UM0586
User manual

STM32 Cryptographic Library

Introduction

This manual describes the API of the STM32 cryptographic library (STM32-CRYP-LIB) that supports the following cryptographic algorithms:

- AES-128, AES-192, AES-256 bits. Supported modes are:
  - ECB (Electronic Codebook Mode)
  - CBC (Cipher-Block Chaining) with support for ciphertext stealing
  - CTR (CounTer Mode)
  - CCM (Counter with CBC-MAC)
  - GCM (Galois Counter Mode)
  - CMAC
  - KEY WRAP
- ARC4
- DES, TripleDES. Supported modes are:
  - ECB (Electronic Codebook Mode)
  - CBC (Cipher-Block Chaining)
- HASH functions with HMAC support:
  - MD5
  - SHA-1
  - SHA-224
  - SHA-256
- Random engine based on DRBG-AES-128
- RSA signature functions with PKCS#1v1.5
- ECC (Elliptic Curve Cryptography):
  - Key generation
  - Scalar multiplication (the base for ECDH)
  - ECDSA

These cryptographic algorithms can run in the series STM32F1, STM32 L1, STM32F2, STM32F4, STM32F0 and STM32F3 with hardware enhancement accelerators.

Contents

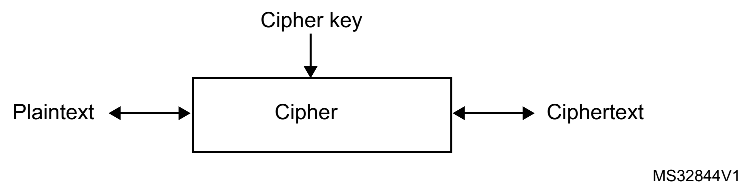
List of tables

List of figures

1 Terminology

Encryption is a branch of cryptographic science. It is the transformation that converts data to illegible data, with the view of making it secure. The following block diagram (see *Figure 1*) shows a commonly used encryption system structure.

Figure 1. Block diagram of a common cipher



The following terms are used throughout this document:

- Cipher: a suite of transformations that converts plaintext to ciphertext and ciphertext to plaintext, using the cipher key:
  - transformation from plaintext to ciphertext is called enciphering or encryption
  - transformation from ciphertext to plaintext is called deciphering or decryption
- Cipher key: a private key that is used by the cipher to perform cryptographic operations. The cipher key size is the important element that determines the security level of the encryption algorithm.
- Plaintext: raw data to be encrypted.
  - In the case of an encryption, it is the input of the cipher,
  - In the case of a decryption, it is the output of the cipher.
- Ciphertext: converted data. result of plaintext encryption.
- Symmetric cipher: cipher that uses a single key for enciphering and deciphering
- Asymmetric cipher: cipher that uses two keys, one for enciphering and the other for deciphering.

2 STM32 cryptographic library package presentation

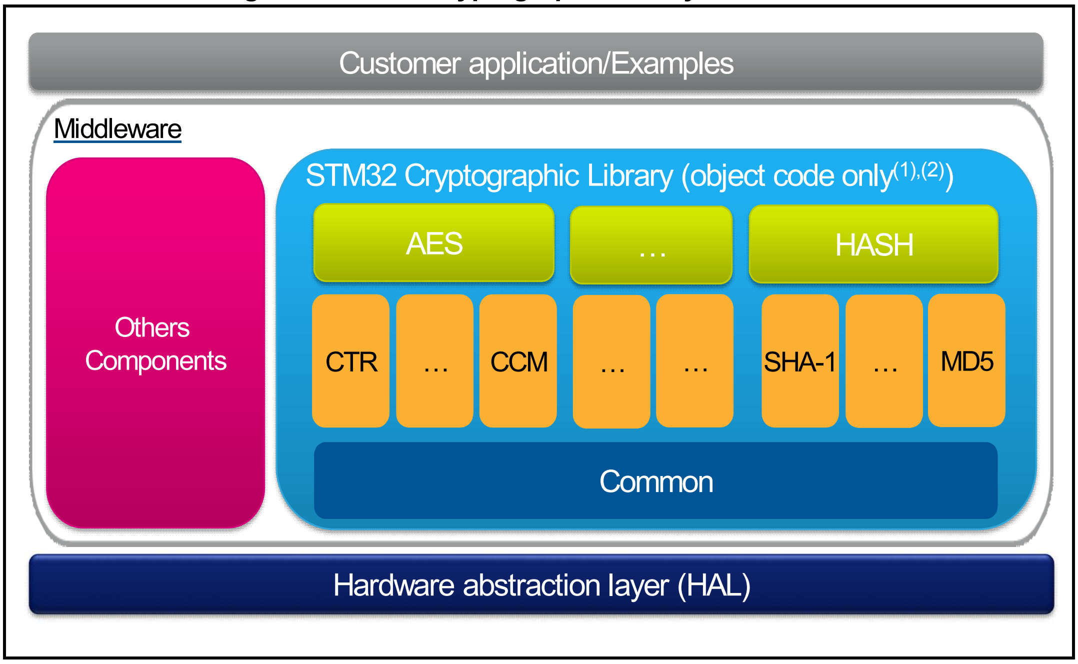
2.1 Architecture

The library is built around a modular programming model ensuring:

- independencies between the components building the main application
- easy porting on a large product range
- use of integrated firmware components for other applications with minimum changes to common code.

The following figure provides a global view of the STM32 cryptographic library usage and interaction with other firmware components.

Figure 2. STM32 cryptographic library architecture



Note: 1. For algorithms that are not supported by the HW Cryptographic peripheral, only the firmware version will be available when enabling HW acceleration.

Note: 2. HW acceleration is only available for STM32F21x and STM32F41x devices. For other devices, all cryptographic algorithms are implemented in firmware.

Note: 3. CRC peripheral is used.

- The HAL controls the STM32 device registers and features based on two main libraries:
  - CMSIS layer:
    Core Peripheral Access Layer.
    STM32xx Device Peripheral Access Layer.
  - STM32 standard peripheral driver.
- STM32 cryptographic library: As presensented in *Figure 2*, the STM32 cryptographic library is based on modular archictecture that means new algorithms can be added

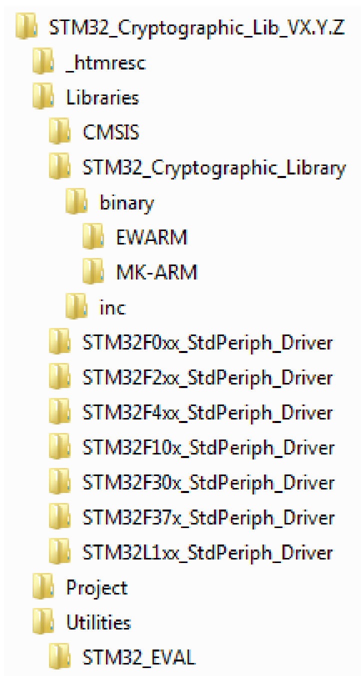
without any impact on the current implementation. To provide flexibility for cryptographic functions, each algorithm can be compiled with different options to manage the memory and execution speed.
Chapter 11 is dedicated to the performance evaluation of the cryptographic library for the STM32 microcontroller series. This analysis targets the STM32F4xx family in particular as the series STM32F41x includes some cryptographic accelerators.

- Application layer: The application layer consists of a set of examples covering all available algorithms with template projects for the most common development tools. Even without the appropriate hardware evaluation board, this layer allows you to rapidly get started with a brand new STM32 cryptographic library.

2.2 Package organization

The library is supplied in a zip file. The extraction of the zip file generates one folder, STM32_Cryptographic_Lib_VX.Y.Z, which contains the following subfolders:

Figure 3. STM32 cryptographic library package organization



Note: VX.Y.Z refers to the library version, ex. V1.0.0

The STM32 cryptographic library package consists of three main folders: Libraries, Projects and Utilities.

2.2.1 Libraries

This folder contains two subfolders, CMSIS files and STM32 cryptographic library, followed by drivers for STM32 Standard Peripheral.

- CMSIS subfolder: contains STM32F0xx, STM32F2xx, STM32F4xx, STM32F10x, STM32F30x, STM32F37x and STM32L1xx CMSIS files
- STM32_Cryptographic_library: contains two subfolders: binary and inc

  binary: contains five STM32 cryptographic libraries for each Development Toolchain, in the EWARM and MDK-ARM subfolders:

  a) EWARM: Contains eight STM32 crytographic libraries compiled with IAR toolchain 6.5.30 with high speed optimization:
  M4_CryptoHW_2_0_6.a: STM32 Cryptographic Library for **STM32F41x** families.
  M4_CryptoFW_RngHW_2_0_6.a: STM32 Cryptographic Library Firmware with Hardware RNG pheripheral for **STM32F4xx** families.
  M4_CryptoFW_2_0_6.a: STM32 Cryptographic Library Firmware for **STM32F40x** families.
  M3_CryptoHW_2_0_6.a: STM32 Cryptographic Library for **STM32F21x** families.
  M3_CryptoFW_RngHW_2_0_6.a: STM32 Cryptographic Library Firmware with Hardware RNG pheripheral for **STM32F20x** families.
  M3_CryptoFW_2_0_6.a: STM32 Cryptographic Library Firmware for **STM32F10x** and **STM32F3xx**.
  M3_CryptoFW_L1xx_2_0_6.a: STM32 Cryptographic Library Firmware for **STM32L1xx** families.
  M0_CryptoFW_2_0_6.a: STM32 Cryptographic Library Firmware for **STM32F0xx** families.

  b) MDK-ARM: Contains eight STM32 crytographic libraries compiled with Keil toolchain 4.70 with optimization level 3(-O 3):
  M4_CryptoHW_2_0_6.lib: STM32 Cryptographic Library for **STM32F41x** families.
  M4_CryptoFW_RngHW_2_0_6. lib: STM32 Cryptographic Library Firmware with Hardware RNG pheripheral for **STM32F4xx** families.
  M4_CryptoFW_2_0_6. lib: STM32 Cryptographic Library Firmware for **STM32F40x** families.
  M3_CryptoHW_2_0_6. lib: STM32 Cryptographic Library for **STM32F21x** families.
  M3_CryptoFW_RngHW_2_0_6. lib: STM32 Cryptographic Library Firmware with Hardware RNG pheripheral for **STM32F20x** families.
  M3_CryptoFW_2_0_6. lib: STM32 Cryptographic Library Firmware for **STM32F10x** and **STM32F3xx**.
  M3_CryptoFW_L1xx_2_0_6. lib: STM32 Cryptographic Library Firmware for **STM32L1xx** families.
  M0_CryptoFW_2_0_6. lib: STM32 Cryptographic Library Firmware for **STM32F0xx** families.

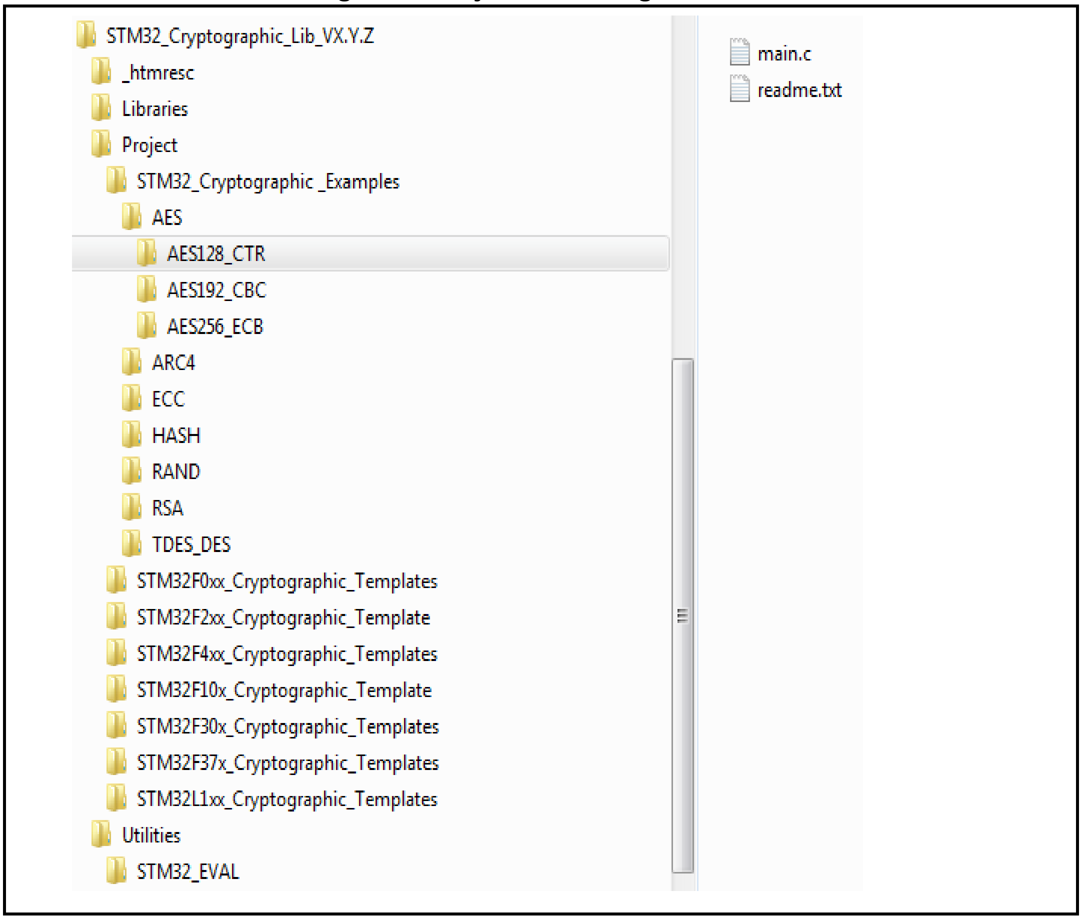
  inc: contains all header files used by STM32 cryptographic library
- The remaining folders contain standard drivers for STM32 standard peripherals.

2.2.2 Project

This folder contains dedicated subfolders of STM32_Cryptographic_Examples that contain sets of examples by algorithms as presented in *Figure 4*.

We provide a project template for EWARM, MDK-ARM tool chain for each STM32 series STM32F0xx, STM32F2xx, STM324xx, STM3210, STM32F30x, STM32F37x and STM32L1xx.

Figure 4. Project folder organization



2.2.3 Utilities

This folder contains the abstraction layer that allows interaction with the human interface resources (buttons, LEDs, LCD and COM ports (USARTs)) available on STMicroelectronics evaluation boards.

Note: *All examples provided in this package are independant of external hardware.*

3 DES and Triple-DES algorithms

3.1 Description

The data encryption standard (DES) is a symmetric cipher algorithm that can process data blocks of 64 bits under the control of a 64-bit key. The DES core algorithm uses 56 bits for enciphering and deciphering, and 8 bits for parity, so the DES cipher key size is 56 bits.

The DES cipher key size has become insufficient to guarantee algorithm security, thus the Triple-DES (TDES) has been devised to expand the key from 56 bits to 168 bits (56 × 3) while keeping the same algorithm core.

The Triple-DES is a suite of three DES in series, making three DES encryptions with three different keys.

The STM32 cryptographic library includes the functions required to support DES and Triple-DES modules to perform encryption and decryption using the following modes:

- ECB (Electronic Codebook Mode)
- CBC (Cipher-Block Chaining)

These modes can run with the STM32F1, STM32L1, STM32F20x, STM32F05x, STM32F40x, STM32F37x and the STM32F30x series using a software algorithm implementation.

You can optimize the performance by using pure hardware accelerators in the STM32F21x and STM32F41x devices.

For DES and Triple-DES library settings, refer to *Section 10: STM32 encryption library settings*.

For DES and Triple-DES library performance and memory requirements, refer to *Section 11: Cryptographic library performance and memory requirements*.

3.2 DES library functions

Table 1 describes the encryption library's DES functions.

Table 1. DES algorithm functions (DDD = ECB or CBC)

Function name	Description
DES_DDD_Encrypt_Init	Initialization for DES Encryption in DDD mode
DES_DDD_Encrypt_Append	DES Encryption in DDD mode
DES_DDD_Encrypt_Finish	DES Encryption Finalization of DDD mode
DES_DDD_Decrypt_Init	Initialization for DES Decryption in DDD mode
DES_DDD_Decrypt_Append	DES Decryption in DDD mode
DES_DDD_Decrypt_Finish	DES Decryption Finalization in DDD mode

DDD represents the mode of operation of the DES algorithm, it is either ECB or CBC.

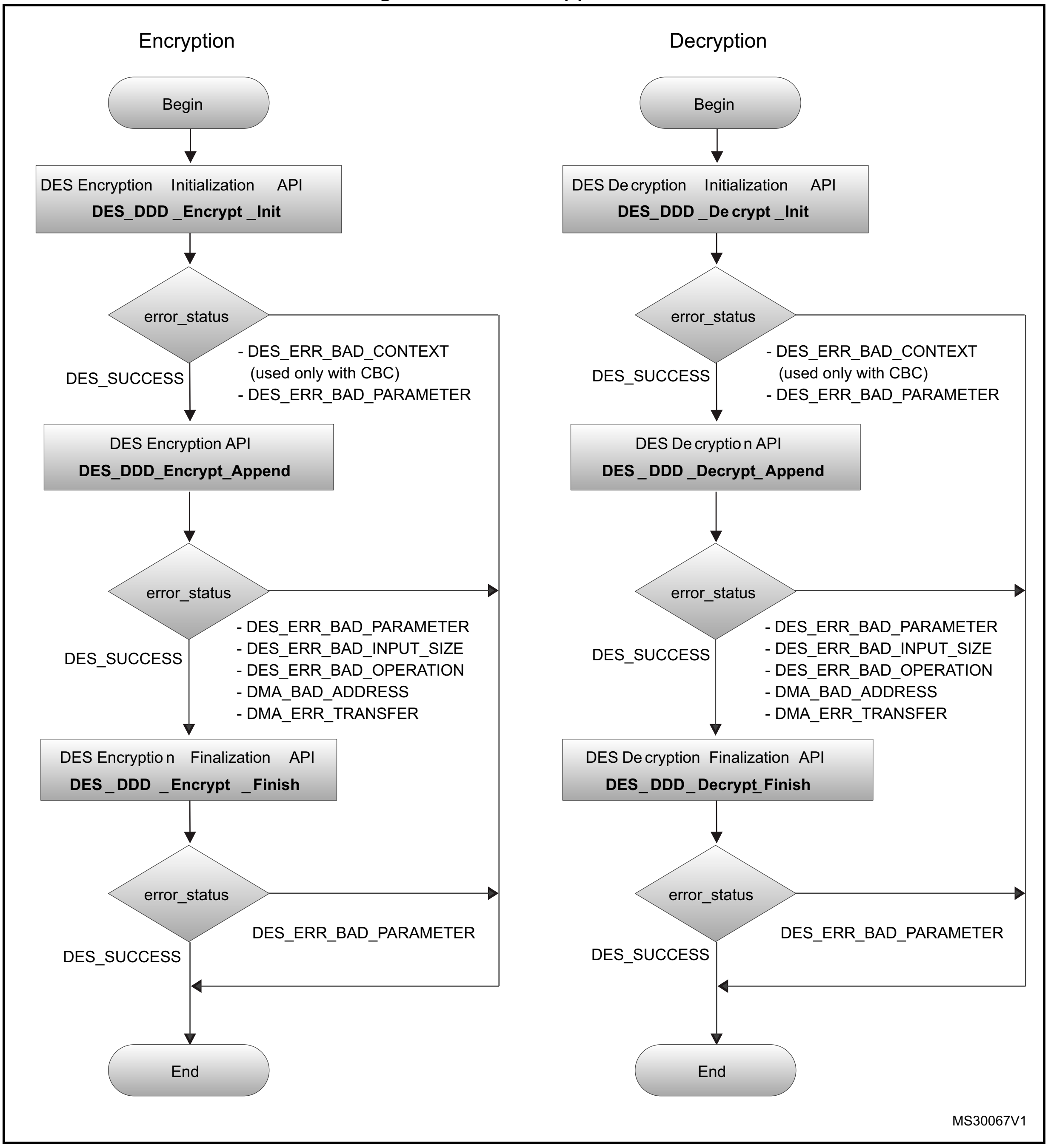
For example, if you want to use ECB mode as a DES algorithm, you can use the following functions:

Table 2. DES ECB algorithm functions

Function name	Description
DES_ECB_Encrypt_Init	Initialization for DES Encryption in ECB mode
DES_ECB_Encrypt_Append	DES Encryption in ECB mode
DES_ECB_Encrypt_Finish	DES Encryption Finalization of ECB mode
DES_ECB_Decrypt_Init	Initialization for DES Decryption in ECB mode
DES_ECB_Decrypt_Append	DES Decryption in ECB mode
DES_ECB_Decrypt_Finish	DES Decryption Finalization in ECB mode

Figure 5 describes the DES algorithm.

