**.

6.6.5.2 Code size

Size in bytes
5284 bytes in ROM for the complete ECC lib

6.6.5.3 RAM

Size in bytes
No global RAM

6.6.5.4 Stack

Size in bytes
Approximately 600 bytes

6.6.5.5 Performances

Function	Time in ms at 48Mhz
ECCSchnorr_GenerateVerifierChallenge	0.287
ECCSchnorr_Verify	127

6.6.5.6 Dependencies

The ECC lib depends on :

- PRNG.lib for the random number generation
- AES.lib which is used by PRNG.lib

6.6.6 Example

Next C code shows a basic example of the Schnorr's protocol APIs in the context of an external reader authenticating to EM9305.

```
////////////////////////////////////////////////////////////////////////////////
///
/// @file       ExampleECCSchnoorUserManual.c
///
/// @project    T9305
///
/// @author     SAS
///
/// @brief          Example of use of Schnoor lib functions
/// @classification  Confidential
///
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
///
////////////////////////////////////////////////////////////////////////////////
///
/// @copyright Copyright (C) 2020 EM Microelectronic
/// @cond
///
```

(continues on next page)

(continued from previous page)

```
/// All rights reserved.
///
/// Redistribution and use in source and binary forms, with or without
/// modification, are permitted provided that the following conditions are met:
/// 1. Redistributions of source code must retain the above copyright notice,
/// this list of conditions and the following disclaimer.
/// 2. Redistributions in binary form must reproduce the above copyright notice,
/// this list of conditions and the following disclaimer in the documentation
/// and/or other materials provided with the distribution.
///
////////////////////////////////////////////////////////////////////////////////
///
/// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
/// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
/// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
/// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
/// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
/// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
/// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
/// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
/// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
/// POSSIBILITY OF SUCH DAMAGE.
/// @endcond
////////////////////////////////////////////////////////////////////////////////
#include <stdint.h>
#include "ECCTypedef.h"
#include "ECCStatus.h"
#include "ECCSchnoor.h"

//Basic example of Schnoor functions with fix test value
//This is the most basic and thus quickest test.
uint8_t Example_ECC_Schnoor_UM(void) {
        ECC_Error_t SW;
        uint32_t u32Error = 0;

        //Supposing that the system keys are :

        //1- the private key of the system
        /*static const uint32_t PrivateKey[CURVE_SIZE_P256] =
         * { 0x3f49f6d4, 0xa3c55f38, 0x74c9b3e3,0xd2103f50, 0x4aff607b, 0xeb40b799,␣
↪0x5899b8a6, 0xcd3c1abd };*/

        //2- It correspond to the public key(normaly stored in the device T9305)
        static const POINT_P256 PubKey = { { 0x20b003d2, 0xf297be2c, 0x5e2c83a7,
                        0xe9f9a5b9, 0xeff49111, 0xacf4fddb, 0xcc030148, 0x0e359de6 }, {
                        0xdc809c49, 0x652aeb6d, 0x63329abf, 0x5a52155c, 0x766345c2,
                        0x8fed3024, 0x741c8ed0, 0x1589d28b } };

        //Supposing that the prover(Reader) generates the following challenge
        //static uint32_t ProverChallenge[CURVE_SIZE_P256]=  {0x12345678,0x9ABCDEF0,
↪0xAABBCCDD,0xEEFF0011,0x12345678,0x9ABCDEF0,0xAABBCCDD,0xEEFF0011};
```

(continues on next page)

(continued from previous page)

```
        // and the the Verifier(Device) generates this challenge
        static uint32_t VerifierChallenge[CURVE_SIZE_P256] = { 0xABCDEF12,
                        0xFEDCBA12, 0x12121212, 0x34343434, 0x56565656, 0x78787878,
                        0x9A9A9A9A, 0x1B1B1B1B };

        //then the Prover(reader) should generate the following commitment
        static const POINT_P256 Commitment = { { 0xbf4a7502, 0x11c5a2f2, 0x17c557e1,
                        0x9c1b43a3, 0xd5547099, 0x23181f94, 0x91521db7, 0x5ceb8b16 }, {
                        0xfebcceb1, 0x66124ba2, 0x3bae58c6, 0xe7d4152c, 0x4609b2f7,
                        0x1bcc47e9, 0x4b1e95e0, 0x325a9805 } };

        //and the following signature
        static const uint32_t Signature[CURVE_SIZE_P256] = { 0x14ef47ab, 0x128a3627,
                        0x73df4636, 0x7ecc1d15, 0x64b5a898, 0xba454da7, 0xdb47b279,
                        0x2c439c92 };

        //Perform the Verification of the signature by the verifier (Device)
        SW = ECCSchnoor_Verify((POINT*) &PubKey, (POINT*) &Commitment,
                        VerifierChallenge, (uint32_t*) Signature);
        if (SW != SW_SCHNOOR_SIGNATURE_VALID)
                u32Error++;

        //return the test status
        if (u32Error)
                return (1);
        else
                return (0);
}
```

CHAPTER

SEVEN

HASH FUNCTIONS

7.1 SHA-1

7.1.1 Bibliography

[1] FIPS 180-4: Secure Hash Standard

FIPS 180-4: https://csrc.nist.gov/publications/detail/fips/180/4/final

7.1.2 Goal of the document

The goal of this document is to describe the functionality of the library **SHA1**. It describes:

- the supported functions,
- the parameters of the functions,
- the error cases,
- the performances of the functions.

This library implements SHA-1 algorithm according to the specification FIPS 180-4 [1].

7.1.3 Types of APIs

It implements two sets of APIs:

- A first API allows to perform SHA-1 in one shot: the message and its size are provided and the function computes the digest in one shot.
- A second set of APIs allows to provide the data to hash block by block. The computation of the digest can be performed step by step, accumulating data until the final processing.

7.1.4 APIs

7.1.4.1 Enumerations

enum **SHA1_Lib_error_t**

Error status words for SHA1.

Values:

enumerator **SHA1_SUCCESS**

SHA1 computation successful.

enumerator **SHA1_INCORRECT_RESULT_POINTER**

Result pointer is null.

7.1.4.2 Defines

SHA1_DIGEST_SIZE

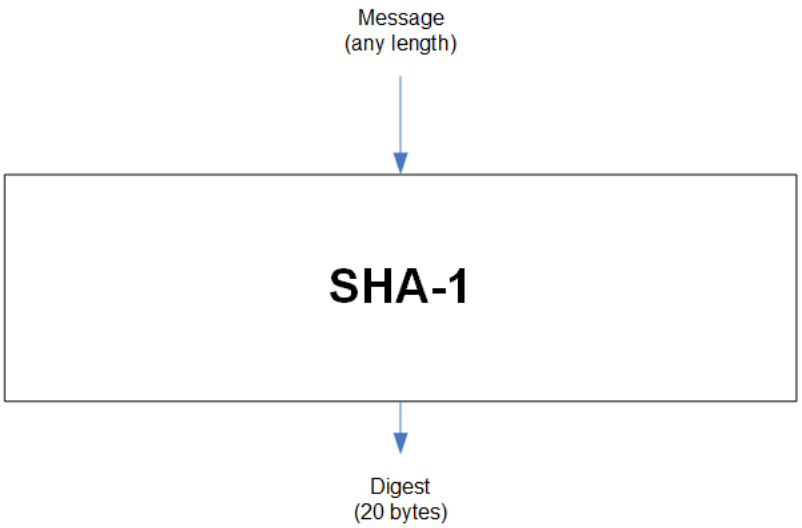
Digest size for SHA1 in bytes

SHA1_BLOCK_SIZE

Block size for SHA1 in bytes

7.1.4.3 ONE SHOT COMPUTATION FUNCTION : SHA1 API

This function computes the SHA1 digest of a data of any length. The computation is performed in one-shot. Next figure shows the general scheme.



7.1.4.3.1 Function

SHA1_Lib_error_t **SHA1**(uint8_t *Message, uint8_t *Result, uint32_t MessageLength)

Computes the hash of the message with SHA1.

Parameters

- **Message** – **[in]** Message to hash
- **Result** – **[out]** Digest of the message
- **MessageLength** – **[in]** Message length in bytes

Return values

- **SHA1_SUCCESS** – Successful computation
- **SHA1_INCORRECT_RESULT_POINTER** – Result buffer not initialized

Returns Error status

7.1.4.3.2 Parameters

- **Message** : The message to hash. It can be of any length.
- **Result**: Pointer on the 20-byte result.
- **MessageLength** : Length of the message in bytes. It can be any value from 0 (included) to 0xFFFFFFFF (included).

7.1.4.3.3 Return values

Type	Description	OK \ NOK
SHA1_SUCCESS	SHA1 computation successful.	OK
SHA1_INCORRECT_RESULT_POINTER	Result pointer is null.	NOK

7.1.4.4 STEP BY STEP FUNCTIONS

7.1.4.4.1 Goal of the functions

According to [1] SHA1 algorithm is performed in several steps.

- The first step consists in initializing a context of 20 bytes with known constants.
- The second step consists in splitting the data to hash into blocks of 64 bytes. Each block is processed to update the context.
- Finally the data to hash is padded with a fix pattern and the data length. The context is updated accordingly. The digest consists in the last value of the 20-byte context.

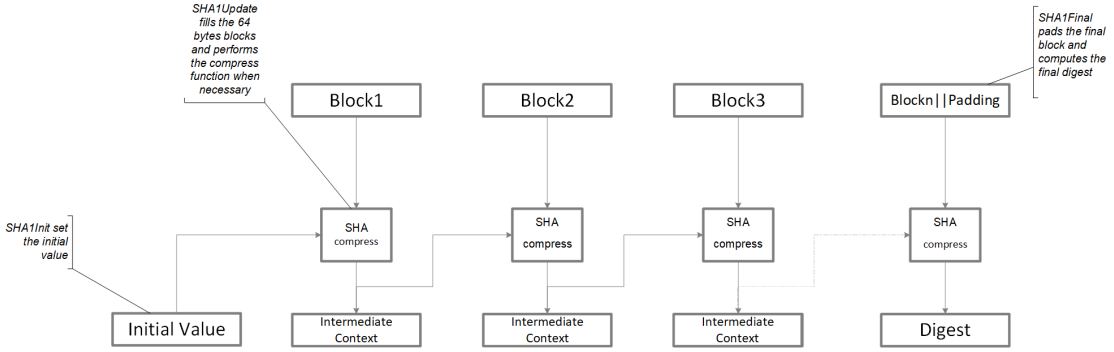
The step by step functions allow to perform SHA-1 in several steps. Especially it permits to provide the data by block of any size.

The set of functions is constituted of 3 functions:

- An initialization function that initializes the context.

- An update function that accumulates the input data and processes the data as soon as 64 bytes are accumulated.
- A final function that performs the padding and computes the final digest.