Figure 5. DES DDD(*) flowchart



3.2.1 DES_DDD_Encrypt_Init function

Table 3. DES_DDD_Encrypt_Init

Function name	DES_DDD_Encrypt_Init [(1)]

Prototype	`int32_t DES_DDD_Encrypt_Init (`
	`DESDDDctx_stt * P_pDESDDDctx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pIv )`

Behavior	Initialization for DES Encryption in DDD mode
Parameter	– [in, out] *P_pDESDDDctx: DES DDD context – [in] *P_pKey: Buffer with the Key – [in] *P_pIv: Buffer with the IV[(2)]
Return value	– DES_SUCCESS : Operation Successful – DES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – DES_ERR_BAD_CONTEXT: Context not initialized with valid values, see note 2 below (This return value is only used with CBC algorithm)

1. DDD is ECB or CBC
2. In ECB: IV is not used, so the value of P_pIv is not checked or used.
In CBC: IV size must be already written inside the fields of P_pDESCBCctx. The IV size must be at least 1 and at most 16 to avoid the DES_ERR_BAD_CONTEXT return.

Note: 1. P_pDESDDDctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.

2. P_pDESCBCctx.mIvSize must be set with the size of the IV (default CRL_DES_BLOCK) prior to calling this function.

DESDDDctx_stt data structure

Structure type for public key.

Table 4. DESDDDctx_stt data structure

Field name	Description
`uint32_t mContextId`	Unique ID of this context. Not used in current implementation
`SKflags_et mFlags`	32 bit mFlags, used to perform keyschedule see *SKflags_et mFlags*
`const uint8_t * pmKey`	Pointer to original key buffer
`const uint8_t * pmIv`	Pointer to original initialization vector buffer
`int32_t mIvSize`	Size of the initialization vector in bytes
`uint32_t amIv[2]`	Temporary result/IV
`uint32_t amExpKey[32]`	Expanded DES key

SKflags_et mFlags

Enumeration of allowed flags in a context for symmetric key operations.

Table 5. SKflags_et mFlags

Field name	Description
E_SK_DEFAULT	User Flag: No flag specified. This is the default value for this flag
E_SK_DONT_PERFORM_KEY_SCHEDULE	User Flag: Forces the initialization to **not** perform key schedule. The classic example is where the same key is used on a new message. In this case redoing key scheduling is a waste of computation.
E_SK_USE_DMA	User Flag: Used when there is a HW engine for DES. It specifies whether DMA or CPU should transfer data. It is common to always use the DMA, except when DMA is very busy or input data is very small.
E_SK_FINAL_APPEND	User Flag: Must be set in some modes before final Append call occurs.
E_SK_OPERATION_COMPLETED	Internal Flag: Checks that the Finish function has been called.
E_SK_NO_MORE_APPEND_ALLOWED	Internal Flag: Set when the last append has been called. Used where the append is called with an InputSize not multiple of the block size, which means that is the last input.
E_SK_NO_MORE_HEADER_APPEND_ALLOWED	Internal Flag: only for authenticated encryption modes. It is set when the last header append has been called. Used where the header append is called with an InputSize not multiple of the block size, which means that is the last input.
E_SK_APPEND_DONE	Internal Flag: not used in this algorithm

3.2.2 DES_DDD_Encrypt_Append function

Table 6. DES_DDD_Encrypt_Append

Function name	DES_DDD_Encrypt_Append(1)

Prototype	`int32_t DES_DDD_Encrypt_Append(`
	`DESDDDctx_stt * P_pDESDDDctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize)`

Behavior	DES Encryption in DDD mode
Parameter	– [in] *P_pDESDDDctx: DES DDD, already initialized, context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize Size: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– DES_SUCCESS: Operation Successful – DES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – DES_ERR_BAD_INPUT_SIZE: the P_inputSize is not a multiple of CRL_DES_BLOCK or less than 8 – DMA_BAD_ADDRESS: Input or output buffer addresses are not word aligned – DMA_ERR_TRANSFER: Error occurred in the DMA transfer – DES_ERR_BAD_OPERATION: Append not allowed

1. DDD is ECB or CBC.

Note: This function can be called multiple times, provided that P_inputSize is a multiple of 8.

3.2.3 DES_DDD_Encrypt_Finish function

Table 7. DES_DDD_Encrypt_Finish

Function name	DES_DDD_Encrypt_Finish(1)

Prototype	`int32_t DES_DDD_Encrypt_Finish (`
	`DESDDDctx_stt * P_pDESDDDctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	DES Encryption Finalization of DDD mode
Parameter	– [in,out] *P_pDESDDDctx: DES DDD, already initialized, context – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– DES_SUCCESS: Operation Successful – DES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer

1. DDD is ECB or CBC.

Note: This function won't write output data, thus it can be skipped.It is kept for API compatibility.

3.2.4 DES_DDD_Decrypt_Init function

Table 8. DES_DDD_Decrypt_Init

Function name	DES_DDD_Decrypt_Init(1)

Prototype	`int32_t DES_DDD_Decrypt_Init (`
	`DESDDDctx_stt * P_pDESDDDctx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pIv )`

Behavior	Initialization for DES Decryption in DDD Mode
Parameter	– [in,out] *P_pDESDDDctx: DES DDD context – [in] *P_pKey: Buffer with the Key – [in] *P_pIv: Buffer with the IV(2)
Return value	– DES_SUCCESS: Operation Successful – DES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – DES_ERR_BAD_CONTEXT: Context not initialized with valid values, see note 2 below (This return value is only used with CBC algorithm)

1. DDD is ECB or CBC
2. In ECB: IV is not used, so the value of P_pIv is not checked or used.
In CBC: IV size must be already written inside the fields of P_pDESCBCctx. The IV size must be at least 1 and at most 16 to avoid the DES_ERR_BAD_CONTEXT return.

Note: *1. P_pDESDDDctx.mFlags must be set before calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.*

2. P_pDESCBCctx.mIvSize must be set with the size of the IV (default CRL_DES_BLOCK) prior to calling this function.

3.2.5 DES_DDD_Decrypt_Append function

Table 9. DES_DDD_Decrypt_Append

Function name	DES_DDD_Decrypt_Append(1)

Prototype	`int32_t DES_DDD_Decrypt_Append (`
	`DESDDDctx_stt * P_pDESDDDctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	DES Decryption in DDD mode
Parameter	– [in] *P_pDESDDDctx: DES DDD, already initialized, context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize Size: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– DES_SUCCESS: Operation Successful – DES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – DES_ERR_BAD_INPUT_SIZE: P_inputSize is not a multiple of CRL_DES_BLOCK or less than 8 – DMA_BAD_ADDRESS: Input or output buffer addresses are not word aligned – DMA_ERR_TRANSFER: Error occurred in the DMA transfer – DES_ERR_BAD_OPERATION: Append not allowed

1. DDD is ECB or CBC

Note: This function can be called multiple times, provided that P_inputSize is a multiple of 8.

3.2.6 DES_DDD_Decrypt_Finish function

Table 10. DES_DDD_Decrypt_Finish

Function name	DES_DDD_Decrypt_Finish(1)

Prototype	`int32_t DES_ECB_Decrypt_Finish (`
	`DESDDDctx_stt * P_pDESECBctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	DES Decryption Finalization of DDD Mode
Parameter	– [in,out] *P_pDESDDDctx: DES DDD, already initialized, context – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– DES_SUCCESS: Operation Successful – DES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer

1. DDD is ECB or CBC

Note: This function won't write output data, thus it can be skipped.It is kept for API compatibility.

3.3 TDES library functions

Table 11 describes the encryption library's TDES functions.below

Table 11. TDES algorithm functions (TTT = ECB or CBC)

Function name	Description
TDES_TTT_Encrypt_Init	Initialization for TDES Encryption in TTT mode
TDES_TTT_Encrypt_Append	TDES Encryption in TTT mode
TDES_TTT_Encrypt_Finish	TDES Encryption Finalization of TTT mode
TDES_TTT_Decrypt_Init	Initialization for TDES Decryption in TTT mode
TDES_TTT_Decrypt_Append	TDES Decryption in TTT mode
TDES_TTT_Decrypt_Finish	TDES Decryption Finalization in TTT mode

TTT represents the mode of operations of the TDES algorithm.

The following modes of operation can be used for TDES algorithm:

- ECB
- CBC

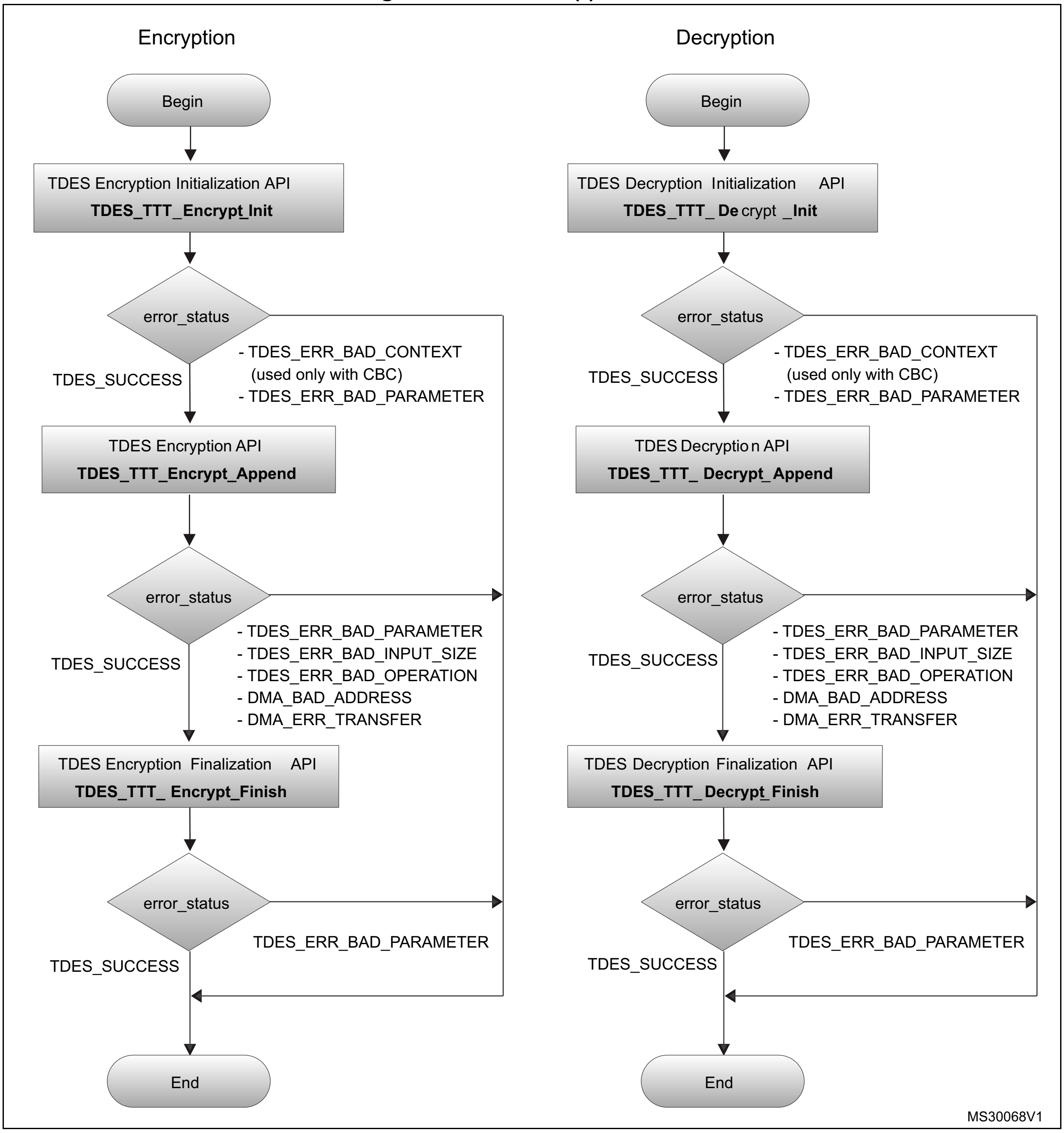
Figure 6 describes the TDES algorithm:

For example, if you want to use ECB mode as a TDES algorithm, you can use the following functions:

Table 12. TDES ECB algorithm functions

Function name	Description
TDES_ECB_Encrypt_Init	Initialization for TDES Encryption in ECB mode
TDES_ECB_Encrypt_Append	TDES Encryption in ECB mode
TDES_ECB_Encrypt_Finish	TDES Encryption Finalization of ECB mode
TDES_ECB_Decrypt_Init	Initialization for TDES Decryption in ECB mode
TDES_ECB_Decrypt_Append	TDES Decryption in ECB mode
TDES_ECB_Decrypt_Finish	TDES Decryption Finalization in ECB mode

Figure 6. TDES TTT(*) flowchart



3.3.1 TDES_TTT_Encrypt_Init function

Table 13. TDES_TTT_Encrypt_Init

Function name	TDES_TTT_Encrypt_Init(1)

Prototype	`int32_t TDES_DDD_Encrypt_Init (`
	`TDESTTTctx_stt * P_pTDESTTTctx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pIv )`

Behavior	Initialization for TDES Encryption in DDD Mode
Parameter	– [in,out] *P_pTDESDDDctx: TDES TTT context – [in] *P_pKey: Buffer with the Key – [in] *P_pIv: Buffer with the IV(2)
Return value	– TDES_SUCCESS: Operation Successful – TDES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – TDES_ERR_BAD_CONTEXT: Context not initialized with valid values, see note 2 below (This return value is only used with CBC algorithm)

1. TTT is ECB or CBC
2. In ECB: IV is not used, so the value of P_pIv is not checked or used.

   In CBC: IV size must be already written inside the fields of P_pTDESCBCctx. The IV size must be at least 1 and at most 16 to avoid the TDES_ERR_BAD_CONTEXT return.

Note: *1. P_pTDESTTTctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.*

2. P_pTDESCBCctx.mIvSize must be set with the size of the IV (default CRL_TDES_BLOCK) prior to calling this function.

TDESTTTctx_stt data structure

Structure type for public key.

Table 14. TDESTTTctx_stt data structure

Field name	Description
`uint32_t mContextId`	Unique ID of this context. Not used in current implementation
`SKflags_et mFlags`	32 bit mFlags, used to perform keyschedule, see *SKflags_et mFlags*
`const uint8_t * pmKey`	Pointer to original Key buffer
`const uint8_t * pmIv`	Pointer to original Initialization Vector buffer
`int32_t mIvSize`	Size of the Initialization Vector in bytes
`uint32_t amIv[2]`	Temporary result/IV
`uint32_t amExpKey[96]`	Expanded TDES key

3.3.2 TDES_TTT_Encrypt_Append function

Table 15. TDES_TTT_Encrypt_Append

Function name	TDES_TTT_Encrypt_Append(1)

Prototype	`int32_t TDES_TTT_Encrypt_Append(`
	`TDESTTTctx_stt * P_pTDESTTTctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize)`

Behavior	TDES Encryption in TTT mode
Parameter	– [in] *P_pTDESTTTctx: TDES TTT, already initialized, context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize Size: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– TDES_SUCCESS: Operation Successful – TDES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – TDES_ERR_BAD_INPUT_SIZE: the P_inputSize is not a multiple of CRL_DES_BLOCK or less than 8 – DMA_BAD_ADDRESS: Input/output buffer address not word aligned – DMA_ERR_TRANSFER: Error occurred in the DMA transfer – TDES_ERR_BAD_OPERATION: Append not allowed

1. TTT is ECB or CBC

Note: This function can be called multiple times, provided that P_inputSize is a multiple of 8.

3.3.3 TDES_TTT_Encrypt_Finish function

Table 16. TDES_TTT_Encrypt_Finish

Function name	TDES_TTT_Encrypt_Finish(1)

Prototype	`int32_t TDES_TTT_Encrypt_Finish (`
	`TDESTTTctx_stt * P_pTDESTTTctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	TDES Encryption Finalization of TTT mode
Parameter	– [in,out] *P_pTDESTTTctx: TDES TTT, already initialized, context – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– TDES_SUCCESS: Operation Successful – TDES_ERR_BAD_PARAMETER: At least one parameter is NULL pointer

1. TTT is ECB or CBC

Note: This function won't write output data, thus it can be skipped.It is kept for API compatibility.

3.3.4 TDES_TTT_Decrypt_Init function

Table 17. TDES_TTT_Decrypt_Init

Function name	**TDES_TTT_Decrypt_Init** (1)

Prototype	`int32_t TDES_TTT_Decrypt_Init (`
	`TDESTTTctx_stt * P_pTDESTTTctx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pIv )`

Behavior	Initialization for TDES Decryption in TTT Mode
Parameter	– [in,out] *P_pTDESTTTctx: TDES TTT context – [in] *P_pKey: Buffer with the Key – [in] *P_pIv: Buffer with the IV(2)
Return value	– TDES_SUCCESS: Operation Successful – TDES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – TDES_ERR_BAD_CONTEXT: Context not initialized with valid values, see note 2 below (This return value is only used with CBC algorithm)

1. TTT is ECB or CBC
2. In ECB: IV is not used, so the value of P_pIv is not checked or used.
In CBC: IV size must be already written inside the fields of P_pTDESCBCctx. The IV size must be at least 1 and at most 16 to avoid the TDES_ERR_BAD_CONTEXT return.

Note: *1. P_pTDESTTTctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.*

2. P_pTDESCBCctx.mIvSize must be set with the size of the IV (default CRL_TDES_BLOCK) prior to calling this function.

3.3.5 TDES_TTT_Decrypt_Append function

Table 18. TDES_TTT_Decrypt_Append

Function name	TDES_TTT_Decrypt_Append(1)

Prototype	`int32_t TDES_TTT_Decrypt_Append (`
	`TDESTTTctx_stt * P_pTDESTTTctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	TDES Decryption in TTT mode
Parameter	– [in] *P_pTDESTTTctx: DES TTT, already initialized, context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize Size: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– TDES_SUCCESS: Operation Successful – TDES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – TDES_ERR_BAD_INPUT_SIZE: the P_inputSize is not a multiple of CRL_DES_BLOCK or less than 8 – DMA_BAD_ADDRESS: Input or output buffer addresses are not word aligned – DMA_ERR_TRANSFER: Error occurred in the DMA transfer – TDES_ERR_BAD_OPERATION: Append not allowed

1. TTT is ECB or CBC

Note: This function can be called multiple times, provided that P_inputSize is a multiple of 8.