Next image shows the principle of the step by step functions:



7.1.4.4.2 Context

SHA1_CTX is a structure that accumulates the intermediate data necessary to the hash computation. It is defined as follows in SHA1_CTX.

struct **SHA1_CTX**

SHA1 context

Public Members

uint32_t **Hash**[SHA1_DIGEST_SIZE / 4]

Digest buffer

uint8_t **Block**[SHA1_BLOCK_SIZE]

Working buffer to accumulate data to hash

uint32_t **MessageLen**

Size in bytes of the message accumulated till now

uint8_t **Index**

Size of the data accumulated in the Block buffer

7.1.4.4.3 SHA1Init

Function

This function initializes a SHA1 computation. It initializes the SHA1_CTX.

SHA1_Lib_error_t **SHA1Init**(*SHA1_CTX* *ctx)

Initializes a SHA1 computation.

Parameters **ctx** – **[out]** SHA-1 context

Return values `SHA1_SUCCESS` – Successful initialization

Returns Error status

7.1.4.4.4 Parameters

- **ctx** : a SHA1 context

7.1.4.4.5 Return values

Type	Description	OK \ NOK
SHA1_SUCCESS	SHA1 successfully initialized.	OK

7.1.4.4.6 SHA1Update

Function

This function processes the data provided by pbInput buffer. It accumulates the data in the context until the block is full (64 bytes). If the block is full, the data is hashed and the 20-byte long context is updated. The inputLength can be any size from 0x00(included) to 0xFFFFFFFF(included).One will usually call SHA1Update several times. All the bytes by of the message must be processed SHA1Update, including the potential incomplete block. On the other hand, the padding should not be provided to SHA1Update.

SHA1_Lib_error_t **SHA1Update**(*SHA1_CTX* *ctx, uint8_t *pbInput, uint32_t InputLength)

Updates the context with the data given in input.

Parameters

- `ctx` – **[inout]** SHA-1 context
- `pbInput` – **[in]** Data to hash
- `InputLength` – **[inout]** Length of the data pointed by pbInput (in bytes)

Return values `SHA1_SUCCESS` – Successful computation

Returns Error status

7.1.4.4.7 Parameters

- **ctx** : a SHA1 context that was initialized previously.
- **pbInput** : data to hash of length InputLength
- **InputLength** : length in bytes of the data (from 0x00 to 0xFFFFFFFF)

7.1.4.4.8 Return values

Type	Description	OK \ NOK
SHA1_SUCCESS	SHA1 update successful.	OK

7.1.4.4.9 SHA1Final

Function

SHA1Final adds the final padding and processes the context a final time to produce the digest.

SHA1_Lib_error_t **SHA1Final**(*SHA1_CTX* *ctx, uint8_t *pbResult)

Finalizes the hash computation and returns the result.

Parameters

- **ctx** – **[in]** SHA-1 context
- **pbResult** – **[out]** Final digest

Return values

- **SHA1_SUCCESS** – Successful computation
- **SHA1_INCORRECT_RESULT_POINTER** – Result buffer not initialized

Returns Error status

7.1.4.4.10 Parameters

- **ctx** : a SHA1 context that was initialized previously
- **pbResult** : pointer on the 20-byte hash digest

7.1.4.4.11 Return values

Type	Description	OK \ NOK
SHA1_SUCCESS	SHA1 update successful.	OK
SHA1_INCORRECT_RESULT_POINTER	pbResult is null pointer	NOK

7.1.5 Performances

7.1.5.1 Library location

The lib is located in ROM.

7.1.5.2 Code size

Size in bytes
476 bytes in ROM for the SHA1 module
In addition, the generic hash framework is 564 bytes

7.1.5.3 RAM

Size in bytes
No usage of global RAM except the SHA1_CTX

7.1.5.4 Stack

Size in bytes
Approximately 336 bytes

7.1.5.5 Execution time

The execution time of SHA1 naturally depends on the length of the message to hash. The data being processed by blocks of 64 bytes, the time will essentially depend on the number of 64-byte blocks the message has. Nonetheless for small data, one will take care that, due to the padding, an extra block may be added.

The next table shows the number of processed block(s) for various message lengths as example.

Data length in bytes	Number of processed blocks
0	1
1	1
55	1
56	1
57	2
63	2
64	2
128	3
1024	17

The performances of each function are given in the next table.

For the SHA1Update, besides the fact that it depends on the data length, it also depends on the amount of data already accumulated. The process of an extra 64-byte block may be necessary or not.

For the SHA1Final, 1 or 2 blocks must be processed depending on the overall length of the data to digest.

For the SHA1 API, it also depends on the data length. All together, the processing time of one 64-byte block is the most relevant figure.

Next table gives some examples of execution time.

Function	Number of cycles	Time in us at 48Mhz
Process of 1 64-byte block	6018	125
SHA1Init	81	1.6
SHA1Update(for 64 bytes)	6018	125
SHA1Final(for 1 block processed)	5378	112
SHA1(for 112 bytes)	12195	254

7.1.6 Dependencies

SHA1 lib depends on the generic hash framework library **hash.lib**.

7.1.7 Example

```
////////////////////////////////////////////////////////////////////////////////
///
/// @file       ExampleSHA1.c
///
/// @project    T9305
///
/// @author     SAS
///
/// @brief           Example of use of SHA1 library
///
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
///
////////////////////////////////////////////////////////////////////////////////
///
/// @copyright Copyright (C) 2022 EM Microelectronic
/// @cond
///
/// All rights reserved.
///
/// Redistribution and use in source and binary forms, with or without
/// modification, are permitted provided that the following conditions are met:
/// 1. Redistributions of source code must retain the above copyright notice,
/// this list of conditions and the following disclaimer.
/// 2. Redistributions in binary form must reproduce the above copyright notice,
/// this list of conditions and the following disclaimer in the documentation
/// and/or other materials provided with the distribution.
///
////////////////////////////////////////////////////////////////////////////////
///
/// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
/// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
/// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
/// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
/// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
/// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
```

(continues on next page)

(continued from previous page)

```
/// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
/// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
/// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
/// POSSIBILITY OF SUCH DAMAGE.
/// @endcond
////////////////////////////////////////////////////////////////////////////////
#include <stdint.h>
#include "SHA1.h"

/*****************************************************************************
 Function: ExampleSHA1

 Basic example of use of the SHA1 library

 Return:
 0 if everything is ok
 1 otherwise

 Remark 1:
 SHA1 uses the generic hash Engine SHA_ARC.lib. Therefore, to call SHA1 one needs:
 - SHA_ARC.lib
 - SHA1_ARC.lib

 Remark 2:
 Two kinds of APIs are supported:
 * SHA1 API allows to perform a SHA1 hash in one step
 * SHA1Init, SHA1Update, SHA1Final allow to hash a message step by step.

 *****************************************************************************/
uint8_t ExampleSHA1(void) {
        uint8_t Digest[SHA1_DIGEST_SIZE];
        uint8_t i;
        uint8_t Error = 0;
        SHA1_CTX MyCtx;
        SHA1_Lib_error_t sw;

#define MESSAGE_LEN 112

        const uint8_t Message[MESSAGE_LEN] = { 0x61, 0x62, 0x63, 0x64, 0x65, 0x66,
                        0x67, 0x68, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x63,
                        0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x64, 0x65, 0x66, 0x67,
                        0x68, 0x69, 0x6a, 0x6b, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b,
                        0x6c, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x67, 0x68,
                        0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x68, 0x69, 0x6a, 0x6b, 0x6c,
                        0x6d, 0x6e, 0x6f, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70,
                        0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x6b, 0x6c, 0x6d,
                        0x6e, 0x6f, 0x70, 0x71, 0x72, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71,
                        0x72, 0x73, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x6e,
                        0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75 };

        const uint8_t ExpectedResult[SHA1_DIGEST_SIZE] = { 0xa4, 0x9b, 0x24, 0x46,
                        0xa0, 0x2c, 0x64, 0x5b, 0xf4, 0x19, 0xf9, 0x95, 0xb6, 0x70, 0x91,
```

(continues on next page)

(continued from previous page)

```
                        0x25, 0x3a, 0x04, 0xa2, 0x59 };

        //Example 1: we perform the computation in one step
        //-------------------------------------------------
        //perform the computation in one step
        sw = SHA1((uint8_t*) Message, Digest, MESSAGE_LEN);
        if (sw != SHA1_SUCCESS)
                Error++;

        //compare the result
        for (i = 0; i < SHA1_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //Example 2: we perform the computation in several steps
        //------------------------------------------------------
        // Note that in the two following examples, the size of the cuts message is␣
↪purely
        // arbitrary for the sake of examples. Any size, including 0 is allowed.

        //Example 2a. We first perform the update 64 bytes per 64 bytes.
        //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        //init
        sw = SHA1Init(&MyCtx);
        if (sw != SHA1_SUCCESS)
                Error++;
        //accumulate and perform computation of first 64 bytes
        sw = SHA1Update(&MyCtx, (uint8_t*) &Message[0], 64);
        if (sw != SHA1_SUCCESS)
                Error++;
        //continue on the next 112-64 following bytes
        sw = SHA1Update(&MyCtx, (uint8_t*) &Message[64], MESSAGE_LEN - 64);
        if (sw != SHA1_SUCCESS)
                Error++;
        //finalize the computation and get the result
        sw = SHA1Final(&MyCtx, Digest);
        if (sw != SHA1_SUCCESS)
                Error++;
        //compare the result
        for (i = 0; i < SHA1_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //Example 2b. We first perform the update 16 bytes per 16 bytes.
        //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        //init
        sw = SHA1Init(&MyCtx);
        if (sw != SHA1_SUCCESS)
```

(continues on next page)

(continued from previous page)

```
                Error++;
        for (i = 0; i < 7; i++) {
                //accumulate and perform computation on 16 bytes
                sw = SHA1Update(&MyCtx, (uint8_t*) &Message[16 * i], 16);
                if (sw != SHA1_SUCCESS)
                        Error++;
        }
        //finalize the computation and get the result
        sw = SHA1Final(&MyCtx, Digest);
        if (sw != SHA1_SUCCESS)
                Error++;

        //compare the result
        for (i = 0; i < SHA1_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //final status of the test
        if (Error != 0)
                return (1);
        else
                return (0);

}
```

7.2 SHA224

7.2.1 Bibliography

[1] FIPS 180-4: Secure Hash Standard

FIPS 180-4: https://csrc.nist.gov/publications/detail/fips/180/4/final

7.2.2 Goal of the document

The goal of this document is to describe the functionality of the library **SHA224**. It describes:

- the supported functions,
- the parameters of the functions,
- the error cases,
- the performances of the functions.

This library implements SHA224 algorithm according to the specification FIPS 180-4 [1].