3.3.6 TDES_TTT_Decrypt_Finish function

Table 19. TDES_TTT_Decrypt_Finish

Function name	TDES_TTT_Decrypt_Finish(1)

Prototype	`int32_t TDES_ECB_Decrypt_Finish (`
	`TDESTTTctx_stt * P_pTDESECBctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	TDES Decryption Finalization of TTT Mode
Parameter	– [in,out] *P_pTDESTTTctx: DES TTT, already initialized, context – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– TDES_SUCCESS: Operation Successful – TDES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer

1. TTT is ECB or CBC

Note: This function won't write output data, thus it can be skipped.It is kept for API compatibility

3.4 DES with ECB mode example

Main DES enciphering and deciphering example:

```
#include "crypto.h"
const uint8_t Plaintext[PLAINTEXT_LENGTH] = { 0x54, 0x68, 0x65, 0x20, 0x71,
0x75, 0x66, 0x63, 0x6B, 0x20, 0x62, 0x72, 0x6F, 0x77, 0x6E, 0x20, 0x66,
0x6F, 0x78, 0x20, 0x6A, 0x75, 0x6D, 0x70};
/* Key to be used for AES encryption/decryption */
uint8_t Key[CRL_TDES_KEY] = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD,
0xEF, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF, 0x01, 0x45, 0x67, 0x89,
0xAB, 0xCD, 0xEF, 0x01, 0x23 };
int32_t main()
 {
/* Buffer to store the output data */
uint8_t OutputMessage[PLAINTEXT_LENGTH];
TDESECBctx_stt TDESctx;
uint32_t error_status = TDES_SUCCESS;
int32_t outputLength = 0;
/* Set flag field to default value */
  TDESctx.mFlags = E_SK_DEFAULT;
/* Initialize the operation, by passing the key.
   * Third parameter is NULL because ECB doesn't use any IV */
  error_status = TDES_ECB_Encrypt_Init(&TDESctx, TDES_Key, NULL );
/* check for initialization errors */
  if (error_status == TDES_SUCCESS)
  {
    /* Encrypt Data */
    error_status = TDES_ECB_Encrypt_Append(&TDESctx, Plaintext,
PLAINTEXT_LENGTH
                                           OutputMessage, &outputLength);
if (error_status == TDES_SUCCESS)
    {
      /* Write the number of data written*/
      *OutputMessageLength = outputLength;
      /* Do the Finalization */
      error_status = TDES_ECB_Encrypt_Finish(&TDESctx, OutputMessage +
*OutputMessageLength, &outputLength);
      /* Add data written to the information to be returned */
      *OutputMessageLength += outputLength;
    }
  }
}
```

4 AES algorithm

4.1 Description

The advanced encryption standard (AES), known as the Rijndael algorithm, is a symmetric cipher algorithm that can process data blocks of 128 bits, using a key with a length of 128, 192 or 256 bits.

The STM32 cryptographic library includes AES 128-bit, 192-bit and 256-bit modules to perform encryption and decryption in the following modes:

- ECB (Electronic Codebook mode)
- CBC (Cipher-Block Chaining) with support for Ciphertext Stealing
- CTR (CounTer mode)
- CCM (Counter with CBC-MAC)
- GCM (Galois Counter mode)
- CMAC
- KEY WRAP

These modes can run with the STM32F1, STM32L1, STM32F20x, STM32F05x, STM32F40x, STM32F37x and the STM32F30x series using a software algorithm implementation. The STM32F21x and STM32F41x series include cryptographic accelerators, in particular a cryptographic Accelerator capable of encrypting/decrypting with:

- AES in ECB, CBC, CTR with all three key sizes (128, 192, 256 bits)
- For other modes, CCM, GCM,CMAC, KEY WRAP run using software algorithm implementation

For AES library settings, refer to *Section 10: STM32 encryption library settings*.

For AES library performances and memory requirements, refer to *Section 11: Cryptographic library performance and memory requirements*.

4.2 AES library functions (ECB, CBC and CTR)

Table 20 describes the AES functions of the encryption library.

Table 20. AES algorithm functions (AAA = ECB, CBC or CTR)

Function name	Description
AES_AAA_Encrypt_Init	Loads the key and ivec, performs key schedule
AES_AAA_Encrypt_Append	Launches cryptographic operation, can be called several times
AES_AAA_Encrypt_Finish	AES encryption finalization of AAA mode
AES_AAA_Decrypt_Init	Loads the key and ivec, eventually performs key schedule
AES_AAA_Decrypt_Append	Launches cryptographic operation, can be called several times
AES_AAA_Decrypt_Finish	AES decryption finalization of AAA mode

AAA represents the mode of operations of the AES algorithm. The following modes of operation can be used for AES algorithm:

- ECB
- CBC
- CTR

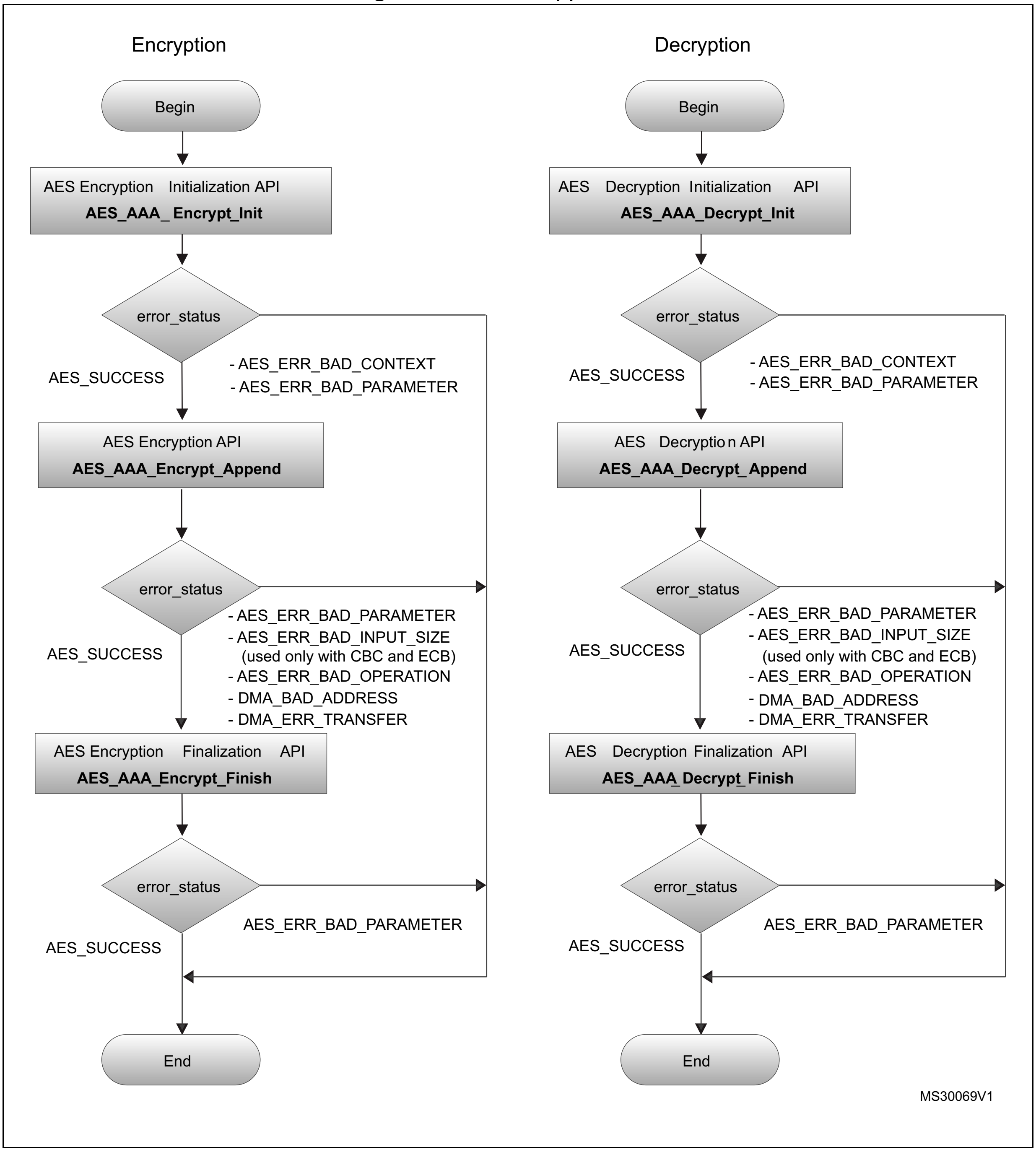
Figure 7 describes the AES_AAA algorithm.

For example, if you want to use ECB mode for an AES algorithm, you can use the following functions:

Table 21. AES ECB algorithm functions

Function name	Description
`AES_ECB_Encrypt_Init`	Loads the key and ivec, performs key schedule
`AES_ECB_Encrypt_Append`	Launches cryptographic operation, can be called several times
`AES_ECB_Encrypt_Finish`	Possible final output
`AES_ECB_Decrypt_Init`	Loads the key and ivec, performs key schedule, init hw, and so on.
`AES_ECB_Decrypt_Append`	Launches cryptographic operation, can be called several times
`AES_ECB_Decrypt_Finish`	Possible final output

Figure 7. AES AAA(*) flowchart



4.2.1 AES_AAA_Encrypt_Init function

Table 22. AES_AAA_Encrypt_Init

Function name	AES_AAA_Encrypt_Init[1]

Prototype	`int32_t AES_AAA_Encrypt_Init(`
	`AESAAActx_stt *P_pAESAAActx,`
	`const uint8_t *P_pKey,`
	`const uint8_t *P_pIv)`

Behavior	Initialization for AES Encryption in AAA Mode
Parameter	– [in, out] *P_pAESAAActx: AES AAA context – [in] *P_pKey: Buffer with the Key. – [int] *P_pIv: Buffer with the IV(Can be NULL since no IV is required in ECB).
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values. See note.

1. AAA is ECB, CBC or CTR.

Note: *1. P_pAESCTRctx.mKeySize (see AESCTRctx_stt) must be set with the size of the key prior to calling this function. Instead of the size of the key, you can also use the following predefined values:*
- CRL_AES128_KEY
- CRL_AES192_KEY
- CRL_AES256_KEY

2. P_pAESCTRctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for detail.

3. P_pAESCTRctx.mIvSize must be set with the size of the IV (default CRL_AES_BLOCK) prior to calling this function.

AESAAActx_stt data structure

Structure type for public key.

Table 23. AESAAActx_stt data structure

Field name	Description
`uint32_t mContextId`	Unique ID of this context. **Not used** in current version
`SKflags_et mFlags`	32 bit mflags, performs keyschedule, see *SKflags_et mFlags*
`const uint8_t * pmKey`	Pointer to original Key buffer
`const uint8_t * pmIv`	Pointer to original Initialization Vector buffer
`int32_t mIvSize`	Size of the Initialization Vector in bytes
`uint32_t amIv[4]`	Temporary result/IV
`uint32_t mKeySize`	Key length in bytes
`uint32_t amExpKey [CRL_AES_MAX_EXPKEY_SIZE]`	Expanded DES key

4.2.2 AES_AAA_Encrypt_Append function

Table 24. AES_AAA_Encrypt_Append

Function name	AES_AAA_Encrypt_Append(1)

Prototype	`int32_t AES_AAA_Encrypt_Append (`
	`AESAAActx_stt * P_pAESAAActx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize)`

Behavior	AES Encryption in AAA Mode
Parameter	– [in] * P_pAESAAActx: AES AAA, already initialized, context – [in] * P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data, expressed in bytes – [out] * P_pOutputBuffer: Output buffer – [out] * P_pOutputSize: Pointer to integer containing size of written output data, expressed in bytes
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer. – AES_ERR_BAD_INPUT_SIZE:(Only with CBC and ECB.) Size of input is less than CRL_AES_BLOCK(CBC) or is not a multiple of CRL_AES_BLOCK(ECB). – AES_ERR_BAD_OPERATION: Append not allowed. – DMA_BAD_ADDRESS: Input or output buffer addresses are not word aligned. – DMA_ERR_TRANSFER: Error occurred in the DMA transfer.

1. AAA is ECB, CBC or CTR.

Note: This function can be called multiple times, provided that P_inputSize is a multiple of 16.
- In CBC mode for a call where P_inputSize is greater than 16 and not multiple of 16, Ciphertext Stealing will be activated. See CBC-CS2 of <"SP 800-38 A - Addendum"> NIST SP 800-38A Addendum.
- In CTR mode, a single, final, call with P_inputSize not multiple of 16 is allowed.

4.2.3 AES_AAA_Encrypt_Finish function

Table 25. AES_AAA_Encrypt_Finish

Function name	AES_AAA_Encrypt_Finish(1)

Prototype	`int32_t AES_AAA_Encrypt_Finish (`
	`AESAAActx_stt * P_pAESAAActx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize)`

Behavior	AES Finalization of AAA mode
Parameter	– [in,out] * P_pAESAAActx: AES AAA, already initialized, context – [out] * P_pOutputBuffer: Output buffer – [out] * P_pOutputSize: Pointer to integer containing size of written output data, in bytes
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer.

1. AAA is ECB, CBC or CTR.

Note: This function won't write output data, thus it can be skipped. It is kept for API compatibility.

4.2.4 AES_AAA_Decrypt_Init function

Table 26. AES_AAA_Decrypt_Init

Function name	**AES_AAA_Decrypt_Init(1)**

Prototype	`int32_t AES_AAA_Decrypt_Init (`
	`AESAAActx_stt * P_pAESAAActx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pIv )`

Behavior	Initialization for AES Decryption in AAA Mode
Parameter	– [in,out] *P_pAESAAActx: AES AAA context. – [in] *P_pKey: Buffer with the Key. – [in] *P_pIv: Buffer with the IV.
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values, see note.

1. AAA is ECB, CBC or CTR.

Note: *1. P_pAESAAActx.mKeySize (see AESAAActx_stt) must be set before calling this function with the size of the key, or with the following predefined values:*
- CRL_AES128_KEY
- CRL_AES192_KEY
- CRL_AES256_KEY

2. P_pAESAAActx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.

3. P_pAESAAActx.mIvSize(used with CBC and CTR modes) must be set with the size of the IV (default CRL_AES_BLOCK) prior to calling this function

4. In ECB the IV is not used, so the value of P_pIv is not checked or used

4.2.5 AES_AAA_Decrypt_Append function

Table 27. AES_AAA_Decrypt_Append

Function name	AES_AAA_Decrypt_Append(1)

Prototype	`int32_t AES_AAA_Decrypt_Append (`
	`AESAAActx_stt * P_pAESAAActx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES Decryption in AAA Mode
Parameter	– [in] *P_pAESAAActx: AES AAA context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Size of written output data in bytes
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_OPERATION: Append not allowed – AES_ERR_BAD_INPUT_SIZE: P_inputSize < 16(in CBC mode) or is not a multiple of CRL_AES_BLOCK(in ECB mode) – DMA_BAD_ADDRESS: Input or output buffer addresses are not word aligned – DMA_ERR_TRANSFER: Error occurred in the DMA transfer

1. AAA is ECB, CBC or CTR.

Note: *1. This function can be called multiple times, provided that P_inputSize is a multiple of 16*

2. In CBC mode and in case of a call where P_inputSize is greater than 16 and not multiple of 16, Ciphertext Stealing will be activated. See CBC-CS2 of <"SP 800-38 A - Addendum"> NIST SP 800-38A Addendum.

3. IN CTR mode, a single, final, call with P_inputSize not multiple of 16 is allowed.

4. In CTR mode: This is a wrapper for AES_CTR_Encrypt_Append as the Counter Mode is equal in encryption and decryption.

4.2.6 AES_AAA_Decrypt_Finish function

Table 28. AES_AAA_Decrypt_Finish

Function name	**AES_AAA_Decrypt_Finish**[1]

Prototype	`int32_t AES_AAA_Decrypt_Finish (`
	`AESAAActx_stt * P_pAESAAActx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES Decryption Finalization of AAA Mode
Parameter	– [in,out] *P_pAESAAActx: AES AAA context – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer

1. AAA is ECB, CBC or CTR.

Note: 1. In CTR mode: This is a wrapper for AES_CTR_Encrypt_Final as the Counter Mode is equal in encryption and decryption

2. This function won't write output data, thus it can be skipped. It is kept for API compatibility

4.3 AES GCM library functions

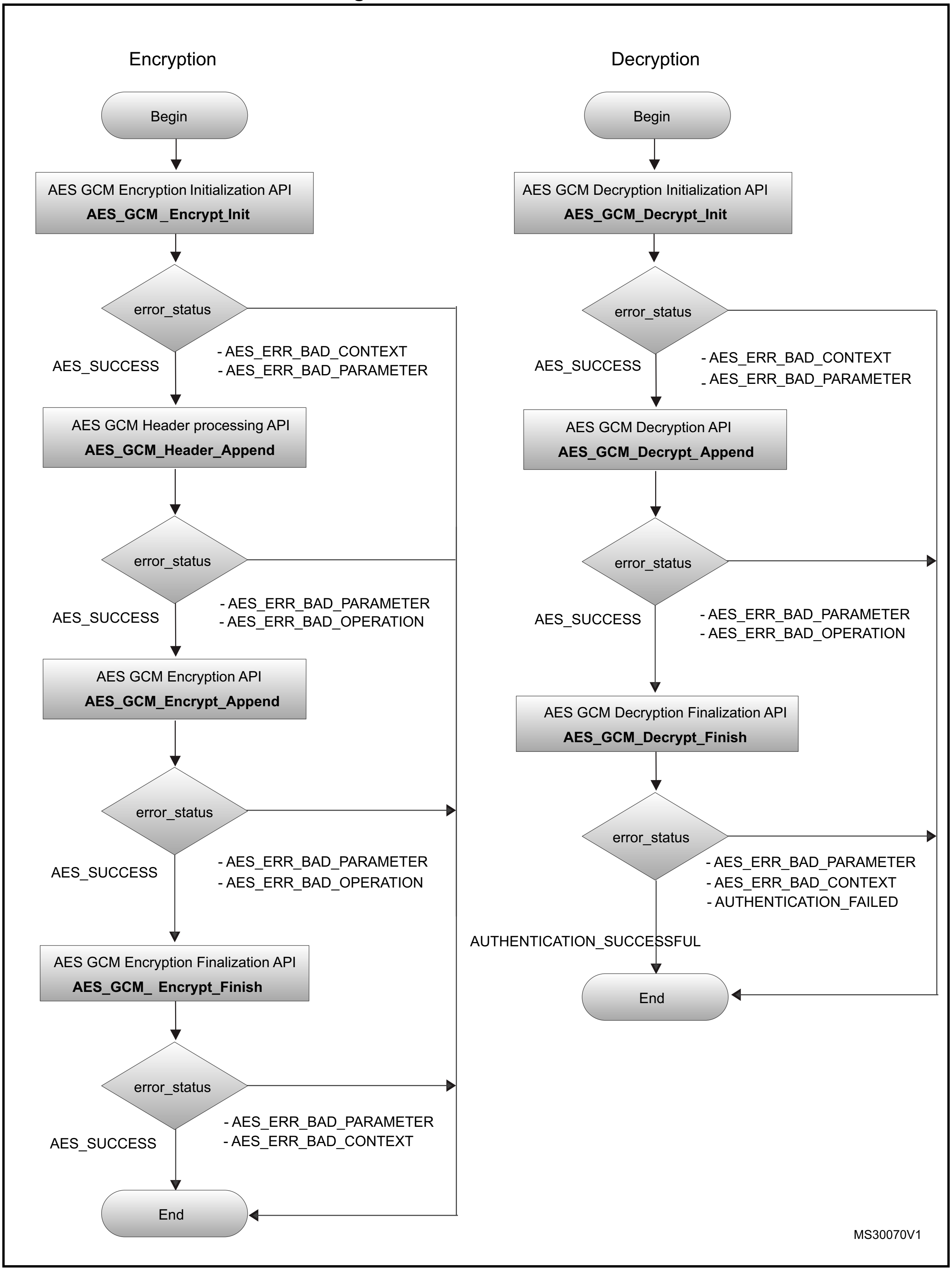
Table 29 describes the AES GCM library.

Table 29. AES GCM algorithm functions

Function name	Description
AES_GCM_Encrypt_Init	Initialization for AES GCM encryption
AES_GCM_Header_Append	Header processing function
AES_GCM_Encrypt_Append	AES GCM encryption function
AES_GCM_Encrypt_Finish	AES GCM finalization during encryption, this will create the Authentication TAG
AES_GCM_Decrypt_Init	Initialization for AES GCM decryption
AES_GCM_Decrypt_Append	AES GCM decryption function
AES_GCM_Decrypt_Finish	AES GCM finalization during decryption, the authentication TAG will be checked

The following flowchart describes the AES_GCM algorithm.