7.2.3 Types of APIs

It implements two sets of APIs:

- A first API allows to perform SHA224 in one shot: the message and its size are provided and the function computes the digest in one shot.
- A second set of APIs allows to provide the data to hash block by block. The computation of the digest can be performed step by step, accumulating data until the final processing.

7.2.4 APIs

7.2.4.1 Enumerations

enum **SHA224_Lib_error_t**

Error status words for SHA224.

Values:

enumerator **SHA224_SUCCESS**

SHA224 computation successful.

enumerator **SHA224_INCORRECT_RESULT_POINTER**

Result pointer is null.

7.2.4.2 Defines

SHA224_DIGEST_SIZE

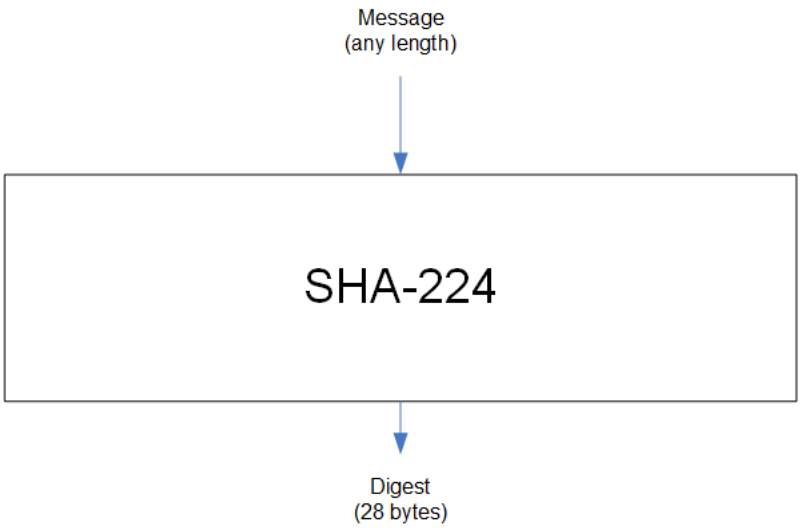
Digest size for SHA224 in bytes

SHA224_BLOCK_SIZE

Block size for SHA224 in bytes

7.2.4.3 ONE SHOT COMPUTATION FUNCTION : SHA224 API

This function computes the SHA224 digest of a data of any length. The computation is performed in one-shot. Next figure shows the general scheme.



7.2.4.3.1 Function

SHA224_Lib_error_t **SHA224**(uint8_t *Message, uint8_t *Result, uint32_t MessageLength)

Computes the hash of the message with SHA224.

Parameters

- **Message** – **[in]** Message to hash
- **Result** – **[out]** Digest of the message
- **MessageLength** – **[in]** Message length in bytes

Return values

- **SHA224_SUCCESS** – Successful computation
- **SHA224_INCORRECT_RESULT_POINTER** – Result buffer not initialized

Returns Error status

7.2.4.3.2 Parameters

- **Message** : The message to hash. It can be of any length.
- **Result**: Pointer on the 28-byte result.
- **MessageLength** : Length of the message in bytes. It can be any value from 0x00 (included) to 0xFFFFFFFF (included).

7.2.4.3.3 Return values

Type	Description	OK \ NOK
SHA224_SUCCESS	SHA224 computation successful.	OK
SHA224_INCORRECT_RESULT_POINTER	Result pointer is null.	NOK

7.2.4.4 STEP BY STEP FUNCTIONS

7.2.4.4.1 Goal of the functions

According to [1] SHA224 algorithm is performed in several steps.

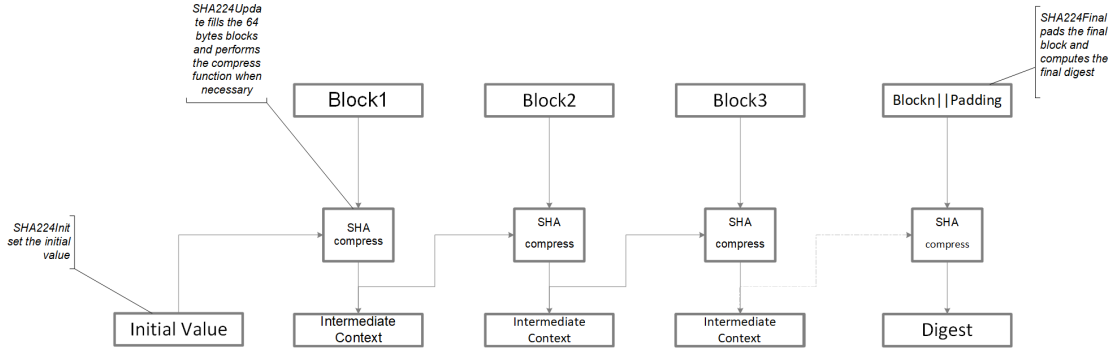
- The first step consists in initializing a context of 32 bytes with known constants.
- The second step consists in splitting the data to hash into blocks of 64 bytes. Each block is processed to update the context.
- Finally the data to hash is padded with a fix pattern and the data length. The context is updated accordingly. The digest consists in the 28 first bytes of last value of the context.

The step by step functions allow to perform SHA224 in several steps. Especially it permits to provide the data by block of any size.

The set of functions is constituted of 3 functions:

- An initialization function that initializes the context.
- An update function that accumulates the input data and processes the data as soon as 64 bytes are accumulated.
- A final function that performs the padding and computes the final digest.

Next image shows the principle of the step by step functions:



7.2.4.4.2 Context

SHA224_CTX is a structure that accumulates the intermediate data necessary to the hash computation. It is defined as follows in SHA224_CTX.

struct **SHA224_CTX**

SHA224 context

Public Members

uint32_t **Hash**[SHA224_INTERNAL_DIGEST_SIZE / 4]

Internal digest size

uint8_t **Block**[SHA224_BLOCK_SIZE]

Working buffer to accumulate data to hash

uint32_t **MessageLen**

Size in bytes of the message accumulated till now

uint8_t **Index**

Size of the data accumulated in the Block buffer

7.2.4.4.3 SHA224Init

Function

This function initializes a SHA224 computation. It initializes the SHA224_CTX.

SHA224_Lib_error_t **SHA224Init**(*SHA224_CTX* *ctx)

Initializes a SHA224 computation.

Parameters **ctx** – **[out]** SHA224 context

Return values **SHA224_SUCCESS** – Successful initialization

Returns Error status

7.2.4.4.4 Parameters

- **ctx** : a SHA224 context

7.2.4.4.5 Return values

Type	Description	OK \ NOK
SHA224_SUCCESS	SHA224 successfully initialized.	OK

7.2.4.4.6 SHA224Update

Function

This function processes the data provided by pbInput buffer. It accumulates the data in the context until the block is full (64 bytes). If the block is full, the data is hashed and the 32-byte long context is updated. The inputLength can be any size from 0 (included) to 0xFFFFFFFF. One will usually call SHA224Update several times. All the bytes of the message must be processed SHA224Update, including the potential incomplete block. On the other hand, the padding should not be provided to SHA224Update.

SHA224_Lib_error_t **SHA224Update**(*SHA224_CTX* *ctx, uint8_t *pbInput, uint32_t InputLength)

Updates the context with the data given in input.

Parameters

- **ctx** – **[inout]** SHA224 context
- **pbInput** – **[in]** Data to hash
- **InputLength** – **[inout]** Length of the data pointed by pbInput (in bytes)

Return values **SHA224_SUCCESS** – Successful computation

Returns Error status

7.2.4.4.7 Parameters

- **ctx** : a SHA224 context that was initialized previously.
- **pbInput** : data to hash of length InputLength.
- **InputLength** : length in bytes of the data (from 0x00 to 0xFFFFFFFF)

7.2.4.4.8 Return values

Type	Description	OK \ NOK
SHA224_SUCCESS	SHA224 update successful.	OK

7.2.4.4.9 SHA224Final

Function

SHA224Final adds the final padding and processes the context a final time to produce the digest.

SHA224_Lib_error_t **SHA224Final**(*SHA224_CTX* *ctx, uint8_t *pbResult)

Finalizes the hash computation and returns the result.

Parameters

- **ctx** – **[in]** SHA224 context
- **pbResult** – **[out]** Final digest (28 bytes)

Return values

- **SHA224_SUCCESS** – Successful computation
- **SHA224_INCORRECT_RESULT_POINTER** – Result buffer not initialized

Returns Error status

7.2.4.4.10 Parameters

- **ctx** : a SHA224 context that was initialized previously
- **pbResult** : pointer on the 28-byte hash digest

7.2.4.4.11 Return values

Type	Description	OK \ NOK
SHA224_SUCCESS	SHA224 update successful.	OK
SHA224_INCORRECT_RESULT_POINTER	pbResult is null pointer	NOK

7.2.5 Performances

7.2.5.1 Library location

The lib is located in ROM.

7.2.5.2 Code size

Size in bytes
948 bytes in ROM for the SHA224 module
In addition, the generic hash framework is 564 bytes

7.2.5.3 RAM

Size in bytes
No usage of global RAM except the SHA224_CTX

7.2.5.4 Stack

Size in bytes
Approximately 336 bytes

7.2.5.5 Execution time

The execution time of SHA224 naturally depends on the length of the message to hash. The data being processed by blocks of 64 bytes, the time will essentially depend on the number of 64-byte blocks the message has. Nonetheless for small data, one will take care that, due to the padding, an extra block may be added.

The next table shows the number of processed block(s) for various message lengths as example.

Data length in bytes	Number of processed blocks
0	1
1	1
55	1
56	2
57	2
63	2
64	2
128	3
1024	17

The performances of each function are given in the next table.

For the SHA224Update, besides the fact that it depends on the data length, it also depends on the amount of data already accumulated. The process of an extra 64-byte block may be necessary or not.

For the SHA224Final, 1 or 2 blocks must be processed depending on the overall length of the data to digest.

For the SHA224 API, it also depends on the data length. All together, the processing time of one 64-byte block is the most relevant figure.

Next table gives some examples of execution time.

Function	Number of cycles	Time in us at 48Mhz
Process of 1 64-byte block	5524	115
SHA224Init	115	2.3
SHA224Update(for 64 bytes)	5524	115
SHA224Final(for 1 block processed)	4917	102
SHA224(for 112 bytes)	11283	235

7.2.6 Dependencies

SHA224 lib depends on the generic hash framework library **hash.lib**.

Note that SHA224 and SHA256 algorithms are included in the same library.

7.2.7 Example

```
/*****************************************************************************
 *
 * FILE : ExampleSHA224.c
 *
 * Provide examples of use of SHA224 library
 *****************************************************************************/
#include <stdint.h>
#include "SHA224.h"

/*****************************************************************************
 Function: ExampleSHA224

 Basic example of use of the SHA224 library

 Return:
 0 if everything is ok
 1 otherwise

 Remark 1:
 SHA224 uses the generic hash Engine SHA_ARC.lib. Therefore, to call SHA224 one needs:
 - SHA_ARC.lib
 - SHA224_256_ARC.lib

 Remark 2:
 Two kinds of APIs are supported:
 * SHA224 API allows to perform a SHA224 hash in one step
 * SHA224Init, SHA224Update, SHA224Final allow to hash a message step by step.

 *****************************************************************************/
uint8_t ExampleSHA224(void) {
        uint8_t Digest[SHA224_DIGEST_SIZE];
        uint8_t i;
        uint8_t Error = 0;
        SHA224_CTX MyCtx;
        SHA224_Lib_error_t sw;

#define MESSAGE_LEN 112

        const uint8_t Message[MESSAGE_LEN] = { 0x61, 0x62, 0x63, 0x64, 0x65, 0x66,
                        0x67, 0x68, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x63,
                        0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x64, 0x65, 0x66, 0x67,
                        0x68, 0x69, 0x6a, 0x6b, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b,
                        0x6c, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x67, 0x68,
                        0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x68, 0x69, 0x6a, 0x6b, 0x6c,
```

(continues on next page)

(continued from previous page)

```
                        0x6d, 0x6e, 0x6f, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70,
                        0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x6b, 0x6c, 0x6d,
                        0x6e, 0x6f, 0x70, 0x71, 0x72, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71,
                        0x72, 0x73, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x6e,
                        0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75 };

        const uint8_t ExpectedResult[SHA224_DIGEST_SIZE] = { 0xc9, 0x7c, 0xa9, 0xa5,
                        0x59, 0x85, 0x0c, 0xe9, 0x7a, 0x04, 0xa9, 0x6d, 0xef, 0x6d, 0x99,
                        0xa9, 0xe0, 0xe0, 0xe2, 0xab, 0x14, 0xe6, 0xb8, 0xdf, 0x26, 0x5f,
                        0xc0, 0xb3 };

        //Example 1: we perform the computation in one step
        //-------------------------------------------------
        //perform the computation in one step
        sw = SHA224((uint8_t*) Message, Digest, MESSAGE_LEN);
        if (sw != SHA224_SUCCESS)
                Error++;

        //compare the result
        for (i = 0; i < SHA224_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //Example 2: we perform the computation in several steps
        //------------------------------------------------------
        // Note that in the two following examples, the size of the cuts message is␣
↪purely
        // arbitrary for the sake of examples. Any size, including 0 is allowed.

        //Example 2a. We first perform the update 64 bytes per 64 bytes.
        //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        //init
        sw = SHA224Init(&MyCtx);
        if (sw != SHA224_SUCCESS)
                Error++;
        //accumulate and perform computation of first 64 bytes
        sw = SHA224Update(&MyCtx, (uint8_t*) &Message[0], 64);
        if (sw != SHA224_SUCCESS)
                Error++;
        //continue on the next 112-64 following bytes
        sw = SHA224Update(&MyCtx, (uint8_t*) &Message[64], MESSAGE_LEN - 64);
        if (sw != SHA224_SUCCESS)
                Error++;
        //finalize the computation and get the result
        sw = SHA224Final(&MyCtx, Digest);
        if (sw != SHA224_SUCCESS)
                Error++;
        //compare the result
        for (i = 0; i < SHA224_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
```

(continues on next page)

(continued from previous page)

```
                        Error++;
                }
        }

        //Example 2b. We first perform the update 16 bytes per 16 bytes.
        //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        //init
        sw = SHA224Init(&MyCtx);
        if (sw != SHA224_SUCCESS)
                Error++;

        for (i = 0; i < 7; i++) {
                //accumulate and perform computation on 16 bytes
                sw = SHA224Update(&MyCtx, (uint8_t*) &Message[16 * i], 16);
                if (sw != SHA224_SUCCESS)
                        Error++;
        }
        //finalize the computation and get the result
        sw = SHA224Final(&MyCtx, Digest);
        if (sw != SHA224_SUCCESS)
                Error++;

        //compare the result
        for (i = 0; i < SHA224_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //final status of the test
        if (Error != 0)
                return (1);
        else
                return (0);

}
```

7.3 SHA256

7.3.1 Bibliography

[1] FIPS 180-4: Secure Hash Standard

FIPS 180-4: https://csrc.nist.gov/publications/detail/fips/180/4/final

7.3.2 Goal of the document

The goal of this document is to describe the functionality of the library **SHA256**. It describes:

- the supported functions,
- the parameters of the functions,
- the error cases,
- the performances of the functions.

This library implements SHA256 algorithm according to the specification FIPS 180-4 [1].

7.3.3 Types of APIs

It implements two sets of APIs:

- A first API allows to perform SHA256 in one shot: the message and its size are provided and the function computes the digest in one shot.
- A second set of APIs allows to provide the data to hash block by block. The computation of the digest can be performed step by step, accumulating data until the final processing.

7.3.4 APIs

7.3.4.1 Enumerations

enum **SHA256_Lib_error_t**

Error status words for SHA256.

Values:

enumerator **SHA256_SUCCESS**

SHA256 computation successful.

enumerator **SHA256_INCORRECT_RESULT_POINTER**

Result pointer is null.

7.3.4.2 Defines

SHA256_DIGEST_SIZE

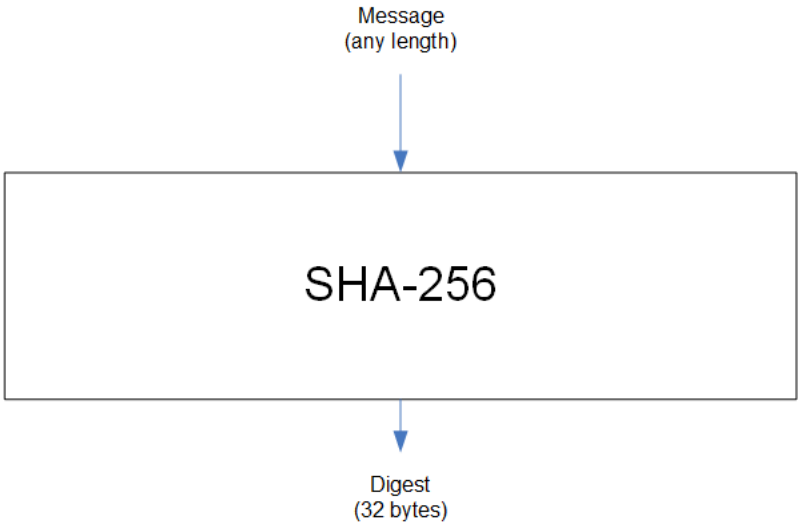
Digest size for SHA256 in bytes

SHA256_BLOCK_SIZE

Block size for SHA256 in bytes

7.3.4.3 ONE SHOT COMPUTATION FUNCTION : SHA256 API

This function computes the SHA256 digest of a data of any length. The computation is performed in one-shot. Next figure shows the general scheme.



7.3.4.3.1 Function

SHA256_Lib_error_t **SHA256**(uint8_t *Message, uint8_t *Result, uint32_t MessageLength)

Computes the hash of the message with SHA256.

Parameters

- **Message** – **[in]** Message to hash
- **Result** – **[out]** Digest of the message
- **MessageLength** – **[in]** Message length in bytes

Return values

- **SHA256_SUCCESS** – Successful computation
- **SHA256_INCORRECT_RESULT_POINTER** – Result buffer not initialized

Returns Error status

7.3.4.3.2 Parameters

- **Message** : The message to hash. It can be of any length.
- **Result**: Pointer on the 32-byte result.
- **MessageLength** : Length of the message in bytes. It can be any value from 0x00(included) to 0xFFFFFFFF (included).

7.3.4.3.3 Return values

Type	Description	OK \ NOK
SHA256_SUCCESS	SHA256 computation successful.	OK
SHA256_INCORRECT_RESULT_POINTER	Result pointer is null.	NOK

7.3.4.4 STEP BY STEP FUNCTIONS

7.3.4.4.1 Goal of the functions

According to [1] SHA256 algorithm is performed in several steps.

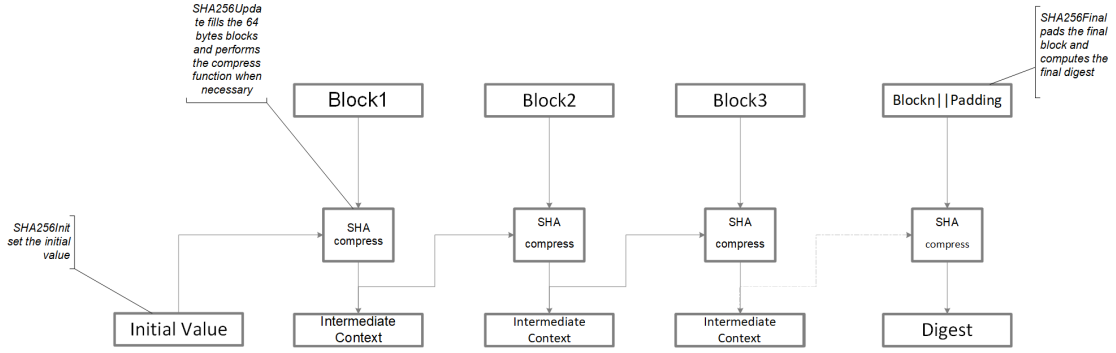
- The first step consists in initializing a context of 32 bytes with known constants.
- The second step consists in splitting the data to hash into blocks of 64 bytes. Each block is processed to update the context.
- Finally the data to hash is padded with a fix pattern and the data length. The context is updated accordingly. The digest consists in the last value of the 32-byte context.

The step by step functions allow to perform SHA256 in several steps. Especially it permits to provide the data by block of any size.

The set of functions is constituted of 3 functions:

- An initialization function that initializes the context.
- An update function that accumulates the input data and processes the data as soon as 64 bytes are accumulated.
- A final function that performs the padding and computes the final digest.

Next image shows the principle of the step by step functions:



7.3.4.4.2 Context

SHA256_CTX is a structure that accumulates the intermediate data necessary to the hash computation. It is defined as follows in SHA256_CTX.

struct **SHA256_CTX**

SHA256 context

Public Members

uint32_t **Hash**[SHA256_DIGEST_SIZE / 4]

Digest buffer

uint8_t **Block**[SHA256_BLOCK_SIZE]

Working buffer to accumulate data to hash

uint32_t **MessageLen**

Size in bytes of the message accumulated till now

uint8_t **Index**

Size of the data accumulated in the Block buffer

7.3.4.4.3 SHA256Init

Function

This function initializes a SHA256 computation. It initializes the SHA256_CTX.

SHA256_Lib_error_t **SHA256Init**(*SHA256_CTX* *ctx)

Initializes a SHA256 computation.