Figure 8. AES GCM flowchart



4.3.1 AES_GCM_Encrypt_Init function

Table 30. AES_GCM_Encrypt_Init

Function name	AES_GCM_Encrypt_Init

Prototype	`int32_t AES_GCM_Encrypt_Init (`
	`AESGCMctx_stt * P_pAESGCMctx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pIv )`

Behavior	Initialization for AES GCM encryption
Parameter	– [in, out] *P_pAESGCMctx: AES GCM context – [in] *P_pKey: Buffer with the Key – [in] *P_pIv: Buffer with the IV
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values, see note

Note: *1. P_pAESGCMctx.mKeySize (see AESGCMctx_stt) must be set with the size of the key prior to calling this function.Otherwise the following predefined values can be used:*

- CRL_AES128_KEY

- CRL_AES192_KEY

- CRL_AES256_KEY

2. P_pAESGCMctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.

3. P_pAESGCMctx.mIvSize must be set with the size of the IV (12 is the only supported value) prior to calling this function.

4. P_pAESGCMctx.mTagSize must be set with the size of authentication TAG that will be generated by the AES_GCM_Encrypt_Finish.

5. If hardware support is enabled, DMA will not be used even if E_SK_USE_DMA is set inside P_pAESGCMctx->mFlags, as GCM is implemented with an interleaved operation and the AES engine is used one block at a time.

6. Following recommendation by NIST expressed in section 5.2.1.1 of NIST SP 800-38D, this implementation supports only IV whose size is of 96 bits.

AESGCMctx_stt data structure

Structure used to store the expanded key and, eventually, precomputed tables, according to the defined value of CRL_GFMUL in the config.h file.

Table 31. AESGCMctx_stt data structure

Field name	Description
`uint32_t mContextId`	Unique ID of this AES-GCM context. **Not used** in current implementation.
`SKflags_et mFlags`	32 bit mFlags, used to perform keyschedule, see *SKflags_et mFlags*
`const uint8_t * pmKey`	Pointer to original key buffer.
`const uint8_t * pmIv`	Pointer to original initialization vector buffer.
`int32_t mIvSize`	Size of the initialization vector in bytes. This must be set by the caller prior to calling Init.
`uint32_t amIv[4]`	This is the current IV value.
`int32_t mKeySize`	AES key length in bytes. Must be set by the caller prior to calling Init.
`const uint8_t * pmTag`	Pointer to Authentication TAG. Must be set in decryption, and this TAG will be verified.
`int32_t mTagSize`	Size of the Tag to return. Must be set by the caller prior to calling Init.
`int32_t mAADsize`	Additional authenticated data size. For internal use.
`int32_t mPayloadSize`	Payload size. For internal use.
`poly_t mPartialAuth`	Partial authentication value. For internal use. where `poly_t`: typedef uint32_t poly_t[4]: Definition of the way a polynomial of maximum degree 127 is represented.
`uint32_t amExpKey [CRL_AES_MAX_EXPKEY_SIZE]`	AES Expanded key. For internal use.
`table8x16_t mPrecomputedValues`	(CRL_GFMUL==2) Precomputation of polynomial according to Shoup's 8-bit table (requires 4096 bytes of key-dependent data and 512 bytes of constant data). For internal use. where `table8x16_t`: typedef poly_t table8x16_t[8][16]: Definition of the type used for the precomputed table

4.3.2 AES_GCM_Header_Append function

Table 32. AES_GCM_Header_Append

Function name	AES_GCM_Header_Append

Prototype	`int32_t AES_GCM_Header_Append (`
	`AESGCMctx_stt * P_pAESGCMctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize )`

Behavior	AES GCM Header processing function
Parameter	– [in,out] *P_pAESGCMctx: AES GCM, already initialized, context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data, expressed in bytes
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_OPERATION Append not allowed

4.3.3 AES_GCM_Encrypt_Append function

Table 33. AES_GCM_Encrypt_Append

Function name	AES_GCM_Encrypt_Append(1)

Prototype	`int32_t AES_GCM_Encrypt_Append (`
	`AESGCMctx_stt * P_pAESGCMctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES GCM Encryption function
Parameter	– [in,out] *P_pAESGCMctx: AES GCM, already initialized, context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data, expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_OPERATION: Append not allowed

1. This function can be called multiple times, provided that P_inputSize is a multiple of 16. A single, final, call with P_inputSize not multiple of 16 is allowed.

4.3.4 AES_GCM_Encrypt_Finish function

Table 34. AES_GCM_Encrypt_Finish

Function name	AES_GCM_Encrypt_Finish(1)

Prototype	`int32_t AES_GCM_Encrypt_Finish (`
	`AESGCMctx_stt * P_pAESGCMctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES GCM Finalization during encryption, this will create the Authentication TAG
Parameter	– [in,out] *P_pAESGCMctx: AES GCM, already initialized, context – [out] *P_pOutputBuffer: Output Authentication TAG – [out] *P_pOutputSize: Size of returned TAG
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values. See note

1. This function requires P_pAESGCMctx mTagSize to contain a valid value between 1 and 16.

4.3.5 AES_GCM_Decrypt_Init function

Table 35. AES_GCM_Decrypt_Init

Function name	AES_GCM_Decrypt_Init

Prototype	`int32_t AES_GCM_Decrypt_Init (`
	`AESGCMctx_stt * P_pAESGCMctx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pIv )`

Behavior	Initialization for AES GCM Decryption
Parameter	– [in,out] *P_pAESGCMctx: AES GCM context – [in] *P_pKey: Buffer with the Key – [in] *P_pIv: Buffer with the IV
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values.

Note: *1. P_pAESGCMctx.mKeySize (see AESGCMctx_stt) must be set with the size of the key prior to calling this function.Otherwise the following predefined values can be used:*
- CRL_AES128_KEY
- CRL_AES192_KEY
- CRL_AES256_KEY

2. P_pAESGCMctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.

3. P_pAESGCMctx.mIvSize must be set with the size of the IV (12 is the only supported value) prior to calling this function.

4. P_pAESGCMctx.mTagSize must be set with the size of authentication TAG that will be generated by the AES_GCM_Encrypt_Finish.

5. If hardware support is enabled, DMA will not be used even if E_SK_USE_DMA is set inside P_pAESGCMctx->mFlags, as GCM is implemented with an interleaved operation and the AES engine is used one block at a time.

6. Following recommendation by NIST expressed in section 5.2.1.1 of NIST SP 800-38D, this implementation supports only IV whose size is of 96 bits.

4.3.6 AES_GCM_Decrypt_Append function

Table 36. AES_GCM_Decrypt_Append

Function name	AES_GCM_Decrypt_Append

Prototype	`int32_t AES_GCM_Decrypt_Append (`
	`AESGCMctx_stt * P_pAESGCMctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES GCM Decryption function
Parameter	– [in,out] *P_pAESGCMctx: AES GCM context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data in uint8_t (octets) – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Size of written output data in uint8_t
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_OPERATION: Append not allowed

4.3.7 AES_GCM_Decrypt_Finish function

Table 37. AES_GCM_Decrypt_Finish

Function name	AES_GCM_Decrypt_Finish

Prototype	`int32_t AES_GCM_Decrypt_Finish (`
	`AESGCMctx_stt * P_pAESGCMctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize)`

Behavior	AES GCM Finalization during decryption, the authentication TAG will be checked
Parameter	– [in,out] *P_pAESGCMctx: AES GCM, already initialized, context – [out] *P_pOutputBuffer: Kept for API compatibility but won't be used, should be NULL – [out] *P_pOutputSize: Kept for API compatibility, must be provided but will be set to zero
Return value	– AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values – AUTHENTICATION_SUCCESSFUL: if the TAG is verified – AUTHENTICATION_FAILED: if the TAG is **not** verified

Note: This function requires:
- P_pAESGCMctx->pmTag to be set to a valid pointer to the tag to be checked.
- P_pAESGCMctx->mTagSize to contain a valid value between 1 and 16.

4.4 AES KeyWrap library functions

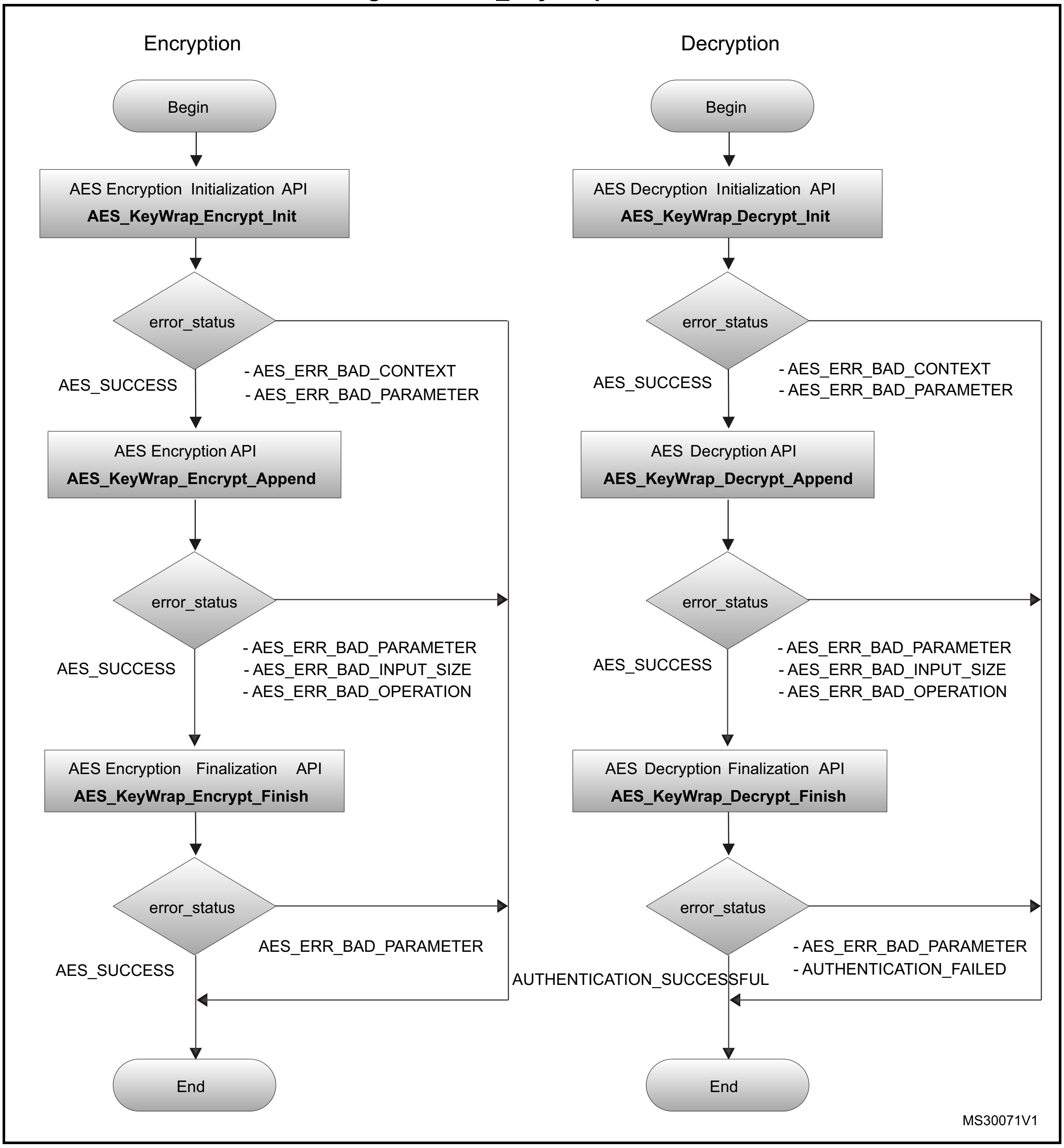
Table 38 describes the AES KeyWrap library.

Table 38. AES KeyWrap algorithm functions

Function name	Description
AES_KeyWrap_Encrypt_Init	Initialization for AES KeyWrap Encryption
AES_KeyWrap_Encrypt_Append	AES KeyWrap Wrapping function
AES_KeyWrap_Encrypt_Finish	AES KeyWrap Finalization
AES_KeyWrap_Decrypt_Init	Initialization for AES KeyWrap Decryption
AES_KeyWrap_Decrypt_Append	AES KeyWrap UnWrapping function
AES_KeyWrap_Decrypt_Finish	AES KeyWrap Finalization during Decryption, the authentication will be checked

The next flowchart describes the AES_KeyWrap algorithm

Figure 9. AES_KeyWrap flowchart



4.4.1 AES_KeyWrap_Encrypt_Init function

Table 39. AES_KeyWrap_Encrypt_Init

Function name	AES_KeyWrap_Encrypt_Init
Prototype	`int32_t AES_KeyWrap_Encrypt_Init ( AESKWctx_stt * P_pAESKWctx, const uint8_t * P_pKey, const uint8_t * P_pIv )`
Behavior	Initialization for AES KeyWrap Encryption
Parameter	– [in,out] *P_pAESKWctx: AES Key Wrap context – [in] *P_pKey: Buffer with the Key (KEK) – [in] *P_pIv: Buffer with the 64 bits IV
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values.

Note: 1. P_pAESKWctx.mKeySize (see AESKWctx_stt) must be set with the size of the key prior to calling this function. Otherwise the following predefined values can be used:
- CRL_AES128KEY
- CRL_AES192_KEY
- CRL_AES256_KEY

2. P_pAESKWctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.

3. If hardware support is enabled, DMA will not be used even if E_SK_USE_DMA is set inside P_pAESKWctx->mFlags, as CCM is implemented with an interleaved operation and the AES engine is used one block at a time.

4. NIST defines the IV equal to 0xA6A6A6A6A6A6A6A6. In this implementation is a required input and can assume any value but its size is limited to 8 byte.

AESKWctx_stt data structure

The AESKWctx_stt data structure is aliased to the *AESAAActx_stt data structure*.

4.4.2 AES_KeyWrap_Encrypt_Append function

Table 40. AES_KeyWrap_Encrypt_Append

Function name	**AES_KeyWrap_Encrypt_Append**(1)

Prototype	`int32_t AES_KeyWrap_Encrypt_Append (`
	`AESKWctx_stt * P_pAESKWctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES KeyWrap Wrapping function
Parameter	– [in,out] *P_pAESKWctx: AES KeyWrap, already initialized, context – [in] *P_pInputBuffer: Input buffer, containing the Key to be wrapped – [in] P_inputSize: Size of input data, expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_OPERATION: Append not allowed – AES_ERR_BAD_INPUT_SIZE: P_inputSize must be non-zero multiple of 64 bits

1. This function can be called only once, passing in it the whole Key to be Wrapped

Note: *1. P_inputSize must be a non-zero multiple of 64 bits, up to a maximum of 256 or AES_ERR_BAD_INPUT_SIZE is returned.*

2. P_pOutputBuffer must be at least 8 bytes longer than P_pInputBuffer.

4.4.3 AES_KeyWrap_Encrypt_Finish function

Table 41. AES_KeyWrap_Encrypt_Finish

Function name	**AES_KeyWrap_Encrypt_Finish**(1)

Prototype	`int32_t AES_KeyWrap_Encrypt_Finish (`
	`AESKWctx_stt * P_pAESKWctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES KeyWrap Finalization
Parameter	– [in,out] *P_pAESKWctx: AES KeyWrap, already initialized, context – [out] *P_pOutputBuffer: Output buffer (won't be used) – [out] *P_pOutputSize: Size of written output data (It will be zero)
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer

1. This function won't write output data, thus it can be skipped. It is kept for API compatibility.

4.4.4 AES_KeyWrap_Decrypt_Init function

Table 42. AES_KeyWrap_Decrypt_Init

Function name	AES_KeyWrap_Decrypt_Init

Prototype	`int32_t AES_KeyWrap_Decrypt_Init (`
	`AESKWctx_stt * P_pAESKWctx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pIv )`

Behavior	Initialization for AES KeyWrap Decryption
Parameter	– [in,out] *P_pAESKWctx: AES Key Wrap context – [in] *P_pKey: Buffer with the Key (KEK) – [in] *P_pIv: Buffer with the 64 bits IV
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values.

Note: 1. P_pAESKWctx.mKeySize (see AESKWctx_stt) must be set with the size of the key prior to calling this function. Otherwise the following predefined values can be used:
- CRL_AES128KEY
- CRL_AES192_KEY
- CRL_AES256_KEY

2. P_pAESKWctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.

3. If hardware support is enabled, DMA will not be used even if E_SK_USE_DMA is set inside P_pAESKWctx->mFlags, as CCM is implemented with an interleaved operation and the AES engine is used one block at a time.

4. NIST defines the IV equal to 0xA6A6A6A6A6A6A6A6. In this implementation is a required input and can assume any value but its size is limited to 8 bytes.

4.4.5 AES_KeyWrap_Decrypt_Append function

Table 43. AES_KeyWrap_Decrypt_Append

Function name	**AES_KeyWrap_Decrypt_Append**

Prototype	`int32_t AES_KeyWrap_Decrypt_Append (`
	`AESKWctx_stt * P_pAESKWctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`int8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES KeyWrap UnWrapping function
Parameter	– [in,out] *P_pAESKWctx: AES KeyWrap context – [in] *P_pInputBuffer: Input buffer, containing the Key to be unwrapped – [in] P_inputSize: Size of input data in uint8_t (octets) – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Size of written output data in uint8_t
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_OPERATION: Append not allowed – AES_ERR_BAD_INPUT_SIZE: P_inputSize must be a non-zero multiple of 64 bits and at maximum 264

Note: 1. This function can be called only once, passing in it the whole Wrapped Key.

2. P_inputSize must be a non-zero multiple of 64 bits and be a maximum of 264 or AES_ERR_BAD_INPUT_SIZE is returned.

3. P_pOutputBuffer must be at least 8 bytes smaller than P_pInputBuffer.

4.4.6 AES_KeyWrap_Decrypt_Finish function

Table 44. AES_KeyWrap_Decrypt_Finish

Function name	**AES_KeyWrap_Decrypt_Finish**

Prototype	`int32_t AES_KeyWrap_Decrypt_Finish (`
	`AESKWctx_stt * P_pAESKWctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES KeyWrap Finalization during Decryption, the authentication will be checked
Parameter	– [in,out] *P_pAESKWctx: AES KeyWrap context – [out] *P_pOutputBuffer: Won't be used – [out] *P_pOutputSize: Will contain zero
Return value	Result of Authentication or error codes: – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AUTHENTICATION_SUCCESSFUL: Unwrapped key produced by AES_KeyWrap_Decrypt_Append is valid – AUTHENTICATION_FAILED: Unwrapped key produced by AES_KeyWrap_Decrypt_Append is not valid.