Parameters **ctx** – **[out]** SHA256 context

Return values **SHA256_SUCCESS** – Successful initialization

Returns Error status

7.3.4.4.4 Parameters

- **ctx** : a SHA256 context

7.3.4.4.5 Return values

Type	Description	OK \ NOK
SHA256_SUCCESS	SHA256 successfully initialized.	OK

7.3.4.4.6 SHA256Update

Function

This function processes the data provided by pbInput buffer. It accumulates the data in the context until the block is full (64 bytes). If the block is full, the data is hashed and the 32-byte long context is updated. The inputLength can be any size from 0x00(included) to 0xFFFFFFFF(included). One will usually call SHA256Update several times. All the bytes of the message must be processed SHA256Update, including the potential incomplete block. On the other hand, the padding should not be provided to SHA256Update.

SHA256_Lib_error_t **SHA256Update**(*SHA256_CTX* *ctx, uint8_t *pbInput, uint32_t InputLength)

Updates the context with the data given in input.

Parameters

- **ctx** – **[inout]** SHA-256 context
- **pbInput** – **[in]** data to hash
- **InputLength** – **[inout]** Length of the data pointed by pbInput (in bytes)

Return values **SHA256_SUCCESS** – Successful computation

Returns Error status

7.3.4.4.7 Parameters

- **ctx** : a SHA256 context that was initialized previously
- **pbInput** : data to hash of length InputLength
- **InputLength** : length in bytes of the data (from 0x00 to 0xFFFFFFFF)

7.3.4.4.8 Return values

Type	Description	OK \ NOK
SHA256_SUCCESS	SHA256 update successful.	OK

7.3.4.4.9 SHA256Final

Function

SHA256Final adds the final padding and processes the context a final time to produce the digest.

SHA256_Lib_error_t **SHA256Final**(*SHA256_CTX* *ctx, uint8_t *pbResult)

Finalizes the hash computation and returns the result.

Parameters

- **ctx** – **[in]** SHA256 context
- **pbResult** – **[out]** Final digest (32 bytes)

Return values

- **SHA256_SUCCESS** – Successful computation
- **SHA256_INCORRECT_RESULT_POINTER** – Result buffer not initialized

Returns Error status

7.3.4.4.10 Parameters

- **ctx** : a SHA256 context that was initialized previously
- **pbResult** : pointer on the 32-byte hash digest

7.3.4.4.11 Return values

Type	Description	OK \ NOK
SHA256_SUCCESS	SHA256 update successful	OK
SHA256_INCORRECT_RESULT_POINTER	pbResult is null pointer	NOK

7.3.5 Performances

7.3.5.1 Library location

The lib is located in ROM.

7.3.5.2 Code size

Size in bytes
948 bytes in ROM for the SHA256 module
In addition, the generic hash framework is 564 bytes

7.3.5.3 RAM

Size in bytes
No usage of global RAM except the SHA256_CTX

7.3.5.4 Stack

Size in bytes
Approximately 334 bytes

7.3.5.5 Execution time

The execution time of SHA256 naturally depends on the length of the message to hash. The data being processed by blocks of 64 bytes, the time will essentially depend on the number of 64-byte blocks the message has. Nonetheless for small data, one will take care that, due to the padding, an extra block may be added.

The next table shows the number of processed block(s) for various message lengths as example.

Data length in bytes	Number of processed blocks
0	1
1	1
55	1
56	1
57	2
63	2
64	2
128	3
1024	17

The performances of each function are given in the next table.

For the SHA256Update, besides the fact that it depends on the data length, it also depends on the amount of data already accumulated. The process of an extra 64-byte block may be necessary or not.

For the SHA256Final, 1 or 2 blocks must be processed depending on the overall length of the data to digest.

For the SHA256 API, it also depends on the data length. All together, the processing time of one 64-byte block is the most relevant figure.

Next table gives some examples of execution time.

Function	Number of cycles	Time in us at 48Mhz
Process of 1 64-byte block	5531	115
SHA256Init	115	2.3
SHA256Update(for 64 bytes)	5531	115
SHA256Final(for 1 block processed)	4940	102
SHA256(for 112 bytes)	11298	235

7.3.6 Dependencies

SHA256 lib depends on the generic hash framework library **hash.lib**.

Note that SHA224 and SHA256 algorithms are included in the same library.

7.3.7 Example

```
////////////////////////////////////////////////////////////////////////////////
///
/// @file       ExampleSHA256.c
///
/// @project    T9305
///
/// @author     SAS
///
/// @brief           Example of use of SHA256 library
///
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
///
////////////////////////////////////////////////////////////////////////////////
///
/// @copyright Copyright (C) 2022 EM Microelectronic
/// @cond
///
/// All rights reserved.
///
/// Redistribution and use in source and binary forms, with or without
/// modification, are permitted provided that the following conditions are met:
/// 1. Redistributions of source code must retain the above copyright notice,
/// this list of conditions and the following disclaimer.
/// 2. Redistributions in binary form must reproduce the above copyright notice,
/// this list of conditions and the following disclaimer in the documentation
/// and/or other materials provided with the distribution.
///
////////////////////////////////////////////////////////////////////////////////
///
/// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
/// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
/// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
/// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
/// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
/// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
/// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
/// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
/// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
/// POSSIBILITY OF SUCH DAMAGE.
/// @endcond
////////////////////////////////////////////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
#include <stdint.h>
#include "SHA256.h"

/*****************************************************************************
 Function: ExampleSHA256

 Basic example of use of the SHA256 library

 Return:
 0 if everything is ok
 1 otherwise

 Remark 1:
 SHA256 uses the generic hash Engine SHA_ARC.lib. Therefore, to call SHA256 one needs:
 - SHA_ARC.lib
 - SHA224_256_ARC.lib

 Remark 2:
 Two kinds of APIs are supported:
 * SHA256 API allows to perform a SHA256 hash in one step
 * SHA256Init, SHA256Update, SHA256Final allow to hash a message step by step.

 *****************************************************************************/
uint8_t ExampleSHA256(void) {
        uint8_t Digest[SHA256_DIGEST_SIZE];
        uint8_t i;
        uint8_t Error = 0;
        SHA256_CTX MyCtx;
        SHA256_Lib_error_t sw;

#define MESSAGE_LEN 112

        const uint8_t Message[MESSAGE_LEN] = { 0x61, 0x62, 0x63, 0x64, 0x65, 0x66,
                        0x67, 0x68, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x63,
                        0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x64, 0x65, 0x66, 0x67,
                        0x68, 0x69, 0x6a, 0x6b, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b,
                        0x6c, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x67, 0x68,
                        0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x68, 0x69, 0x6a, 0x6b, 0x6c,
                        0x6d, 0x6e, 0x6f, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70,
                        0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x6b, 0x6c, 0x6d,
                        0x6e, 0x6f, 0x70, 0x71, 0x72, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71,
                        0x72, 0x73, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x6e,
                        0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75 };

        const uint8_t ExpectedResult[SHA256_DIGEST_SIZE] = { 0xcf, 0x5b, 0x16, 0xa7,
                        0x78, 0xaf, 0x83, 0x80, 0x03, 0x6c, 0xe5, 0x9e, 0x7b, 0x04, 0x92,
                        0x37, 0x0b, 0x24, 0x9b, 0x11, 0xe8, 0xf0, 0x7a, 0x51, 0xaf, 0xac,
                        0x45, 0x03, 0x7a, 0xfe, 0xe9, 0xd1 };

        //Example 1: we perform the computation in one step
        //-------------------------------------------------
        //perform the computation in one step
```

(continues on next page)

(continued from previous page)

```
        sw = SHA256((uint8_t*) Message, Digest, MESSAGE_LEN);
        if (sw != SHA256_SUCCESS)
                Error++;

        //compare the result
        for (i = 0; i < SHA256_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //Example 2: we perform the computation in several steps
        //------------------------------------------------------
        // Note that in the two following examples, the size of the cuts message is␣
↪purely
        // arbitrary for the sake of examples. Any size, including 0 is allowed.

        //Example 2a. We first perform the update 64 bytes per 64 bytes.
        //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        //init
        sw = SHA256Init(&MyCtx);
        if (sw != SHA256_SUCCESS)
                Error++;
        //accumulate and perform computation of first 64 bytes
        sw = SHA256Update(&MyCtx, (uint8_t*) &Message[0], 64);
        if (sw != SHA256_SUCCESS)
                Error++;
        //continue on the next 112-64 following bytes
        sw = SHA256Update(&MyCtx, (uint8_t*) &Message[64], MESSAGE_LEN - 64);
        if (sw != SHA256_SUCCESS)
                Error++;
        //finalize the computation and get the result
        sw = SHA256Final(&MyCtx, Digest);
        if (sw != SHA256_SUCCESS)
                Error++;
        //compare the result
        for (i = 0; i < SHA256_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //Example 2b. We first perform the update 16 bytes per 16 bytes.
        //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        //init
        sw = SHA256Init(&MyCtx);
        if (sw != SHA256_SUCCESS)
                Error++;
        for (i = 0; i < 7; i++) {
                //accumulate and perform computation on 16 bytes
                sw = SHA256Update(&MyCtx, (uint8_t*) &Message[16 * i], 16);
                if (sw != SHA256_SUCCESS)
```

(continues on next page)

(continued from previous page)

```
                        Error++;
        }
        //finalize the computation and get the result
        sw = SHA256Final(&MyCtx, Digest);
        if (sw != SHA256_SUCCESS)
                Error++;
        //compare the result
        for (i = 0; i < SHA256_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //final status of the test
        if (Error != 0)
                return (1);
        else
                return (0);

}
```

7.4 SHA384

7.4.1 Bibliography

[1] FIPS 180-4: Secure Hash Standard

FIPS 180-4: https://csrc.nist.gov/publications/detail/fips/180/4/final

7.4.2 Goal of the document

The goal of this document is to describe the functionality of the library **SHA384**. It describes:

- the supported functions,
- the parameters of the functions,
- the error cases,
- the performances of the functions.

This library implements SHA384 algorithm according to the specification FIPS 180-4 [1].

7.4.3 Types of APIs

It implements two sets of APIs:

- A first API allows to perform SHA384 in one shot: the message and its size are provided and the function computes the digest in one shot.
- A second set of APIs allows to provide the data to hash block by block. The computation of the digest can be performed step by step, accumulating data until the final processing.

7.4.4 APIs

7.4.4.1 Enumerations

enum **SHA384_Lib_error_t**

Error status words for SHA384.

Values:

enumerator **SHA384_SUCCESS**

SHA384 computation successful.

enumerator **SHA384_INCORRECT_RESULT_POINTER**

Result pointer is null.