4.5 AES CMAC library functions

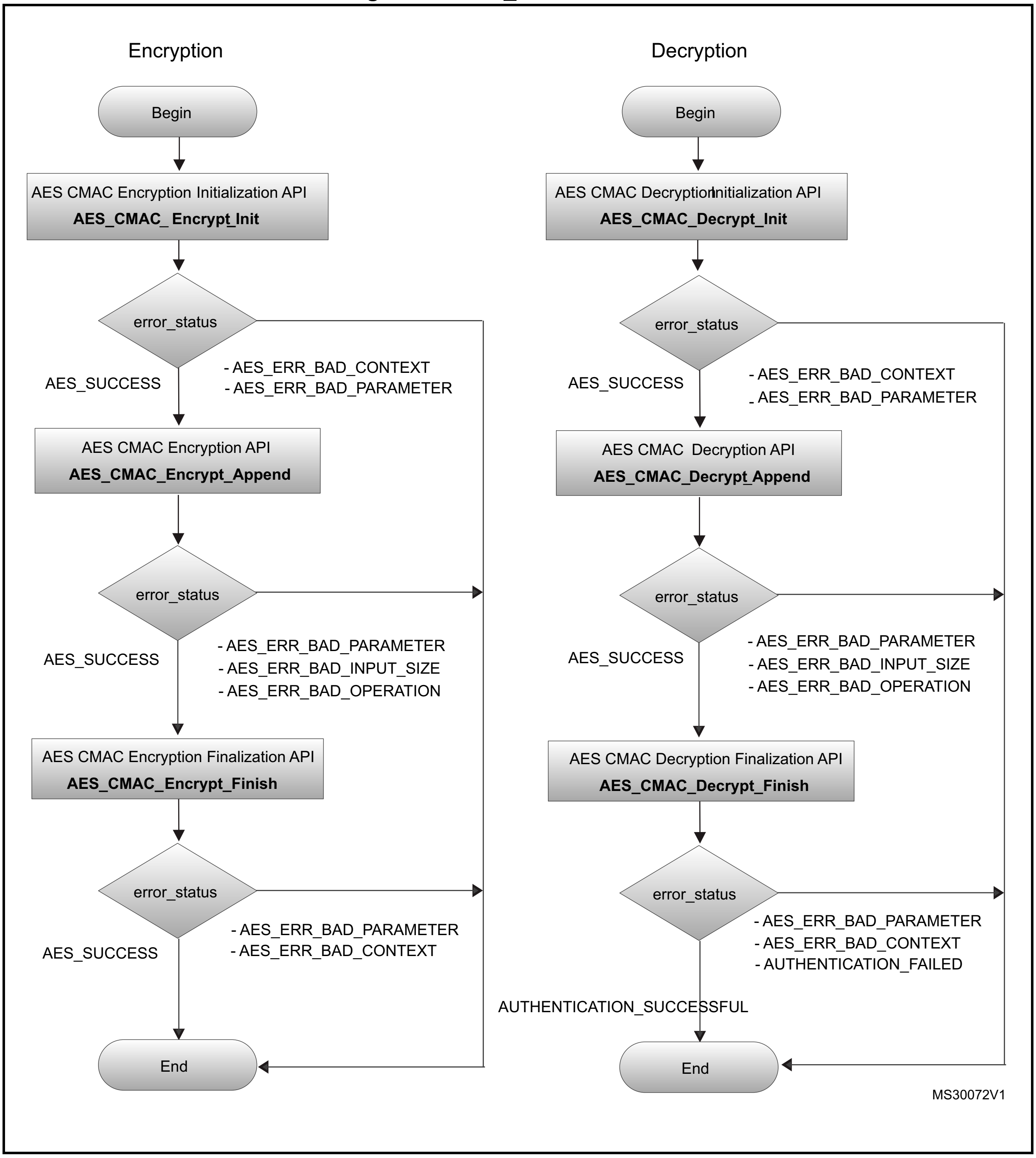
Table 45 describes the AES CMAC library.

Table 45. AES CMAC algorithm functions

Function name	Description
AES_CMAC_Encrypt_Init	Initialization for AES-CMAC for Authentication TAG Generation
AES_CMAC_Encrypt_Append	AES Encryption in CMAC Mode
AES_CMAC_Encrypt_Finish	AES Finalization of CMAC Mode
AES_CMAC_Decrypt_Init	Initialization for AES-CMAC for Authentication TAG Verification
AES_CMAC_Decrypt_Append	AES-c Data Processing
AES_CMAC_Decrypt_Finish	AES Finalization of CMAC Mode

The next flowchart describes the AES_CMAC algorithm.

Figure 10. AES_CMAC flowchart



4.5.1 AES_CMAC_Encrypt_Init function

Table 46. AES_CMAC_Encrypt_Init

Function name	AES_CMAC_Encrypt_Init
Prototype	`int32_t AES_CMAC_Encrypt_Init ( AESCMACctx_stt * P_pAESCMACctx )`
Behavior	Initialization for AES-CMAC for Authentication TAG Generation
Parameter	– [in,out] *P_pAESCMACctx: AES CMAC context
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values

Note: *1. P_pAESCMACctx.pmKey (see AESCMACctx_stt) must be set with a pointer to the AES key before calling this function.*

2. P_pAESCMACctx.mKeySize must be set with the size of the key prior to calling this function. Otherwise the following predefined values can be used:
- CRL_AES128_KEY
- CRL_AES192_KEY
- CRL_AES256_KEY

3. P_pAESCMACctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.

4. P_pAESCMACctx.mTagSize must be set with the size of authentication TAG that will be generated by the AES_CMAC_Encrypt_Finish.

5. If hardware support is enabled, DMA will not be used even if E_SK_USE_DMA is set inside P_pAESCMACctx->mFlags, as CCM is implemented with an interleaved operation and the AES engine is used one block at a time.

AESCMACctx_stt data structure

Table 47. AESCMACctx_stt data structure

Field name	Description
`uint32_t mContextId`	Unique ID of this AES-GCM Context. **Not used** in this implementation.
`SKflags_et mFlags`	32 bit mFlags, used to perform keyschedule, see *SKflags_et mFlags* choose between hw/sw/hw+dma and future use
`const uint8_t * pmKey`	Pointer to original Key buffer
`const uint8_t * pmIv`	Pointer to original initialization vector buffer
`int32_t mIvSize`	Initialization vector size (bytes) Must be set by caller prior to calling Init
`uint32_t amIv[4]`	This is the current IV value.
`int32_t mKeySize`	AES Key length in bytes. Must be set by the caller prior to calling Init
`uint32_t amExpKey[CRL_AES_MAX_EXPKEY_SIZE]`	AES Key length in bytes. This must be set by the caller prior to calling Init
`const uint8_t * pmTag`	Size of the Tag to return. Must be set by the caller prior to calling Init
`int32_t mTagSize`	Size of the Tag to return. Must be set by the caller prior to calling Init

4.5.2 AES_CMAC_Encrypt_Append function

Table 48. AES_CMAC_Encrypt_Append

Function name	AES_CMAC_Encrypt_Append(1)

Prototype	`int32_t AES_CMAC_Encrypt_Append (`
	`AESCMACctx_stt * P_pAESCMACctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize )`

Behavior	AES Encryption in CMAC Mode
Parameter	– [in,out] *P_pAESCMACctx AES CMAC, already initialized, context – [in] *P_pInputBuffer Input buffer – [in] P_inputSize Size of input data in uint8_t (octets)
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_INPUT_SIZE: – P_inputSize < 0 | (P_inputSize % 16 != 0 && P_pAESCMACctx mFlags & E_SK_FINAL_APPEND) != E_SK_FINAL_APPEND) – AES_ERR_BAD_OPERATION: Append not allowed

1. This function can be called multiple times with P_inputSize multiple of 16 bytes. The last call allows any positive value for P_inputSize but flag E_SK_FINAL_APPEND must be set inside P_pAESCMACctx mFlags (i.e. with a simple P_pAESCMACctx->mFlags |= E_SK_FINAL_APPEND).

4.5.3 AES_CMAC_Encrypt_Finish function

Table 49. AES_CMAC_Encrypt_Finish

Function name	AES_CMAC_Encrypt_Finish(1)

Prototype	`int32_t AES_CMAC_Encrypt_Finish (`
	`AESCMACctx_stt * P_pAESCMACctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES Finalization of CMAC Mode
Parameter	– [in,out] *P_pAESCMACctx AES CMAC, already initialized, context – [out] *P_pOutputBuffer Output buffer – [out] *P_pOutputSize Size of written output data in uint8_t
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values, see note.

1. This function requires P_pAESCMACctx mTagSize to contain valid value between 1 and 16.

4.5.4 AES_CMAC_Decrypt_Init function

Table 50. AES_CMAC_Decrypt_Init

Function name	AES_CMAC_Decrypt_Init
Prototype	`int32_t AES_CMAC_Decrypt_Init ( AESCMACctx_stt * P_pAESCMACctx )`
Behavior	Initialization for AES-CMAC for Authentication TAG Verification
Parameter	– [in,out] *P_pAESCMACctx AES CMAC context
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT Context not initialized with valid values, see the note below

Note: *1. P_pAESCMACctx.pmKey (see AESCMACctx_stt) must be set with a pointer to the AES key before calling this function.*

2. P_pAESCMACctx.mKeySize must be set with the size of the key prior to calling this function. Otherwise the following predefined values can be used:
- CRL_AES128_KEY
- CRL_AES192_KEY
- CRL_AES256_KEY

3. P_pAESCMACctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.

4. P_pAESCMACctx.pmTag must be set with a pointer to the authentication TAG that will be checked during AES_CMAC_Decrypt_Finish.

5. P_pAESCMACctx.mTagSize must be set with the size of authentication TAG that will be generated by the AES_CMAC_Encrypt_Finish.

6. If hardware support is enabled, DMA will not be used even if E_SK_USE_DMA is set inside P_pAESCMACctx->mFlags, as CCM is implemented with an interleaved operation and the AES engine is used one block at a time.

4.5.5 AES_CMAC_Decrypt_Append function

Table 51. AES_CMAC_Decrypt_Append

Function name	AES_CMAC_Decrypt_Append

Prototype	`int32_t AES_CMAC_Decrypt_Append (`
	`AESCMACctx_stt * P_pAESCMACctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize )`

Behavior	AES-CMAC Data Processing
Parameter	– [in,out] *P_pAESCMACctx AES CMAC, already initialized, context – [in] *P_pInputBuffer Input buffer – [in] P_inputSize Size of input data in uint8_t (octets)
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER At least one parameter is a NULL pointer – AES_ERR_BAD_INPUT_SIZE:P_inputSize <= 0 | (P_inputSize % 16 != 0 && P_pAESCMACctx->mFlags & E_SK_FINAL_APPEND) != E_SK_FINAL_APPEND) – AES_ERR_BAD_OPERATION: Append not allowed

Note: This function can be called multiple times with P_inputSize multiple of 16 bytes. The last call allows any positive value for P_inputSize but flag E_SK_FINAL_APPEND must be set inside P_pAESCMACctx mFlags (i.e. with a simple P_pAESCMACctx->mFlags |= E_SK_FINAL_APPEND).

4.5.6 AES_CMAC_Decrypt_Finish function

Table 52. AES_CMAC_Decrypt_Finish

Function name	AES_CMAC_Decrypt_Finish

Prototype	`int32_t AES_CMAC_Decrypt_Finish (`
	`AESCMACctx_stt * P_pAESCMACctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES Finalization of CMAC Mode
Parameter	– [in,out] *P_pAESCMACctx AES CMAC, already initialized, context – [out] *P_pOutputBuffer Output buffer – [out] *P_pOutputSize Size of written output data in uint8_t
Return value	– AES_ERR_BAD_PARAMETER At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT Context not initialized with valid values, see note. – AUTHENTICATION_SUCCESSFUL if the TAG is verified – AUTHENTICATION_FAILED if the TAG is not verified

Note: This function requires:

- P_pAESGCMctx->pmTag to be set to a valid pointer to the tag to be checked.

- P_pAESCMACctx->mTagSize to contain a valid value between 1 and 16.

4.6 AES CCM library functions

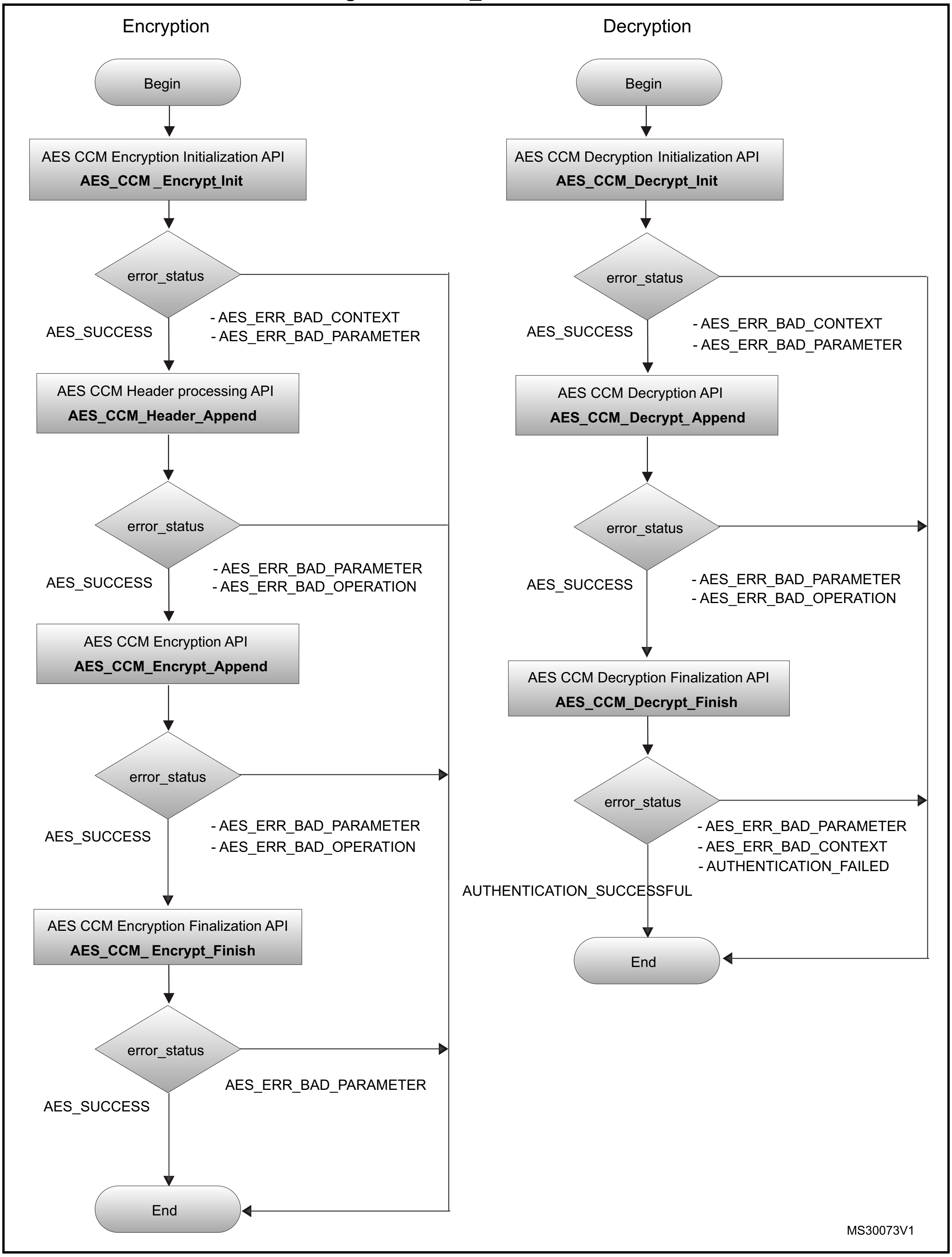
Table 53 describes the AES CCM library.

Table 53. AES CCM algorithm functions

Function name	Description
AES_CMAC_Encrypt_Init	Initialization for AES CCM encryption
AES_CCM_Header_Append	Header Processing Function
AES_CCM_Encrypt_Append	AES CCM encryption function
AES_CCM_Encrypt_Finish	AES CCM Finalization during encryption, this will create the Authentication TAG
AES_CCM_Decrypt_Init	Initialization for AES CCM decryption
AES_CCM_Decrypt_Append	AES CCM decryption function
AES_CCM_Decrypt_Finish	AES CCM Finalization during decryption, the authentication TAG will be checked

The next flowchart describes the AES_CCM algorithm.

Figure 11. AES_CCM flowchart



4.6.1 AES_CCM_Encrypt_Init function

Table 54. AES_CCM_Encrypt_Init

Function name	AES_CCM_Encrypt_Init

Prototype	`int32_t AES_CCM_Encrypt_Init (`
	`AESCCMctx_stt * P_pAESCCMctx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pNonce )`

Behavior	Initialization for AES CCM encryption
Parameter	– [in, out] *P_pAESCCMctx: AES CCM context – [in] *P_pKey: Buffer with the Key – [in] *P_pNonce: Buffer with the Nonce
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values,

Note: *1. P_pAESCCMctx.mKeySize (see \ref AESCCMctx_stt) must be set with the size of the key prior to calling this function. Otherwise the following predefined values can be used:*
- CRL_AES128_KEY
- CRL_AES192_KEY
- CRL_AES256_KEY

2. P_pAESCCMctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.

3. P_pAESCCMctx.mNonceSize must be set with the size of the CCM Nonce. Possible values are {7,8,9,10,11,12,13}.

4. P_pAESCCMctx.mTagSize must be set with the size of authentication TAG that will be generated by the AES_CCM_Encrypt_Finish. Possible values are values are {4,6,8,10,12,14,16}.

5. P_pAESCCMctx.mAssDataSize must be set with the size of the Associated Data (i.e. Header or any data that will be authenticated but not encrypted).

6. P_pAESCCMctx.mPayloadSize must be set with the size of the Payload (i.e. Data that will be authenticated and encrypted).

7. In CCM standard the TAG is appended to the Ciphertext. In this implementation, for API compatibility with GCM, the user must supply a pointer to AES_CCM_Encrypt_Finish that will be used to output the authentication TAG.

8. If hardware support is enabled, DMA will not be used even if E_SK_USE_DMA is set inside P_pAESCCMctx->mFlags, as CCM is implemented with an interleaved operation and the AES engine is used one block at a time.

AESCCMctx_stt data structure

Table 55. AESCCMctx_stt data structure

Field name	Description
`uint32_t mContextId`	Unique ID of this AES-CCM Context. **Not used** in current implementation.
`SKflags_et mFlags`	32 bit mFlags, used to perform keyschedule, see *SKflags_et mFlags*
`const uint8_t * pmKey`	Pointer to original Key buffer.
`const uint8_t * pmNonce`	Pointer to original Nonce buffer
`int32_t mNonceSize`	Size of the Nonce in bytes. This must be set by the caller prior to calling Init. Possible values are {7,8,9,10,11,12,13}
`uint32_t amIvCTR[4]`	This is the current IV value for encryption
`uint32_t amIvCBC[4]`	This is the current IV value for authentication
`int32_t mKeySize`	AES Key length in bytes. This must be set by the caller prior to calling Init.
`const uint8_t * pmTag`	Pointer to Authentication TAG. This value must be set in decryption, and this TAG will be verified.
`int32_t mTagSize`	Size of the Tag to return. This must be set by the caller prior to calling Init. Possible values are values are {4,6,8,10,12,14,16}
`int32_t mAssDataSize`	Size of the associated data to be processed yet. This must be set by the caller prior to calling Init
`int32_t mPayloadSize`	Size of the payload data to be processed yet size. This must be set by the caller prior to calling Init
`uint32_t amExpKey [CRL_AES_MAX_EXPKEY_SIZE]`	AES Expanded key. For internal use
`uint32_t amTmpBuf[CRL_AES_BLOCK/ sizeof(uint32_t)]`	Temp buffer
`int32_t mTmpBufUse`	Number of bytes actually in use

4.6.2 AES_CCM_Header_Append function

Table 56. AES_CCM_Header_Append

Function name	AES_CCM_Header_Append(1)

Prototype	`int32_t AES_CCM_Header_Append (`
	`AESCCMctx_stt * P_pAESCCMctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize )`

Behavior	AES CCM Header processing function
Parameter	– [in,out] *P_pAESCCMctx: AES CCM context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data, expressed in bytes
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER At least one parameter is a NULL pointer – AES_ERR_BAD_OPERATION Append not allowed

1. This function can be called multiple times, provided that P_inputSize is a multiple of 16. A single, final, call with P_inputSize not multiple of 16 is allowed

4.6.3 AES_CCM_Encrypt_Append function

Table 57. AES_CCM_Encrypt_Append

Function name	AES_CCM_Encrypt_Append(1)

Prototype	`int32_t AES_CCM_Encrypt_Append (`
	`AESCCMctx_stt * P_pAESCCMctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES CCM Encryption function
Parameter	– [in,out] *P_pAESCCMctx: AES CCM context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data, expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Size of written output data, expressed in bytes
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_OPERATION: Append not allowed

1. This function can be called multiple times, provided that P_inputSize is a multiple of 16. A single, final, call with P_inputSize not multiple of 16 is allowed.

4.6.4 AES_CCM_Encrypt_Finish function

Table 58. AES_CCM_Encrypt_Finish

Function name	AES_CCM_Encrypt_Finish

Prototype	`int32_t AES_CCM_Encrypt_Finish (`
	`AESCCMctx_stt * P_pAESCCMctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES CCM Finalization during encryption, this will create the Authentication TAG
Parameter	– [in,out] *P_pAESCCMctx: AES CCM, already initialized, context – [out] *P_pOutputBuffer: Output Authentication TAG – [out] *P_pOutputSize: Size of returned TAG
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer

Note: This function requires: P_pAESCCMctx->mTagSize to contain a valid value in the set {4,6,8,10,12,14,16}

4.6.5 AES_CCM_Decrypt_Init function

Table 59. AES_CCM_Decrypt_Init

Function name	AES_CCM_Decrypt_Init

Prototype	`int32_t AES_CCM_Decrypt_Init (`
	`AESCCMctx_stt * P_pAESCCMctx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pNonce )`

Behavior	Initialization for AES CCM Decryption
Parameter	– [in,out] *P_pAESCCMctx: AES CCM context – [in] *P_pKey: Buffer with the Key – [in] *P_pNonce: Buffer with the Nonce
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: Context not initialized with valid values.