7.4.4.2 Defines

SHA384_DIGEST_SIZE

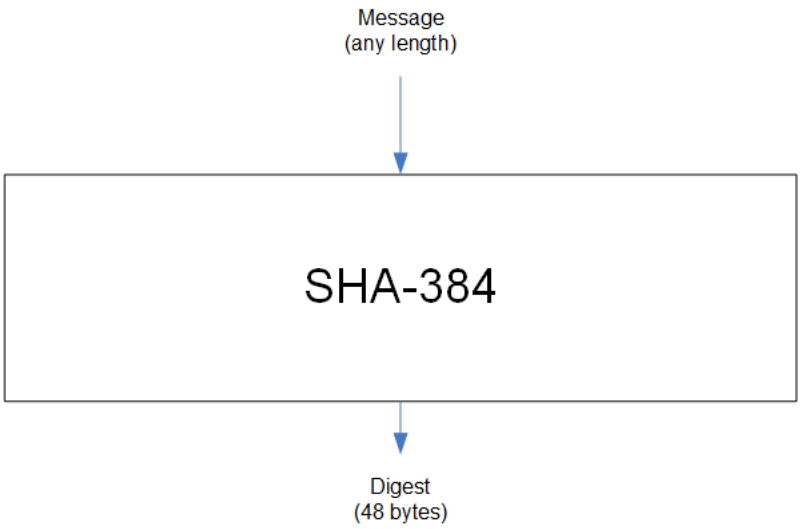
Digest size for SHA384 in bytes

SHA384_BLOCK_SIZE

Block size for SHA384 in bytes

7.4.4.3 ONE SHOT COMPUTATION FUNCTION : SHA384 API

This function computes the SHA384 digest of a data of any length. The computation is performed in one-shot. Next figure shows the general scheme.



7.4.4.3.1 Function

SHA384_Lib_error_t **SHA384**(uint8_t *Message, uint8_t *Result, uint32_t MessageLength)

Computes the hash of the message with SHA384.

Parameters

- **Message** – **[in]** Message to hash
- **Result** – **[out]** Digest of the message
- **MessageLength** – **[in]** Message length in bytes

Return values

- **SHA384_SUCCESS** – Successful computation
- **SHA384_INCORRECT_RESULT_POINTER** – Result buffer not initialized

Returns Error status

7.4.4.3.2 Parameters

- **Message** : The message to hash. It can be of any length.
- **Result**: Pointer on the 48-byte result.
- **MessageLength** : Length of the message in bytes. It can be any value from 0x00(included) to 0xFFFFFFFF(included).

7.4.4.3.3 Return values

Type	Description	OK \ NOK
SHA384_SUCCESS	SHA384 computation successful.	OK
SHA384_INCORRECT_RESULT_POINTER	Result pointer is null.	NOK

7.4.4.4 STEP BY STEP FUNCTIONS

7.4.4.4.1 Goal of the functions

According to [1] SHA384 algorithm is performed in several steps.

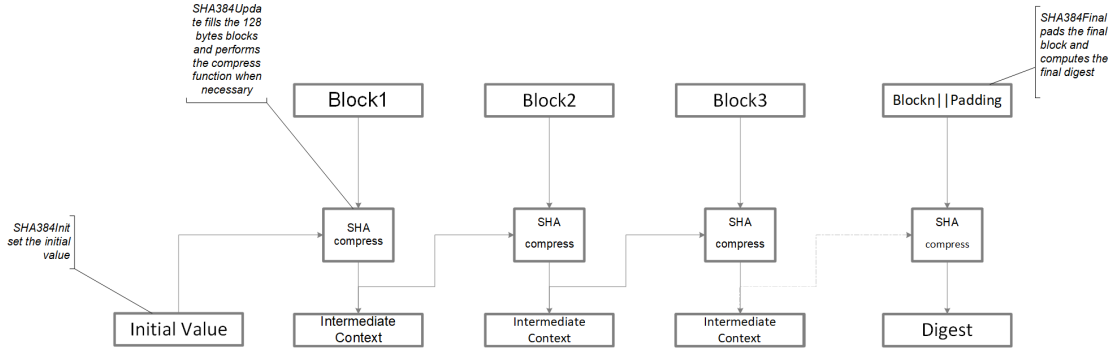
- The first step consists in initializing a context of 48 bytes with known constants.
- The second step consists in splitting the data to hash into blocks of 128 bytes. Each block is processed to update the context.
- Finally the data to hash is padded with a fix pattern and the data length. The context is updated accordingly. The digest consists in the last value of the 48-byte context.

The step by step functions allow to perform SHA384 in several steps. Especially it permits to provide the data by block of any size.

The set of functions is constituted of 3 functions:

- An initialization function that initializes the context.
- An update function that accumulates the input data and processes the data as soon as 128 bytes are accumulated.
- A final function that performs the padding and computes the final digest.

Next image shows the principle of the step by step functions:



7.4.4.4.2 Context

SHA384_CTX is a structure that accumulates the intermediate data necessary to the hash computation. It is defined as follows in SHA384_CTX.

struct **SHA384_CTX**

SHA384 context

Public Members

uint32_t **Hash**[SHA384_INTERNAL_DIGEST_SIZE / 4]

Internal digest size

uint8_t **Block**[SHA384_BLOCK_SIZE]

Working buffer to accumulate data to hash

uint32_t **MessageLen**

Size in bytes of the message accumulated till now

uint8_t **Index**

Size of the data accumulated in the Block buffer

7.4.4.4.3 SHA384Init

Function

This function initializes a SHA384 computation. It initializes the SHA384_CTX.

SHA384_Lib_error_t **SHA384Init**(*SHA384_CTX* *ctx)

Initializes a SHA384 computation.

Parameters **ctx** – **[out]** SHA384 context

Return values **SHA384_SUCCESS** – Successful initialization

Returns Error status

7.4.4.4.4 Parameters

- **ctx** : a SHA384 context

7.4.4.4.5 Return values

Type	Description	OK \ NOK
SHA384_SUCCESS	SHA384 successfully initialized.	OK

7.4.4.4.6 SHA384Update

Function

This function processes the data provided by pbInput buffer. It accumulates the data in the context until the block is full (128 bytes). If the block is full, the data are hashed and the 48-byte long context is updated. The inputLength can be any size from 0x00(included) to 0xFFFFFFFF. One will usually call SHA384Update several times. All the bytes of the message must be processed SHA384Update, including the potential incomplete block. On the other hand, the padding should not be provided to SHA384Update.

SHA384_Lib_error_t **SHA384Update**(*SHA384_CTX* *ctx, uint8_t *pbInput, uint32_t InputLength)

Updates the context with the data given in input.

Parameters

- **ctx** – **[inout]** SHA384 context
- **pbInput** – **[in]** data to hash
- **InputLength** – **[inout]** length of the data pointed by pbInput(in bytes)

Return values **SHA384_SUCCESS** – Successful computation

Returns Error status

7.4.4.4.7 Parameters

- **ctx** : a SHA384 context that was initialized previously
- **pbInput** : data to hash of length InputLength
- **InputLength** : length in bytes of the data (from 0x00 to 0xFFFFFFFF)

7.4.4.4.8 Return values

Type	Description	OK \ NOK
SHA384_SUCCESS	SHA384 update successful.	OK

7.4.4.4.9 SHA384Final

Function

SHA384Final adds the final padding and processes the context a final time to produce the digest.

SHA384_Lib_error_t **SHA384Final**(*SHA384_CTX* *ctx, uint8_t *pbResult)

Finalizes the hash computation and returns the result.

Parameters

- **ctx** – **[in]** SHA384 context
- **pbResult** – **[out]** Final digest (48 bytes)

Return values

- **SHA384_SUCCESS** – Successful computation
- **SHA384_INCORRECT_RESULT_POINTER** – Result buffer not initialized

Returns Error status

7.4.4.4.10 Parameters

- **ctx** : a SHA384 context that was initialized previously
- **pbResult** : pointer on the 48-byte hash digest

7.4.4.4.11 Return values

Type	Description	OK \ NOK
SHA384_SUCCESS	SHA384 update successful	OK
SHA384_INCORRECT_RESULT_POINTER	pbResult is null pointer	NOK

7.4.5 Performances

7.4.5.1 Library location

The lib is located in ROM.

7.4.5.2 Code size

Size in bytes
1996 bytes in ROM for the SHA384 module
In addition, the generic hash framework is 564 bytes

7.4.5.3 RAM

Size in bytes
No usage of global RAM except the SHA384_CTX

7.4.5.4 Stack

Size in bytes
Approximately 384 bytes

7.4.5.5 Execution time

The execution time of SHA384 naturally depends on the length of the message to hash. The data being processed by blocks of 128 bytes, the time will essentially depend on the number of 128-byte blocks the message has. Nonetheless for small data, one will take care that, due to the padding, an extra block may be added.

The next table shows the number of processed block(s) for various message lengths as example.

Data length in bytes	Number of processed blocks
0	1
1	1
111	1
112	2
128	2
256	3
1024	9

The performances of each function are given in the next table.

For the SHA384Update, besides the fact that it depends on the data length, it also depends on the amount of data already accumulated. The process of an extra 128-byte block may be necessary or not.

For the SHA384Final, 1 or 2 blocks must be processed depending on the overall length of the data to digest.

For the SHA384 API, it also depends on the data length. All together, the processing time of one 128-byte block is the most relevant figure.

Next table gives some examples of execution time.

Function	Number of cycles	Time in us at 48Mhz
Process of 1 128-byte block	89240	1864
SHA384Init	211	4.39
SHA384Update(for 128 bytes)	89240	1863
SHA384Final(for 1 block processed)	87674	1926
SHA384(for 112 bytes)	89495	1864

7.4.6 Dependencies

SHA384 lib depends on the generic hash framework library **hash.lib**.

Note that SHA384 and SHA512 algorithms are included in the same library.

7.4.7 Example

```
////////////////////////////////////////////////////////////////////////////////
///
/// @file       ExampleSHA384.c
///
/// @project    T9305
///
/// @author     SAS
///
/// @brief           Example of use of SHA384 library
///
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
///
////////////////////////////////////////////////////////////////////////////////
///
/// @copyright Copyright (C) 2022 EM Microelectronic
/// @cond
///
/// All rights reserved.
///
/// Redistribution and use in source and binary forms, with or without
/// modification, are permitted provided that the following conditions are met:
/// 1. Redistributions of source code must retain the above copyright notice,
/// this list of conditions and the following disclaimer.
/// 2. Redistributions in binary form must reproduce the above copyright notice,
/// this list of conditions and the following disclaimer in the documentation
/// and/or other materials provided with the distribution.
///
////////////////////////////////////////////////////////////////////////////////
///
/// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
/// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
/// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
/// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
/// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
/// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
/// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
/// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
/// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
/// POSSIBILITY OF SUCH DAMAGE.
/// @endcond
////////////////////////////////////////////////////////////////////////////////
#include <stdint.h>
```

(continues on next page)

(continued from previous page)

```
#include "SHA384.h"

/*****************************************************************************
 Function: ExampleSHA384

 Basic example of use of the SHA384 library

 Return:
 0 if everything is ok
 1 otherwise

 Remark 1:
 SHA384 uses the generic hash Engine SHA_ARC.lib. Therefore, to call SHA384 one needs:
 - SHA_ARC.lib
 - SHA384_512_ARC.lib

 Remark 2:
 Two kinds of APIs are supported:
 * SHA384 API allows to perform a SHA384 hash in one step
 * SHA384Init, SHA384Update, SHA384Final allow to hash a message step by step.

 *****************************************************************************/
uint8_t ExampleSHA384(void) {
        uint8_t Digest[SHA384_DIGEST_SIZE];
        uint8_t i;
        uint8_t Error = 0;
        SHA384_CTX MyCtx;
        SHA384_Lib_error_t sw;

#define MESSAGE_LEN 112

        const uint8_t Message[MESSAGE_LEN] = { 0x61, 0x62, 0x63, 0x64, 0x65, 0x66,
                        0x67, 0x68, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x63,
                        0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x64, 0x65, 0x66, 0x67,
                        0x68, 0x69, 0x6a, 0x6b, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b,
                        0x6c, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x67, 0x68,
                        0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x68, 0x69, 0x6a, 0x6b, 0x6c,
                        0x6d, 0x6e, 0x6f, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70,
                        0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x6b, 0x6c, 0x6d,
                        0x6e, 0x6f, 0x70, 0x71, 0x72, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71,
                        0x72, 0x73, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x6e,
                        0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75 };

        const uint8_t ExpectedResult[SHA384_DIGEST_SIZE] = { 0x09, 0x33, 0x0c, 0x33,
                        0xf7, 0x11, 0x47, 0xe8, 0x3d, 0x19, 0x2f, 0xc7, 0x82, 0xcd, 0x1b,
                        0x47, 0x53, 0x11, 0x1b, 0x17, 0x3b, 0x3b, 0x05, 0xd2, 0x2f, 0xa0,
                        0x80, 0x86, 0xe3, 0xb0, 0xf7, 0x12, 0xfc, 0xc7, 0xc7, 0x1a, 0x55,
                        0x7e, 0x2d, 0xb9, 0x66, 0xc3, 0xe9, 0xfa, 0x91, 0x74, 0x60, 0x39␣
↪};

        //Example 1: we perform the computation in one step
        //--------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
        //perform the computation in one step
        sw = SHA384((uint8_t*) Message, Digest, MESSAGE_LEN);
        if (sw != SHA384_SUCCESS)
                Error++;

        //compare the result
        for (i = 0; i < SHA384_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //Example 2: we perform the computation in several steps
        //------------------------------------------------------
        // Note that in the two following examples, the size of the cuts message is␣
↪purely
        // arbitrary for the sake of examples. Any size, including 0 is allowed.

        //Example 2a. We first perform the update 64 bytes per 64 bytes.
        //- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        //init
        sw = SHA384Init(&MyCtx);
        if (sw != SHA384_SUCCESS)
                Error++;
        //accumulate and perform computation of first 64 bytes
        sw = SHA384Update(&MyCtx, (uint8_t*) &Message[0], 64);
        if (sw != SHA384_SUCCESS)
                Error++;
        //continue on the next 112-64 following bytes
        sw = SHA384Update(&MyCtx, (uint8_t*) &Message[64], MESSAGE_LEN - 64);
        if (sw != SHA384_SUCCESS)
                Error++;
        //finalize the computation and get the result
        sw = SHA384Final(&MyCtx, Digest);
        if (sw != SHA384_SUCCESS)
                Error++;
        //compare the result
        for (i = 0; i < SHA384_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //Example 2b. We first perform the update 16 bytes per 16 bytes.
        //- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        //init
        sw = SHA384Init(&MyCtx);
        if (sw != SHA384_SUCCESS)
                Error++;

        for (i = 0; i < 7; i++) {
                //accumulate and perform computation on 16 bytes
```

(continues on next page)

(continued from previous page)

```
                sw = SHA384Update(&MyCtx, (uint8_t*) &Message[16 * i], 16);
                if (sw != SHA384_SUCCESS)
                        Error++;
        }
        //finalize the computation and get the result
        sw = SHA384Final(&MyCtx, Digest);
        if (sw != SHA384_SUCCESS)
                Error++;

        //compare the result
        for (i = 0; i < SHA384_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //final status of the test
        if (Error != 0)
                return (1);
        else
                return (0);

}
```

7.5 SHA512

7.5.1 Bibliography

[1] FIPS 180-4: Secure Hash Standard

FIPS 180-4: https://csrc.nist.gov/publications/detail/fips/180/4/final

7.5.2 Goal of the document

The goal of this document is to describe the functionality of the library **SHA512**. It describes:

- the supported functions,
- the parameters of the functions,
- the error cases,
- the performances of the functions.

This library implements SHA512 algorithm according to the specification FIPS 180-4 [1].

7.5.3 Types of APIs

It implements two sets of APIs:

- A first API allows to perform SHA512 in one shot: the message and its size are provided and the function computes the digest in one shot.
- A second set of APIs allows to provide the data to hash block by block. The computation of the digest can be performed step by step, accumulating data until the final processing.

7.5.4 APIs

7.5.4.1 Enumerations

enum **SHA512_Lib_error_t**

Error status words for SHA512.

Values:

enumerator **SHA512_SUCCESS**

SHA512 computation successful.

enumerator **SHA512_INCORRECT_RESULT_POINTER**

Result pointer is null.

7.5.4.2 Defines

SHA512_DIGEST_SIZE

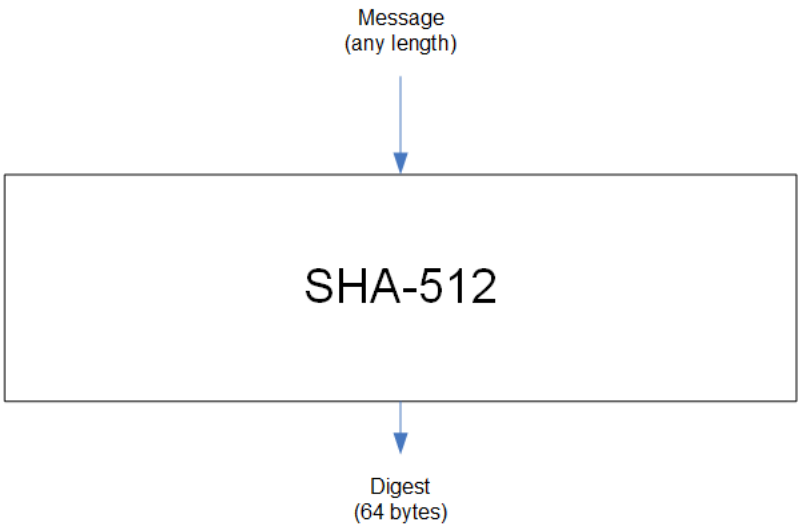
Digest size for SHA512 in bytes

SHA512_BLOCK_SIZE

Block size for SHA512 in bytes

7.5.4.3 ONE SHOT COMPUTATION FUNCTION : SHA512 API

This function computes the SHA512 digest of a data of any length. The computation is performed in one-shot. Next figure shows the general scheme.



7.5.4.3.1 Function

SHA512_Lib_error_t **SHA512**(uint8_t *Message, uint8_t *Result, uint32_t MessageLength)

Computes the hash of the message with SHA512.

Parameters

- **Message** – **[in]** Message to hash
- **Result** – **[out]** Digest of the message
- **MessageLength** – **[in]** Message length in bytes

Return values

- **SHA512_SUCCESS** – Successful computation
- **SHA512_INCORRECT_RESULT_POINTER** – Result buffer not initialized

Returns Error status

7.5.4.3.2 Parameters

- **Message** : The message to hash. It can be of any length.
- **Result**: Pointer on the 64-byte result.
- **MessageLength** : Length of the message in bytes. It can be any value from 0x00 (included) to 0xFFFFFFFF (included).

7.5.4.3.3 Return values

Type	Description	OK \ NOK
SHA512_SUCCESS	SHA512 computation successful.	OK
SHA512_INCORRECT_RESULT_POINTER	Result pointer is null.	NOK

7.5.4.4 STEP BY STEP FUNCTIONS

7.5.4.4.1 Goal of the functions

According to [1] SHA512 algorithm is performed in several steps.

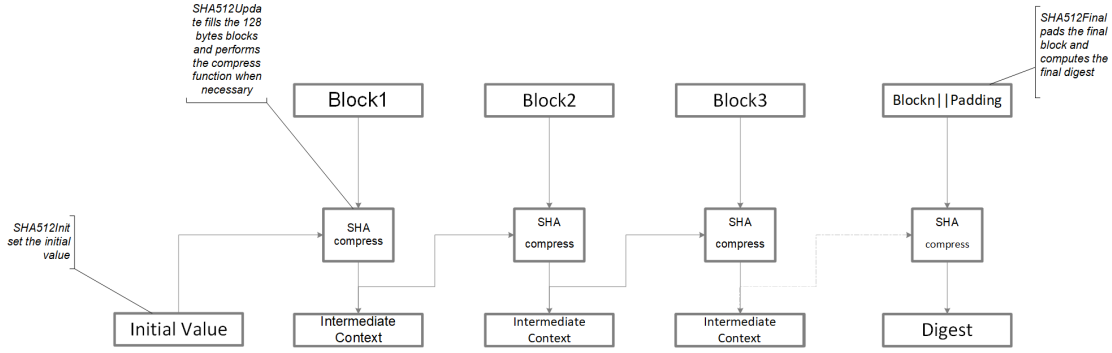
- The first step consists in initializing a context of 64 bytes with known constants.
- The second step consists in splitting the data to hash into blocks of 128 bytes. Each block is processed to update the context.
- Finally the data to hash is padded with a fix pattern and the data length. The context is updated accordingly. The digest consists in the last value of the 64-byte context.

The step by step functions allow to perform SHA512 in several steps. Especially it permits to provide the data by block of any size.

The set of functions is constituted of 3 functions:

- An initialization function that initializes the context.
- An update function that accumulates the input data and processes the data as soon as 128 bytes are accumulated.
- A final function that performs the padding and computes the final digest.

Next image shows the principle of the step by step functions:



7.5.4.4.2 Context

SHA512_CTX is a structure that accumulates the intermediate data necessary to the hash computation. It is defined as follows in SHA512_CTX.

struct **SHA512_CTX**

SHA512 context

Public Members

uint32_t **Hash**[SHA512_DIGEST_SIZE / 4]

Digest buffer

uint8_t **Block**[SHA512_BLOCK_SIZE]

Working buffer to accumulate data to hash

uint32_t **MessageLen**

Size in bytes of the message accumulated till now

uint8_t **Index**

Size of the data accumulated in the Block buffer

7.5.4.4.3 SHA512Init

Function

This function initializes a SHA512 computation. It initializes the SHA512_CTX.