Note: *1. P_pAESCCMctx.mKeySize (see AESCCMctx_stt) must be set with the size of the key prior to calling this function. Otherwise the following predefined values can be used:*
- CRL_AES128_KEY
- CRL_AES192_KEY
- CRL_AES256_KEY.

2. P_pAESCCMctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See SKflags_et for details.

3. P_pAESCCMctx.mNonceSize must be set with the size of the CCM Nonce. Possible values are {7,8,9,10,11,12,13}

4. P_pAESCCMctx.pmTag must be set with a pointer to the authentication TAG that will be checked during AES_CCM_Decrypt_Finish

5. P_pAESCCMctx.mTagSize must be set with the size of authentication TAG that will be checked by the AES_CCM_Decrypt_Finish. Possible values are values are {4,6,8,10,12,14,16}

6. P_pAESCCMctx.mAssDataSize must be set with the size of the Associated Data (i.e. Header or any data that will be authenticated but not encrypted)

7. P_pAESCCMctx.mPayloadSize must be set with the size of the Payload (i.e. Data that will be authenticated and encrypted)

8. CCM standard expects the authentication TAG to be passed as part of the ciphertext. In this implementations the tag should be not be passed to AES_CCM_Decrypt_Append. Instead a pointer to the TAG must be set in P_pAESCCMctx.pmTag and this will be checked by AES_CCM_Decrypt_Finish

9. If hardware support is enabled, DMA will not be used even if E_SK_USE_DMA is set inside P_pAESCCMctx->mFlags, as CCM is implemented with an interleaved operation and the AES engine is used one block at a time.

4.6.6 AES_CCM_Decrypt_Append function

Table 60. AES_CCM_Decrypt_Append

Function name	**AES_CCM_Decrypt_Append**(1)

Prototype	`int32_t AES_CCM_Decrypt_Append (`
	`AESCCMctx_stt * P_pAESCCMctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	AES CCM Decryption function
Parameter	– [in,out] *P_pAESCCMctx: AES CCM, already initialized, context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data, expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_OPERATION: Append not allowed

1. This function can be called multiple times, provided that P_inputSize is a multiple of 16. A single, final, call with P_inputSize not multiple of 16 is allowed.

Note: This function shouldn't process the TAG, which is part of the ciphertext according to CCM standard.

4.6.7 AES_CCM_Decrypt_Finish function

Table 61. AES_CCM_Decrypt_Finish

Function name	**AES_CCM_Decrypt_Finish**

Prototype	`int32_t AES_CCM_Decrypt_Finish  (`
	`AESCCMctx_stt *  P_pAESCCMctx,`
	`uint8_t *  P_pOutputBuffer,`
	`int32_t *  P_pOutputSize)`

Behavior	AES CCM Finalization during decryption, the authentication TAG will be checked
Parameter	– [in,out] *P_pAESCCMctx: AES CCM context – [out] *P_pOutputBuffer: Won't be used – [out] *P_pOutputSize: Will contain zero
Return value	– AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – AES_ERR_BAD_CONTEXT: pmTag should be set and mTagSize must be valid – AUTHENTICATION_SUCCESSFUL: if the TAG is verified – AUTHENTICATION_FAILED: if the TAG is **not** verified

Note: This function requires:
- P_pAESCCMctx->pmTag to be set to a valid pointer to the tag to be checked
- P_pAESCCMctx->mTagSize to contain a valid value in the set {4,6,8,10,12,14,16}

4.7 AES CBC enciphering and deciphering example

The following code performs a CBC encryption with AES-128 of 1024 in 4 Append calls.

```
#include "crypto.h"
 int32_t main()
 {
   uint8_t key_128[CRL_AES128_KEY]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
   uint8_t iv[CRL_AES_BLOCK]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
   uint8_t plaintext[1024]={...}
   uint8_t ciphertext[1024];
   /* outSize is for output size, retval is for return value */
   int32_t outSize, retval;
   AESCBCctx_stt AESctx_st; /* The AES context */
/* Initialize Context Flag with default value */
   AESctx_st.mFlags = E_SK_DEFAULT;
/* Set Iv size to 16 */
   AESctx_st.mIvSize=16;
/* Set key size to 16 */
   AESctx_st.mKeySize=CRL_AES128_KEY;
/* call init function */
   retval = AES_CBC_Encrypt_Init(&AESctx_st, key, iv);
   if (retval != AES_SUCCESS)
   {  ... }
/* Loop to perform four calls to AES_CBC_Encrypt_Append, each processing 
256 bytes */
   for (i = 0; i < 1024; i += 256)
   {
/* Encrypt i bytes of plaintext. Put the output data in ciphertext and 
number of  written bytes in outSize */
     retval = AES_CBC_Encrypt_Append(&AESctx_st, plaintext, 256, 
ciphertext, &outSize);
     if (retval != AES_SUCCESS)
     {  ... }
   }
/* Do the finalization call (in CBC it will not return any output)*/
   retval = AES_CBC_Encrypt_Finish(&context_st, ciphertext+outSize, 
&outSize );
   if (retval != AES_SUCCESS)
   {  ... }
 }
```

5 ARC4 algorithm

5.1 Description

The ARC4 (also known as RC4) encryption algorithm was designed by Ronald Rivest of RSA. It is used identically for encryption and decryption as the data stream is simply XORed with the generated key sequence. The algorithm is serial as it requires successive exchanges of state entries based on the key sequence.

The STM32 cryptographic library includes functions required to support ARC4, a module to perform encryption and decryption using the following modes.

This algorithm can run with the STM32F1, STM32L1, STM32F20x, STM32F05x, STM32F40x, STM32F37x and the STM32F30x series using a software algorithm implementation.

For ARC4 library settings, refer to *Section 10: STM32 encryption library settings*.

For ARC4 library performance and memory requirements, refer to *Section 11: Cryptographic library performance and memory requirements*.

5.2 ARC4 library functions

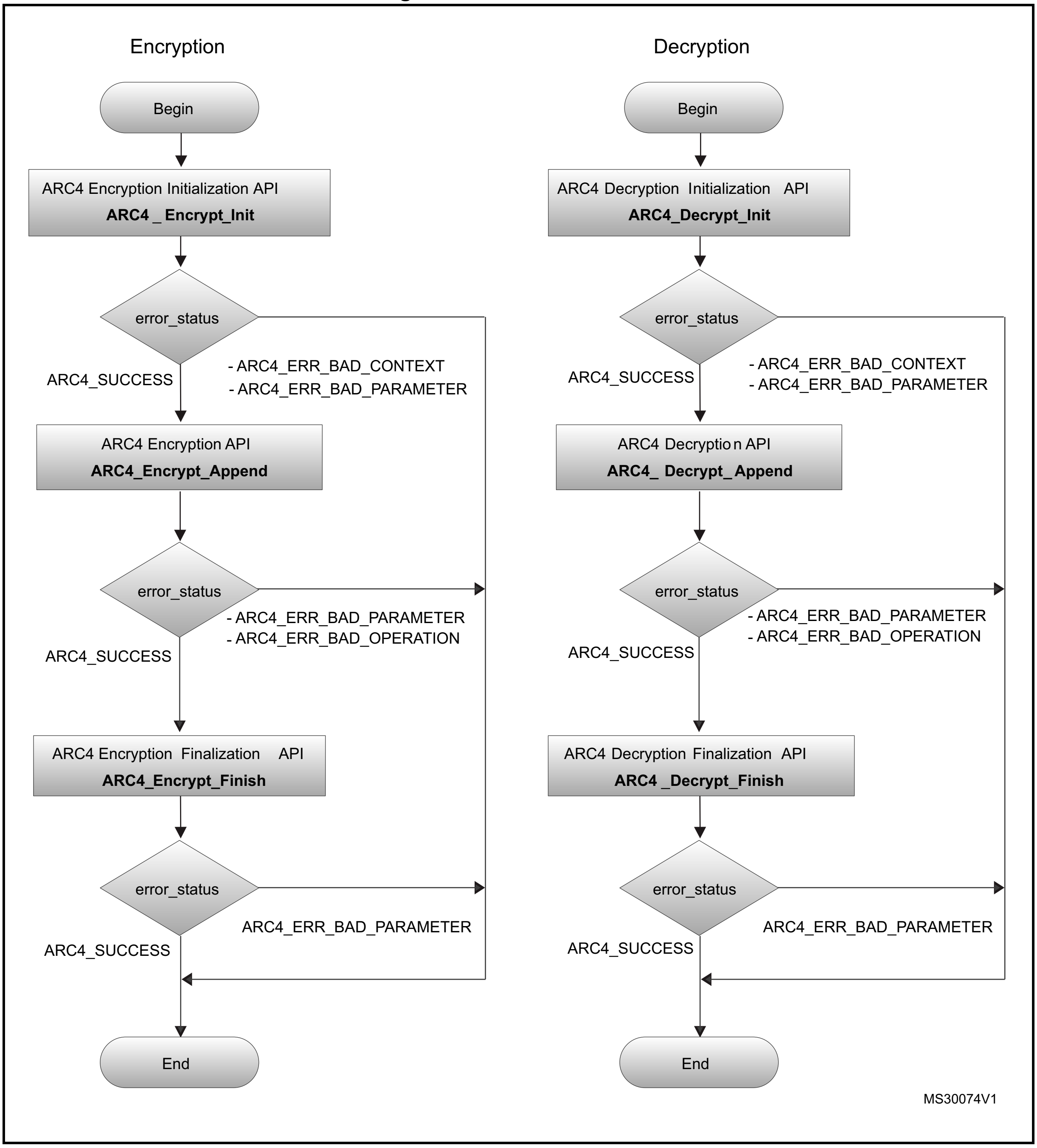
Table 62 describes the ARC library AES functions.

Table 62. ARC4 algorithm functions

Function name	Description
ARC4_Encrypt_Init	Initialization for ARC4 algorithm
ARC4_Encrypt_Append	ARC4 encryption
ARC4_Encrypt_Finish	ARC4 finalization
ARC4_Decrypt_Init	Initialization for ARC4 algorithm
ARC4_Decrypt_Append	ARC4 decryption
ARC4_Decrypt_Finish	ARC4 finalization

The next flowchart describes the ARC4 algorithm.

Figure 12. ARC4 flowchart



5.2.1 ARC4_Encrypt_Init function

Table 63. ARC4_Encrypt_Init

Function name	ARC4_Encrypt_Init

Prototype	`int32_t ARC4_Encrypt_Init (`
	`ARC4ctx_stt * P_pARC4ctx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pIv )`

Behavior	Initialization for ARC4 algorithm
Parameter	– [in,out] *P_pARC4ctx: ARC4 context – [in] *P_pKey: Buffer with the Key – [in] *P_pIv: Buffer with the IV(1)
Return value	– ARC4_SUCCESS: Operation Successful – ARC4_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – ARC4_ERR_BAD_CONTEXT: Context not initialized with valid value. See note (1)

1. In ARC4 the IV is not used, so the value of P_pIv is not checked or used

Note: P_pARC4ctx.mKeySize (see ARC4ctx_stt) must be set with the size of the key prior to calling this function.

5.2.2 ARC4_Encrypt_Append function

Table 64. ARC4_Encrypt_Append

Function name	ARC4_Encrypt_Append

Prototype	`int32_t ARC4_Encrypt_Append (`
	`ARC4ctx_stt * P_pARC4ctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	ARC4 Encryption
Parameter	– [in,out] *P_pARC4ctx: ARC4, already initialized, context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data, expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– ARC4_SUCCESS: Operation Successful – ARC4_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – ARC4_ERR_BAD_OPERATION: Append can't be called after a final

Note: This function can be called multiple times.

ARC4ctx_stt data structure

Structure describing an ARC4 content.

Table 65. ARC4ctx_stt data structure

Field name	Description
`uint32_t mContextId`	Unique ID of this AES-GCM Context. Not used in current implementation.
`SKflags_et mFlags`	32 bit mFlags, used to perform keyschedule, see *SKflags_et mFlags*
`const uint8_t * pmKey`	Pointer to original Key buffer
`int32_t mKeySize`	ARC4 key length in bytes. This must be set by the caller prior to calling Init
`int8_t mX`	Internal members: This describe one of two index variables of the ARC4 state.
`int8_t mY`	Internal members: This describe one of two index variables of the ARC4 state.
`uint8_t amState[256]`	Internal members: This describe the 256 bytes State Matrix

5.2.3 ARC4_Encrypt_Finish function

Table 66. ARC4_Encrypt_Finish

Function name	ARC4_Encrypt_Finish

Prototype	`int32_t ARC4_Encrypt_Finish (`
	`ARC4ctx_stt * P_pARC4ctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	ARC4 Finalization
Parameter	– [in,out] *P_pARC4ctx: ARC4 context – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– ARC4_SUCCESS: Operation Successful – ARC4_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer

5.2.4 ARC4_Decrypt_Init function

Table 67. ARC4_Decrypt_Init

Function name	**ARC4_Decrypt_Init**

Prototype	`int32_t ARC4_Decrypt_Init (`
	`ARC4ctx_stt * P_pARC4ctx,`
	`const uint8_t * P_pKey,`
	`const uint8_t * P_pIv )`

Behavior	Initialization for ARC4 algorithm
Parameter	– [in,out] *P_pARC4ctx: ARC4 context – [in] *P_pKey: Buffer with the Key – [in] *P_pIv: Buffer with the IV(1)
Return value	– ARC4_SUCCESS: Operation Successful – ARC4_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – ARC4_ERR_BAD_CONTEXT: Context not initialized with valid values, see note

1. In ARC4 the IV is not used, so the value of P_pIv is not checked or used

Note: P_pARC4ctx.mKeySize (see ARC4ctx_stt) must be set with the size of the key before calling this function.

5.2.5 ARC4_Decrypt_Append function

Table 68. ARC4_Decrypt_Append

Function name	**ARC4_Decrypt_Append**

Prototype	`int32_t ARC4_Decrypt_Append (`
	`ARC4ctx_stt * P_pARC4ctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	ARC4 Decryption
Parameter	– [in,out] *P_pARC4ctx: ARC4, already initialized, context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– ARC4_SUCCESS: Operation Successful – ARC4_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – ARC4_ERR_BAD_OPERATION: Append can't be called after a Final

Note: This function can be called multiple times.

5.2.6 ARC4_Decrypt_Finish function

Table 69 describes ARC4_Decrypt_Finish function.

Table 69. ARC4_Decrypt_Finish

Function name	**ARC4_Decrypt_Finish**

Prototype	`int32_t ARC4_Decrypt_Finish (`
	`ARC4ctx_stt * P_pARC4ctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	ARC4 Finalization
Parameter	– [in,out] *P_pARC4ctx: ARC4, already initialized, context – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	– ARC4_SUCCESS: Operation Successful – ARC4_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer

Note: This function won't write output data, thus it can be skipped. It is kept for API compatibility.

5.3 ARC4 example

```
#include "crypto.h"
const uint8_t InputMessage[32] = { 0x00,};
uint32_t InputLength = sizeof(InputMessage);
/* Key to be used for ARC4 encryption/decryption */
uint8_t Key[5] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
/* Buffer to store the output data */
uint8_t OutputMessage[ARC4_LENGTH];
/* Size of the output data */
uint32_t OutputMessageLength = 0;
int main(void)
{
  ARC4ctx_stt ARC4ctx;
   uint32_t error_status = ARC4_SUCCESS;
   int32_t outputLength = 0;
  /* Set flag field to default value */
  ARC4ctx.mFlags = E_SK_DEFAULT;
  /* Set key length in the context */
  ARC4ctx.mKeySize = KeyLength;
  /* Initialize the operation, by passing the key.
   * Third parameter is NULL because ARC4 doesn't use any IV */
  error_status = ARC4_Encrypt_Init(&ARC4ctx, ARC4_Key, NULL );
  /* check for initialization errors */
  if(error_status == ARC4_SUCCESS)
  {
    /* Encrypt Data */
    error_status = ARC4_Encrypt_Append(&ARC4ctx,
                                       InputMessage,
                                       InputMessageLength,
                                       OutputMessage,
                                       &outputLength);
    if(error_status == ARC4_SUCCESS)
    {
       /* Write the number of data written*/
      *OutputMessageLength = outputLength;
      /* Do the Finalization */
      error_status = ARC4_Encrypt_Finish(&ARC4ctx, OutputMessage +
*OutputMessageLength, &outputLength);
      /* Add data written to the information to be returned */
      *OutputMessageLength += outputLength;
    }
  }
  return error_status;
}
```

6 RNG algorithm

6.1 Description

The security of cryptographic algorithms relies on the impossibility of guessing the key. The key has to be a random number, otherwise the attacker can guess it.

Random number generation (RNG) is used to generate an unpredictable series of numbers. The random engine is implemented in software using a CTR_ DRBG based on AES-128, while a True RNG is done entirely by the hardware peripheral in the STM32F21x and STM32F41x.

The STM32 cryptographic library includes functions required to support the RNG module to generate a random number.

This algorithm can run with the STM32F1, STM32L1, STM32F20x, STM32F05x, STM32F40x, STM32F37x and the STM32F30x series using a software algorithm implementation. This algorithm can also run using the random number generator peripheral in STM32F21x and STM32F41x.

For RNG library settings, refer to *Section 10: STM32 encryption library settings*.

For RNG library performance and memory requirements, refer to *Section 11: Cryptographic library performance and memory requirements*.

6.2 RNG library functions

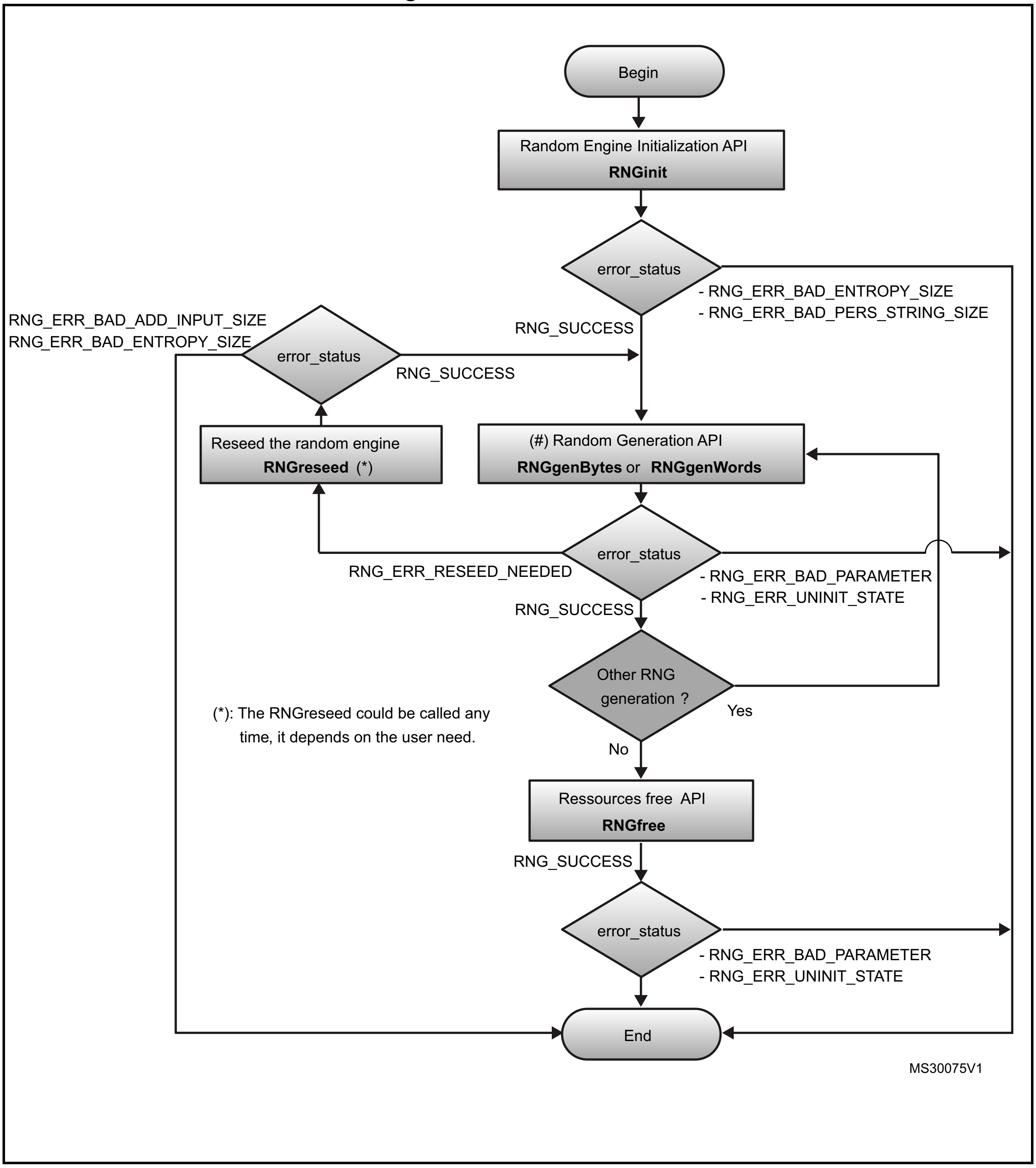
Table 70 describes the RNG library functions.