SHA512_Lib_error_t **SHA512Init**(*SHA512_CTX* *ctx)

Initializes a SHA512 computation.

Parameters **ctx** – **[out]** SHA512 context

Return values **SHA512_SUCCESS** – Successful initialization

Returns Error status

7.5.4.4.4 Parameters

- **ctx** : a SHA512 context

7.5.4.4.5 Return values

Type	Description	OK \ NOK
SHA512_SUCCESS	SHA512 successfully initialized.	OK

7.5.4.4.6 SHA512Update

Function

This function processes the data provided by pbInput buffer. It accumulates the data in the context until the block is full (128 bytes). If the block is full, the data are hashed and the 64-byte long context is updated. The inputLength can be any size from 0x00 (included) to 0xFFFFFFFF. One will usually call SHA512Update several times. All the bytes of the message must be processed SHA512Update, including the potential incomplete block. On the other hand, the padding should not be provided to SHA512Update.

SHA512_Lib_error_t **SHA512Update**(*SHA512_CTX* *ctx, uint8_t *pbInput, uint32_t InputLength)

Finalizes the hash computation and returns the result.

Parameters

- **ctx** – **[inout]** SHA-512 context
- **pbInput** – **[in]** data to hash
- **InputLength** – **[inout]** length of the data pointed by pbInput (in byte)

Return values **SHA512_SUCCESS** – Successful computation

Returns Error status

7.5.4.4.7 Parameters

- **ctx** : a SHA512 context that was initialized previously.
- **pbInput** : data to hash of length InputLength
- **InputLength** : length in bytes of the data (from 0x00 to 0xFFFFFFFF)

7.5.4.4.8 Return values

Type	Description	OK \ NOK
SHA512_SUCCESS	SHA512 update successful	OK

7.5.4.4.9 SHA512Final

Function

SHA512Final adds the final padding and processes the context a final time to produce the digest.

SHA512_Lib_error_t **SHA512Final**(*SHA512_CTX* *ctx, uint8_t *pbResult)

Finalizes the hash computation and returns the result.

Parameters

- **ctx** – **[in]** SHA512 context
- **pbResult** – **[out]** Final digest (64 bytes)

Return values

- **SHA512_SUCCESS** – Successful computation
- **SHA512_INCORRECT_RESULT_POINTER** – Result buffer not initialized

Returns Error status

7.5.4.4.10 Parameters

- **ctx** : a SHA512 context that was initialized previously.
- **pbResult** : pointer on the 64-byte hash digest

7.5.4.4.11 Return values

Type	Description	OK \ NOK
SHA512_SUCCESS	SHA512 update successful.	OK
SHA512_INCORRECT_RESULT_POINTER	pbResult is null pointer	NOK

7.5.5 Performances

7.5.5.1 Library location

The lib is located in ROM.

7.5.5.2 Code size

Size in bytes
1996 bytes in ROM for the SHA512 module
In addition, the generic hash framework is 564 bytes

7.5.5.3 RAM

Size in bytes
No usage of global RAM except the SHA512_CTX

7.5.5.4 Stack

Size in bytes
Approximately 384 bytes

7.5.5.5 Execution time

The execution time of SHA512 naturally depends on the length of the message to hash. The data being processed by blocks of 128 bytes, the time will essentially depend on the number of 128-byte blocks the message has. Nonetheless for small data, one will take care that, due to the padding, an extra block may be added.

The next table shows the number of processed block(s) for various message lengths as example.

Data length in bytes	Number of processed blocks
0	1
1	1
111	1
112	2
128	2
256	3
1024	9

The performances of each function are given in the next table.

For the SHA512Update, besides the fact that it depends on the data length, it also depends on the amount of data already accumulated. The process of an extra 128-byte block may be necessary or not.

For the SHA512Final, 1 or 2 blocks must be processed depending on the overall length of the data to digest.

For the SHA512 API, it also depends on the data length. All together, the processing time of one 128-byte block is the most relevant figure.

Next table gives some examples of execution time.

Function	Number of cycles	Time in us at 48Mhz
Process of 1 128-byte block	89240	1864
SHA512Init	211	4.39
SHA512Update(for 128 bytes)	89240	1863
SHA512Final(for 1 block processed)	87674	1926
SHA512(for 112 bytes)	89495	1864

7.5.6 Dependencies

SHA512 lib depends on the generic hash framework library **hash.lib**.

Note that SHA384 and SHA512 algorithms are included in the same library.

7.5.7 Example

```
////////////////////////////////////////////////////////////////////////////////
///
/// @file       ExampleSHA512.c
///
/// @project    T9305
///
/// @author     SAS
///
/// @brief          Example of use of SHA512 library
///
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
///
////////////////////////////////////////////////////////////////////////////////
///
/// @copyright Copyright (C) 2022 EM Microelectronic
/// @cond
///
/// All rights reserved.
///
/// Redistribution and use in source and binary forms, with or without
/// modification, are permitted provided that the following conditions are met:
/// 1. Redistributions of source code must retain the above copyright notice,
/// this list of conditions and the following disclaimer.
/// 2. Redistributions in binary form must reproduce the above copyright notice,
/// this list of conditions and the following disclaimer in the documentation
/// and/or other materials provided with the distribution.
///
////////////////////////////////////////////////////////////////////////////////
///
/// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
/// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
/// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
/// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
/// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
/// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
/// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
/// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
/// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
/// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
/// POSSIBILITY OF SUCH DAMAGE.
/// @endcond
////////////////////////////////////////////////////////////////////////////////
#include <stdint.h>
```

(continues on next page)

(continued from previous page)

```
#include "SHA512.h"

/*****************************************************************************
 Function: ExampleSHA512

 Basic example of use of the SHA512 library

 Return:
 0 if everything is ok
 1 otherwise

 Remark 1:
 SHA512 uses the generic hash Engine SHA_ARC.lib. Therefore, to call SHA512 one needs:
 - SHA_ARC.lib
 - SHA384_512_ARC.lib

 Remark 2:
 Two kinds of APIs are supported:
 * SHA512 API allows to perform a SHA512 hash in one step
 * SHA512Init, SHA512Update, SHA512Final allow to hash a message step by step.

 *****************************************************************************/
uint8_t ExampleSHA512(void) {
        uint8_t Digest[SHA512_DIGEST_SIZE];
        uint8_t i;
        uint8_t Error = 0;
        SHA512_CTX MyCtx;
        SHA512_Lib_error_t sw;

#define MESSAGE_LEN 112

        const uint8_t Message[MESSAGE_LEN] = { 0x61, 0x62, 0x63, 0x64, 0x65, 0x66,
                        0x67, 0x68, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x63,
                        0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x64, 0x65, 0x66, 0x67,
                        0x68, 0x69, 0x6a, 0x6b, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b,
                        0x6c, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x67, 0x68,
                        0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x68, 0x69, 0x6a, 0x6b, 0x6c,
                        0x6d, 0x6e, 0x6f, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70,
                        0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x6b, 0x6c, 0x6d,
                        0x6e, 0x6f, 0x70, 0x71, 0x72, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71,
                        0x72, 0x73, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x6e,
                        0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75 };

        const uint8_t ExpectedResult[SHA512_DIGEST_SIZE] = { 0x8e, 0x95, 0x9b, 0x75,
                        0xda, 0xe3, 0x13, 0xda, 0x8c, 0xf4, 0xf7, 0x28, 0x14, 0xfc, 0x14,
                        0x3f, 0x8f, 0x77, 0x79, 0xc6, 0xeb, 0x9f, 0x7f, 0xa1, 0x72, 0x99,
                        0xae, 0xad, 0xb6, 0x88, 0x90, 0x18, 0x50, 0x1d, 0x28, 0x9e, 0x49,
                        0x00, 0xf7, 0xe4, 0x33, 0x1b, 0x99, 0xde, 0xc4, 0xb5, 0x43, 0x3a,
                        0xc7, 0xd3, 0x29, 0xee, 0xb6, 0xdd, 0x26, 0x54, 0x5e, 0x96, 0xe5,
                        0x5b, 0x87, 0x4b, 0xe9, 0x09 };

        //Example 1: we perform the computation in one step
```

(continues on next page)

(continued from previous page)

```
        //-------------------------------------------------
        //perform the computation in one step
        sw = SHA512((uint8_t*) Message, Digest, MESSAGE_LEN);
        if (sw != SHA512_SUCCESS)
                Error++;

        //compare the result
        for (i = 0; i < SHA512_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //Example 2: we perform the computation in several steps
        //-------------------------------------------------------
        // Note that in the two following examples, the size of the cuts message is␣
↪purely
        // arbitrary for the sake of examples. Any size, including 0 is allowed.

        //Example 2a. We first perform the update 64 bytes per 64 bytes.
        //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        //init
        sw = SHA512Init(&MyCtx);
        if (sw != SHA512_SUCCESS)
                Error++;
        //accumulate and perform computation of first 64 bytes
        sw = SHA512Update(&MyCtx, (uint8_t*) &Message[0], 64);
        if (sw != SHA512_SUCCESS)
                Error++;
        //continue on the next 112-64 following bytes
        sw = SHA512Update(&MyCtx, (uint8_t*) &Message[64], MESSAGE_LEN - 64);
        if (sw != SHA512_SUCCESS)
                Error++;
        //finalize the computation and get the result
        sw = SHA512Final(&MyCtx, Digest);
        if (sw != SHA512_SUCCESS)
                Error++;
        //compare the result
        for (i = 0; i < SHA512_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //Example 2b. We first perform the update 16 bytes per 16 bytes.
        //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        //init
        sw = SHA512Init(&MyCtx);
        if (sw != SHA512_SUCCESS)
                Error++;

        for (i = 0; i < 7; i++) {
```

(continues on next page)

(continued from previous page)

```
                //accumulate and perform computation on 16 bytes
                sw = SHA512Update(&MyCtx, (uint8_t*) &Message[16 * i], 16);
                if (sw != SHA512_SUCCESS)
                        Error++;
        }
        //finalize the computation and get the result
        sw = SHA512Final(&MyCtx, Digest);
        if (sw != SHA512_SUCCESS)
                Error++;

        //compare the result
        for (i = 0; i < SHA512_DIGEST_SIZE; i++) {
                if (Digest[i] != ExpectedResult[i]) {
                        Error++;
                }
        }

        //final status of the test
        if (Error != 0)
                return (1);
        else
                return (0);

}
```

CHAPTER

EIGHT

DOCUMENT VERSION

Version	Date	Description
1.0	12 Sept 2022	Initial version
2.0	06 Oct 2023	Description of the PRNG health tests
		Add tables with Min, Mean, Max times for InitPRNG for the 4 cores
2.1	23 Oct 2023	Typos corrected