Table 70. RNG algorithm functions

Function name	Description
`RNGreseed`	Reseed the random engine
`RNGinit`	Initialize the random engine
`RNGfree`	Free a random engine state structure
`RNGgenBytes`	Generation of pseudorandom octets to a buffer
`RNGgenWords`	Generation of a random uint32_t array

The next flowchart describes the RNG algorithm.

Figure 13. RNG flowchart



6.2.1 RNGreseed function

Table 71. RNGreseed

Function name	RNGreseed

Prototype	`int32_t RNGreseed (`
	`const RNGreInput_stt * P_pInputData,`
	`RNGstate_stt * P_pRandomState)`

Behavior	Reseed the random engine
Parameter	– [in] *P_pInputData: Pointer to a client in initialized RNGreInput_stt structure containing the required parameters for a DRBG reseed – [in,out] *P_pRandomState: The RNG status that will be reseeded
Return value	– RNG_SUCCESS: Operation Successful – RNG_ERR_BAD_ADD_INPUT_SIZE: Wrong size for P_pAddInput. It must be less than CRL_DRBG_AES_MAX_ADD_INPUT_LEN – RNG_ERR_BAD_ENTROPY_SIZE: Wrong size for P_entropySize

RNGreInput_stt struct reference

Structure used by RNGinit to initialize a DRBG

Table 72. RNGreInput_stt struct reference

Field name	Description
`uint8_t * pmEntropyData`	The entropy data input
`int32_t mEntropyDataSize`	Size of entropy data input
`uint8_t * pmAddInput`	Additional input
`int32_t mAddInputSize`	Size of additional input

RNGstate_stt struct reference

Structure that contains the by RNG state

Table 73. RNGstate_stt struct reference

Field name	Description
`uint8_t mRNGstate[CRL_DRBG_AES128_STATE_SIZE]`	Underlying DRBG context. It is initialized by RNGinit
`int32_t mDRBGtype`	Specify the type of DRBG to use
`uint32_t mFlag`	Used to check if the random state has been mFlag

6.2.2 RNGinit function

Table 74. RNGinit

Function name	RNGinit

Prototype	`int32_t RNGinit (`
	`const RNGinitInput_stt * P_pInputData,`
	`int32_t P_DRBGtype,`
	`RNGstate_stt * P_pRandomState )`

Behavior	Initialize the random engine
Parameter	– [in] *P_pInputData: Pointer to an initialized RNGinitInput_stt structure with the parameters needed to initialize a DRBG. In case P_DRBGtype==C_HW_RNG it can be NULL – [out] *P_pRandomState: The state of the random engine that will be initialized – [in] P_DRBGtype: Specify the type of DRBG to use. Possible choices are: - C_DRBG_AES128 NIST DRBG based on AES-128 - C_HW_RNG Hardware RNG (if device supports it)
Return value	– RNG_SUCCESS: Operation Successful – RNG_ERR_BAD_ENTROPY_SIZE: Wrong size for P_pEntropyInput. It must be greater than CRL_DRBG_AES128_ENTROPY_MIN_LEN and less than CRL_DRBG_AES_ENTROPY_MAX_LEN – RNG_ERR_BAD_PERS_STRING_SIZE: Wrong size for P_pPersStr. It must be less than CRL_DRBG_AES_MAX_PERS_STR_LEN

Note: *1. This function requires that:*
- P_pInputData.pmEntropyData points to a valid buffer containing entropy data.
- P_pInputData->mEntropyDataSize specifies the size of the entropy data (it should be greater than CRL_DRBG_AES128_ENTROPY_MIN_LEN and less than CRL_DRBG_AES_ENTROPY_MAX_LEN).
- P_pInputData->pmNonce points to a valid Nonce or be set to NULL.
- P_pInputData->mNonceSize specifies the size of the Nonce or be set to zero.
- P_pInputData->pmPersData points to a valid Personalization String or be set to NULL.
- P_pInputData->mPersDataSize specifies size of Personalization String or be set to zero.

2. Section 4 of href ="http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf">NIST SP 800-90A explains the meaning of Nonce, Personalization String and Entropy data.

RNGstate_stt struct reference

Structure that contains the by RNG state

Table 75. RNGstate_stt struct reference

Field name	Description
`uint8_t * pmEntropyData`	Entropy data input
`int32_t mEntropyDataSize`	Size of the entropy data input
`uint8_t * pmNonce`	Nonce data
`uint32_t mNonceSize`	Size of the Nonce
`int8_t* pmPersData`	Personalization String
`uint32_t mPersDataSize`	Size of personalization string

6.2.3 RNGfree function

Table 76. RNGfree

Function name	**RNGfree**
Prototype	`int32_t RNGfree (` `RNGstate_stt * P_pRandomState)`
Behavior	Free a random engine state structure
Parameter	– [in,out] *P_pRandomState: The state of the random engine that will be removed
Return value	– RNG_SUCCESS: Operation Successful – RNG_ERR_BAD_PARAMETER: P_pRandomState == NULL – RNG_ERR_UNINIT_STATE: Random engine not initialized

6.2.4 RNGgenBytes function

Table 77. RNGgenBytes

Function name	**RNGgenBytes**

Prototype	`int32_t RNGgenBytes (`
	`RNGstate_stt * P_pRandomState,`
	`const RNGaddInput_stt *P_pAddInput,`
	`uint8_t * P_pOutput,`
	`int32_t P_OutLen )`

Behavior	Generation of pseudorandom octets to a buffer
Parameter	– [in,out] *P_pRandomState: The current state of the random engine – [in] *P_pAddInput: Optional Additional Input (can be NULL) – [in] *P_pOutput: The output buffer – [in] P_OutLen: The number of random octets to generate
Return value	– RNG_SUCCESS: Operation Successful – RNG_ERR_BAD_PARAMETER: P_pRandomState == NULL or P_pOutput == NULL && P_OutLen > 0 – RNG_ERR_UNINIT_STATE: Random engine not initialized – RNG_ERR_RESEED_NEEDED: Returned only if it's defined CRL_RANDOM_REQUIRE_RESEED. The count of number of requests between reseed has reached its limit.Reseed is necessary

Note: The user has to be careful to not invoke this function more than 2^48 times without calling the RNGreseed function.

6.2.5 RNGgenWords function

Table 78. RNGgenWords

Function name	RNGgenWords
Prototype	`int32_t RNGgenWords ( RNGstate_stt * P_pRandomState, const RNGaddInput_stt *P_pAddInput, uint32_t * P_pWordBuf, int32_t P_BufSize )`
Behavior	Generation of a random uint32_t array
Parameter	– [in,out] *P_pRandomState: The random engine current state – [in] *P_pAddInput: Optional Additional Input (can be NULL) – [out] *P_pWordBuf: The buffer where the uint32_t array will be stored – [in] P_BufSize: The number of uint32_t to generate.
Return value	– RNG_SUCCESS: Operation Successful – RNG_ERR_BAD_PARAMETER: P_pRandomState == NULL or P_pOutput == NULL && P_OutLen > 0 – RNG_ERR_UNINIT_STATE: Random engine not initialized. – RNG_ERR_RESEED_NEEDED:Returned only if it's defined CRL_RANDOM_REQUIRE_RESEED. If the count of number of requests between reseed has reached its limit.Reseed is necessary

6.3 RNG example

A simple random generation with C_SW_DRBG_AES128 is shown below:

```
#include "crypt.h"
 int32_t main()
 {
   /* Structure that will keep the random state */
   RNGstate_stt RNGstate;
/* Structure for the parmeters of initialization */
   RNGinitInput_stt RNGinit_st;
/* String of entropy */
   uint8_t
entropy_data[32]={0x9d,0x20,0x1a,0x18,0x9b,0x6d,0x1a,0xa7,0x0e,0x79,0x57,0
x6f,0x36,0xb6,0xaa,0x88,0x55,0xfd,0x4a,0x7f,0x97,0xe9,0x71,0x69,0xb6,0x60,
0x88,0x78,0xe1,0x9c,0x8b,0xa5};
/* Nonce */
   uint8_t nonce[4] = {0,1,2,3};
/* array to keep the returned random bytes */
   uint8_t randombytes[16];
   int32_t retval;

/* Initialize the RNGinit structure */
   RNGinit_st.pmEntropyData = entropy_data;
   RNGinit_st.mEntropyDataSize = sizeof(entropy_data);
   RNGinit_st.pmNonce = nonce;
   RNGinit_st.mNonceSize = sizeof(nonce);
/* There is no personalization data in this case */
   RNGinit_st.mPersDataSize = 0;
   RNGinit_st.pmPersData = NULL;

/* Init the random engine */
   if  ( RNGinit(&RNGinit_st, C_SW_DRBG_AES128,  &RNGstate) != 0)
   {
     printf("Error in RNG initialization\n");
     return(-1);
   }
/* Generate */
   retval = RNGgenBytes(&RNGstate,randombytes,sizeof(randombytes));
   if (retval != 0)
   {
     printf("Error in RNG generation\n");
     return(-1);
   }
   return(0);
 }
```

7 HASH algorithm

7.1 Description

This algorithm provides a way to guarantee the integrity of information, verify digital signatures and message authentication codes. It is based on a one-way hash function that processes a message to produce a small length / condensed message called a message digest.

The STM32 cryprogratphic library includes functions required to support HASH/HMAC modules to guarantee the integrity of information using the following modes:

- MD5
- SHA-1
- SHA-224
- SHA-256

This algorithm can run with the STM32F1, STM32L1, STM32F20x, STM32F05x, STM32F40x, STM32F37x and the STM32F30x series using a software algorithm implementation. You can optimize the performance by using pure hardware accelerators thanks to STM32F21x and STM32F41x devices:

Modes support by the hardware in STM32F21x and STM32F41x are:

- MD5
- SHA-1
- For other modes: SHA-224 or SHA-256 runs using software algorithm implementation

For HASH library settings, refer to *Section 10: STM32 encryption library settings*.

For HASH library performance and memory requirements, refer to *Section 11: Cryptographic library performance and memory requirements*.

7.2 HASH library functions

Table 79. HASH algorithm functions (HHH = MD5, SHA1, SHA224 or SHA256)

Function name	Description
`HHH_Init`	Initialization a Hash algorithm Context
`HHH_Append`	Process input data and the HASH algorithm context that will be updated
`HHH_Finish`	Hash algorithm finish function, produce the output HASH algorithm digest
`HMAC_HHH_Init`	Initialize a new HMAC of select Hash algorithm context
`HMAC_HHH_Append`	Process input data and update a HMAC-Hash algorithm context that will be updated
`HMAC_HHH_Finish`	HMAC-HHH Finish function, produce the output HMAC-Hash algorithm tag

HHH represents the mode of operation of HASH algorithm.

The following mode of operation can be used for HASH algorithm:

- MD5
- SHA1
- SHA224
- SHA256

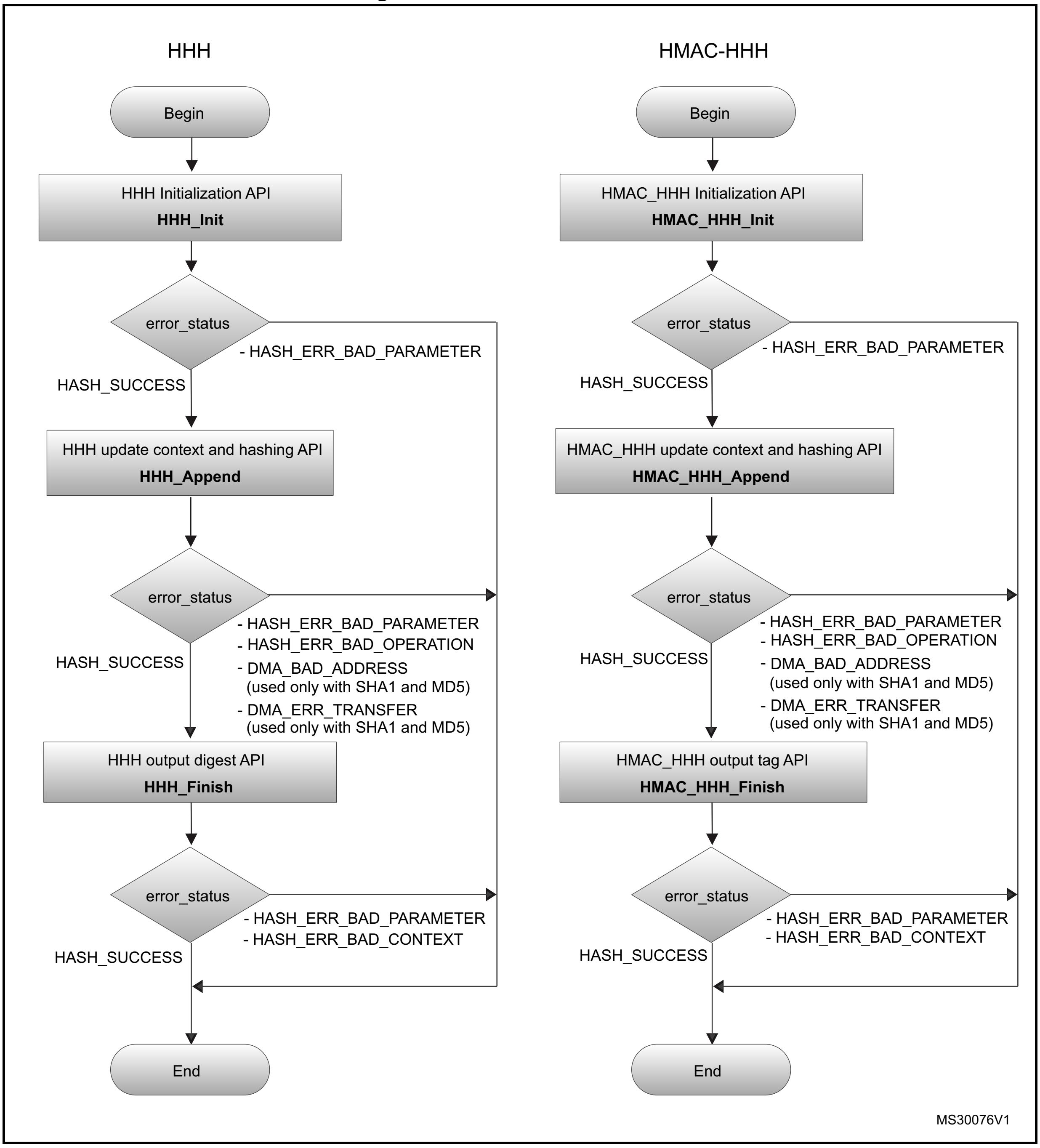
The next flowchart in *Figure 14* describes the HHH algorithm.

For example, if you want to use SHA1 for HASH algorithm, you can call the functions:

Table 80. HASH SHA1 algorithm functions

Function name	Description
`SHA1_Init`	Initialize a new SHA1 context
`SHA1_Append`	SHA1 Update function, process input data and update a SHA1ctx_stt
`SHA1_Finish`	SHA1 Finish function, produce the output SHA1 digest
`HMAC_SHA1_Init`	Initialize a new HMAC SHA1 context
`HMAC_SHA1_Append`	HMAC-SHA1 Update function, process input data and update a HMAC-SHA1 context that will be updated
`HMAC_SHA1_Finish`	HMAC-SHA1 Finish function, produce the output HMAC-SHA1 tag

Figure 14. Hash HHH flowchart



7.2.1 HHH_Init function

Table 81. HHH_Init

Function name	HHH_Init
Prototype	`int32_t HHH_Init (HHHctx_stt *P_pHHHctx)`
Behavior	Initialize a new HHH context
Parameter	– [in, out] *P_pHHHctx: The context that will be initialized
Return value	– HASH_SUCCESS: Operation Successful – HASH_ERR_BAD_PARAMETER: Parameter P_pHHHctx is invalid

Note: *1. HHH is MD5, SHA1, SHA224 or SHA256;*

2. P_pHHHctx.mFlags must be set prior to calling this function. Default value is E_HASH_DEFAULT. See HashFlags_et for details.

3. P_pHHHctx.mTagSize must be set with the size of the required message digest that will be generated by the HHH_Finish . Possible values are values are from 1 to CRL_HHH_SIZE.

HASHctx_stt struct reference

Structure for HASH context

Table 82. HASHctx_stt struct reference

Field name	Description
`uint32_t mContextId`	Unique ID of this context. Not used in current implementation.
`HashFlags_et mFlags`	32 bit mFlags, used to perform keyschedule, see *HashFlags_et mFlags* choose between hw/sw/hw+dma and future use
`int32_t mTagSize`	Size of the required Digest
`uint8_t amBuffer[64]`	Internal: Buffer with the data to be hashed
`uint32_t amCount[2]`	Internal: Keeps the count of processed bits
`uint32_t amState[8]`	Internal: Keeps the internal state

HashFlags_et mFlags

Enumeration of allowed flags in a context for Symmetric Key operations.

Table 83. HashFlags_et mFlags

Field name	Description
`E_HASH_DEFAULT`	User Flag: No flag specified.
`E_HASH_DONT_PERFORM_KEY_SCHEDULE`	User Flag: Forces init to not reperform key processing in HMAC mode.
`E_HASH_USE_DMA`	User Flag: if MD5/SHA-1 has an HW engine; specifies if DMA or CPU transfers data. If DMA, only one call to append is allowed
`E_HASH_OPERATION_COMPLETED`	Internal Flag: checks the Finish function has been already called
`E_HASH_NO_MORE_APPEND_ALLOWED`	Internal Flag: it is set when the last append has been called. Used where the append is called with an InputSize not multiple of the block size, which means that is the last input.

7.2.2 HHH_Append function

Table 84. HHH_Append

Function name	**HHH_Append**

Prototype	`int32_t HHH_Append (`
	`HHHctx_stt * P_pHHHctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize );`

Behavior	Process input data and update a HHHctx_stt
Parameter	– [in,out] *P_pHHHctx: HHH context that will be updated – [in] *P_pInputBuffer:The data that will be processed using HHH. – [in] P_inputSize: Size of input data expressed in bytes
Return value	– HASH_SUCCESS: Operation Successful – HASH_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer. – HASH_ERR_BAD_OPERATION: HHH_Append can't be called after HHH_Finish has been called. If in DMA mode, then SHA1_Append or MD5_Append can be called only once – DMA_BAD_ADDRESS: Input or output buffer addresses are not word aligned (used only in SHA1 and MD5) – DMA_ERR_TRANSFER: Errors in the DMA transfer (used only in SHA1 and MD5)

Note: *1. HHH is MD5, SHA1, SHA224 or SHA256*

2. In DMA mode ((P_pMD5ctx->mFlags & E_HASH_USE_DMA)==E_HASH_USE_DMA) the Append function can be called one time only, otherwise it will return HASH_ERR_BAD_OPERATION

3. In DMA mode ((P_pSHA1ctx->mFlags & E_HASH_USE_DMA)==E_HASH_USE_DMA) the Append function can be called one time only, otherwise it will return HASH_ERR_BAD_OPERATION

4. This function can be called multiple times with no restrictions on the value of P_inputSize

7.2.3 HHH_Finish function

Table 85. HHH_Finish

Function name	**HHH_Finish**

Prototype	`int32_t HHH_Finish (`
	`HHHctx_stt * P_pHHHctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize )`

Behavior	HHH Finish function, produce the output HHH digest
Parameter	– [in,out] *P_pHHHctx: HASH context – [out] *P_pOutputBuffer: Buffer that will contain the digest – [out] *P_pOutputSize: Size of the data written to P_pOutputBuffer
Return value	– HASH_SUCCESS: Operation Successful – HASH_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer. – HASH_ERR_BAD_CONTEXT: P_pHHHctx not initialized with valid values, see the notes below.

Note: *1. HHH is MD5, SHA1, SHA224 or SHA256.*

2. P_pSHA1ctx->mTagSize must contain a valid value, between 1 and CRL_HHH_SIZE before calling this function.

7.2.4 HMAC_HHH_Init function

Table 86. HMAC_HHH_Init

Function name	**HMAC_HHH_Init**
Prototype	`int32_t HMAC_HHH_Init (HMAC_HHHctx_stt * P_pHMAC_HHHctx);`
Behavior	Initialize a new HMAC HHH context
Parameter	– [in, out] *P_pHMAC_HHHctx: The context that will be initialized
Return value	– HASH_SUCCESS: Operation Successful – HASH_ERR_BAD_PARAMETER: Parameter P_pHMAC_HHHctx is invalid

Note: *1. HHH is MD5, SHA1, SHA224 or SHA256).*

2. P_pHMAC_HHHctx.pmKey (see HMAC_HHHctx_stt) must be set with a pointer to HMAC key before calling this function.

3. P_pHMAC_HHHctx.mKeySize (see HMAC_HHHctx_stt) must be set with the size of the key (in bytes) prior to calling this function.

4. P_pHMAC_HHHctx.mFlags must be set prior to calling this function. Default value is E_HASH_DEFAULT. See HashFlags_et for details.

5. P_pHMAC_HHHctx.mTagSize must be set with the size of the required authentication TAG that will be generated by the HMAC_HHH_Finish. Possible values are from 1 to CRL_HHH_SIZE.

HMACctx_stt struct reference

Structure for HMAC context

Table 87. HMACctx_stt struct reference

Field name	Description
`uint32_t mContextId`	Unique ID of this context. **Not used** in current implementation.
`HashFlags_et mFlags`	32 bit mFlags, used to perform keyschedule, see *HashFlags_et mFlags*
`int32_t mTagSize`	Size of the required Digest
`const uint8_t * pmKey`	Pointer for the HMAC key
`int32_t mKeySize`	Size, in uint8_t (bytes) of the HMAC key
`uint8_t amKey64]`	Internal: The HMAC key
`HASHctx_stt mHASHctx_st`	Internal: Hash Context, please refer to *HASHctx_stt struct reference*

7.2.5 HMAC_HHH_Append function

Table 88. HMAC_HHH_Append

Function name	HMAC_HHH_Append

Prototype	`int32_t HMAC_HHH_Append (`
	`HMAC_HHHctx_stt * P_pHMAC_HHHctx,`
	`const uint8_t * P_pInputBuffer,`
	`int32_t P_inputSize )`

Behavior	HMAC-HHH Update function, process input data and update a HMAC_HHHctx_stt
Parameter	– [in,out] *P_pHMAC_HHHctx: The HMAC-HHH context that will be updated – [in] *P_pInputBuffer:The data that will be processed using HMAC-HHH – [in] P_inputSize: Size of input data, expressed in bytes
Return value	– HASH_SUCCESS: Operation Successful – HASH_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer. – HASH_ERR_BAD_OPERATION: HMAC_HHH_Append can't be called after HMAC_HHH_Finish has been called. – DMA_BAD_ADDRESS: Input or output buffer addresses are not word aligned (used only in sha1 and md5) – DMA_ERR_TRANSFER: Errors in DMA transfer (used only in sha1 and md5)

Note:

1. HHH is MD5, SHA1, SHA224 or SHA256).

2. In DMA mode ((P_pHMAC_MD5ctx->mFlags & E_HASH_USE_DMA)== E_HASH_USE_DMA), the Append function can be called one time only, otherwise it will return HASH_ERR_BAD_OPERATION.

3. In DMA mode ((P_pHMAC_SHA1ctx->mFlags & E_HASH_USE_DMA)== E_HASH_USE_DMA), the Append function can be called one time only, otherwise it will return HASH_ERR_BAD_OPERATION.

4. This function can be called multiple times with no restrictions on the value of P_inputSize.

7.2.6 HMAC_HHH_Finish function

Table 89. HMAC_HHH_Finish

Function name	**HMAC_HHH_Finish**

Prototype	`int32_t HHH_Finish (`
	`HMAC_HHHctx_stt * P_pHMAC_HHHctx,`
	`uint8_t * P_pOutputBuffer,`
	`int32_t * P_pOutputSize)`

Behavior	HMAC-HHH Finish function, produce the output HMAC-HHH tag
Parameter	– [in,out] *P_pHMAC_HHHctx: HMAC-HHH context – [out] *P_pOutputBuffer: Buffer that will contain the HMAC tag – [out] *P_pOutputSize: Size of the data written to P_pOutputBuffer
Return value	– HASH_SUCCESS: Operation Successful – HASH_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer – HASH_ERR_BAD_CONTEXT: P_pHHHctx was not initialized with valid values

Note: HHH is MD5, SHA1, SHA224 or SHA256)
P_pHHHctx->mTagSize must contain a valid value, between 1 and CRL_HHH_SIZE;

7.3 HASH SHA1 example

A simple example of using SHA-1 is shown in the following example:

```
#include "crypt.h"
 int32_t main()
 {
   uint8_t input[141]={ ... };
   uint8_t digest[20];
   int32_t outSize;
   /* SHA-1 Context Structure
   SHA1ctx_stt SHA1ctx_st;

/* Set the size of the desired hash digest */
   SHA1ctx_st.mTagSize = 20;
/* Set flag field to default value */
   SHA1ctx_st.mFlags = E_HASH_DEFAULT;

/* Initialize context */
   retval = SHA1_Init(&SHA1ctx_st);
   if (retval != HASH_SUCCESS)
   { ... }

   retval = SHA1_Append(&SHA1ctx_st, input, sizeof(input));
   if (retval != HASH_SUCCESS)
   { ... }

   retval = SHA1_Finish(&SHA1ctx_st, digest, &outSize);
   if (retval != HASH_SUCCESS)
   { ... }

   printf("Resulting SHA-1 digest: ");
   for (i = 0; i < outSize; i++)
   {
     printf("%02X", digest[i]);
   }
   return(0);
 }
```

8 RSA algorithm

8.1 Description

This section describes RSA functions for signature generation/validation.

These functions should only be used for signature verification (modular exponentiation with a small exponent), because the more efficient functions for modular exponentiation have been removed to save memory footprint.

There are two structures that pass keys to the functions:

- RSAprivKey_stt for the private key
- RSApubKey_stt for the public key

The values of the byte arrays pointed to by the above structures, as well as the signature, must be byte arrays, where the byte at index 0 represents the most significant byte of the integer (modulus, signature or exponent).

All members of the above functions should be filled by the user before calls to the following RSA functions:

- RSA_PKCS1v15_Sign
- RSA_PKCS1v15_Verify

Note that the configuration switch RSA_WINDOW_SIZE can speedup operations with the private key - at the expense of RAM memory.

Please refer to *Section 10: STM32 encryption library settings* for more detail.

These modes can run in STM32F1, STM32L1, STM32F2, STM32F05x, STM32F4 and STM32F3 series using a software algorithm implementation.

For RSA library performance and memory requirements, refer to *Section 11: Cryptographic library performance and memory requirements*.

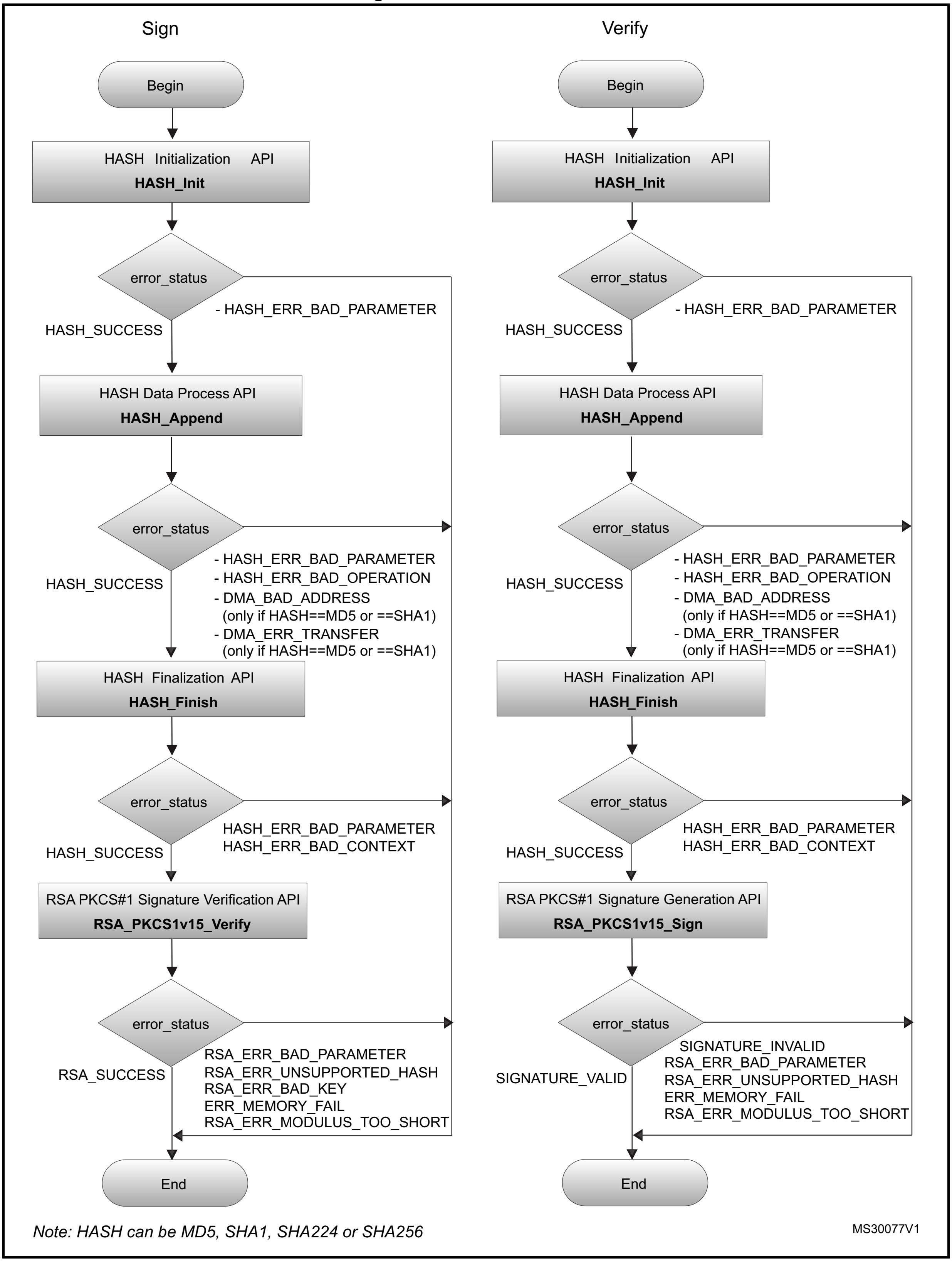
8.2 RSA library functions

Table 90. RSA algorithm functions

Function name	Description
`RSA_PKCS1v15_Sign`	PKCS#1v1.5 RSA Signature Generation Function
`RSA_PKCS1v15_Verify`	PKCS#1v1.5 RSA Signature Verification Function
`RSASP1`	PKCS#1v1.5 RSA function for Signature Generation
`RSAVP1`	PKCS#1v1.5 RSA function for Signature Verification

The flowchart below describes the RSA algorithm

Figure 15. RSA flowchart



8.2.1 RSA_PKCS1v15_Sign function

Table 91. RSA_PKCS1v15_Sign function

Function name	RSA_PKCS1v15_Sign(1)

Prototype	`int32_t RSA_PKCS1v15_Sign(`
	`const RSAprivKey_stt * P_pPrivKey,`
	`const uint8_t * P_pDigest,`
	`hashType_et P_hashType,`
	`uint8_t * P_pSignature,`
	`membuf_stt *P_pMemBuf)`

Behavior	PKCS#1v1.5 RSA Signature Generation Function
Parameter	– [in] *P_pPrivKey: RSA private key structure (RSAprivKey_stt) – [in] *P_pDigest: The message digest that will be signed – [in] P_hashType: Identifies the type of Hash function used – [out] *P_pSignature: The returned message signature – [in] *P_pMemBuf: Pointer to the membuf_stt structure that will be used to store the internal values required by computation
Return value	– RSA_SUCCESS: Operation Successful – RSA_ERR_BAD_PARAMETER: Some of the inputs were NULL – RSA_ERR_UNSUPPORTED_HASH: Hash type passed not supported – RSA_ERR_BAD_KEY: Some member of structure P_pPrivKey were invalid – ERR_MEMORY_FAIL: Not enough memory left available – RSA_ERR_MODULUS_TOO_SHORT: RSA modulus too short for this hash type

1. P_pSignature has to point to a memory area of suitable size (modulus size)
The structure pointed by P_pMemBuf must be properly initialized

RSAprivKey_stt data structure: Structure type for RSA private key

Table 92. RSAprivKey_stt data structure

Field name	Description
uint8_t* pmModulus	RSA Modulus
int32_t mModulusSize	Size of RSA Modulus
uint8_t* pmExponent	RSA Private Exponent
int32_t mExponentSize	Size of RSA Private Exponent

membuf_stt data structure: Structure type definition for a pre-allocated memory buffer

Table 93. membuf_stt data structure

Field name	Description
uint8_t* pmBuffer	Pointer to the pre-allocated memory buffer
uint16_t mSize	Total size of the pre-allocated memory buffer
uint16_t mUsed	Currently used portion of the buffer, should be inititalized by user to zero

8.2.2 RSA_PKCS1v15_Verify function

Table 94. RSA_PKCS1v15_Verify function

Function name	RSA_PKCS1v15_Verify

Prototype	`int32_t RSA_PKCS1v15_Verify( const RSApubKey_stt *P_pPubKey,`
	`const uint8_t *P_pDigest,`
	`hashType_et P_hashType,`
	`const uint8_t *P_pSignature,`
	`membuf_stt *P_pMemBuf)`

Behavior	PKCS#1v1.5 RSA Signature Verification Function
Parameter	– [in] *P_pPubKey: RSA public key structure (RSApubKey_stt) – [in] *P_pDigest: The hash digest of the message to be verified – [in] P_hashType: Identifies the type of Hash function used – [in] *P_pSignature: The signature that will be checked – [in] *P_pMemBuf Pointer to the membuf_stt structure that will be used to store the internal values required by computation
Return value	– SIGNATURE_VALID: The Signature is valid – SIGNATURE_INVALID: The Signature is NOT valid – RSA_ERR_BAD_PARAMETER: Some of the inputs were NULL – RSA_ERR_UNSUPPORTED_HASH: The Hash type passed doesn't correspond to any among the supported ones – ERR_MEMORY_FAIL: Not enough memory left available – RSA_ERR_MODULUS_TOO_SHORT: RSA modulus is too short to handle this hash type

Note: The structure pointed by P_pMemBuf must be properly initialized

RSApubKey_stt data structure

Structure type for RSA public key.

Table 95. RSApubKey_stt data structure

Field name	Description
uint8_t* pmModulus	RSA Modulus
int32_t mModulusSize	Size of RSA Modulus
uint8_t* pmExponent	RSA Public Exponent
int32_t mExponentSize	Size of RSA Public Exponent

8.3 RSA Signature generation/verification example

```
#include "crypt.h"
 int32_t main()
 {
   uint8_t modulus[2048/8]={ ... };
   uint8_t public_exponent[3]={0x01,0x00,0x01};
   uint8_t digest[CRL_SHA256_SIZE]={...};
   uint8_t signature[2048/8];
   uint8_t private_exponent[2048/8]={...};
   int32_t retval;
   RSAprivKey_stt privKey;
   RSApubKey_stt pubKey;

   /* Set values of private key */
   privKey.mExponentSize = sizeof(private_exponent);
   privKey.pmExponent = private_exponent;
   privKey.mModulusSize = sizeof(modulus);
   privKey.pmModulus = modulus;

   /* Generate the signature, knowing that the hash has been generated by
SHA-256 */
   retval = RSA_PKCS1v15_Sign(&privKey, digest, E_SHA256, signature);
   if (retval != RSA_SUCCESS)
   { return(ERROR); }

   /* Set values of public key */
   pubKey.mExponentSize = sizeof(public_exponent);
   pubKey.pmExponent = public_exponent;
   pubKey.mModulusSize = sizeof(modulus);
   pubKey.pmModulus = modulus;

   /* Verify the signature, knowing that the hash has been generated by SHA-
256 */
   retval = RSA_PKCS1v15_Verify(&pubKey, digest, E_SHA256, signature)
   if (retval != SIGNATURE_VALID )
   { return(ERROR); }
   else
   { return(OK); }
 }
```

9 ECC algorithm

9.1 Description

This section describes Elliptic Curve Cryptography (ECC) primitives, an implementation of ECC Cryptography using Montgomery Multiplication. ECC operations are defined for curves over GF(p) field.

Scalar multiplication is the ECC operation that it is used in ECDSA (Elliptic Curve Digital Signature Algorithm) and in ECDH (Elliptic Curve Diffie-Hellman protocol). It is also used to generate a public key, sign a message and verify signatures.

This mode can run in STM32F1, STM32L1 and STM32F2 series using a software algorithm implementation.

For ECC library settings, refer to *Section 10: STM32 encryption library settings*.

For ECC library performance and memory requirements, refer to *Section 11: Cryptographic library performance and memory requirements*.

9.2 ECC library functions

Table 96. ECC algorithm functions

Function name	Description
`ECCinitEC`	Initialize the elliptic curve parameters into a EC_stt structure
`ECCfreeEC`	De-initialize an EC_stt context
`ECCinitPoint`	Initialize an ECC point
`ECCfreePoint`	Free Elliptic curve point
`ECCsetPointCoordinate`	Set the value of one of coordinate of an ECC point
`ECCgetPointCoordinate`	Get the value of one of coordinate of an ECC point
`ECCcopyPoint`	Copy an Elliptic Curve Point
`ECCinitPrivKey`	Initialize an ECC private key
`ECCfreePrivKey`	Free an ECC Private Key
`ECCsetPrivKeyValue`	Set the value of an ECC private key object from a byte array
`ECCgetPrivKeyValue`	Get the private key value from an ECC private key object
`ECCscalarMul`	Computes the point scalar multiplication kP = k*P
`ECDSAinitSign`	Initialize an ECDSA signature structure
`ECDSAfreeSign`	Free an ECDSA signature structure
`ECDSAsetSignature`	Set the value of the parameters (one at a time) of an ECDSAsignature_stt
`ECDSAgetSignature`	Get the values of the parameters (one at a time) of an ECDSAsignature_stt
`ECDSAverify`	ECDSA signature verification with a digest input

Table 96. ECC algorithm functions (continued)

Function name	Description
`ECCvalidatePubKey`	Checks the validity of a public key.
`ECCkeyGen`	Generate an ECC key pair.
`ECDSAsign`	ECDSA Signature Generation
`ECCgetPointFlag`	Reads the flag member of an Elliptic Curve Point structure
`ECCsetPointFlag`	Set the flag member of an Elliptic Curve Point structure
`ECCsetPointGenerator`	Writes the Elliptic Curve Generator point into a ECpoint_stt

The next flowcharts describe the ECC algorithms.

Figure 16. ECC Sign flowchart

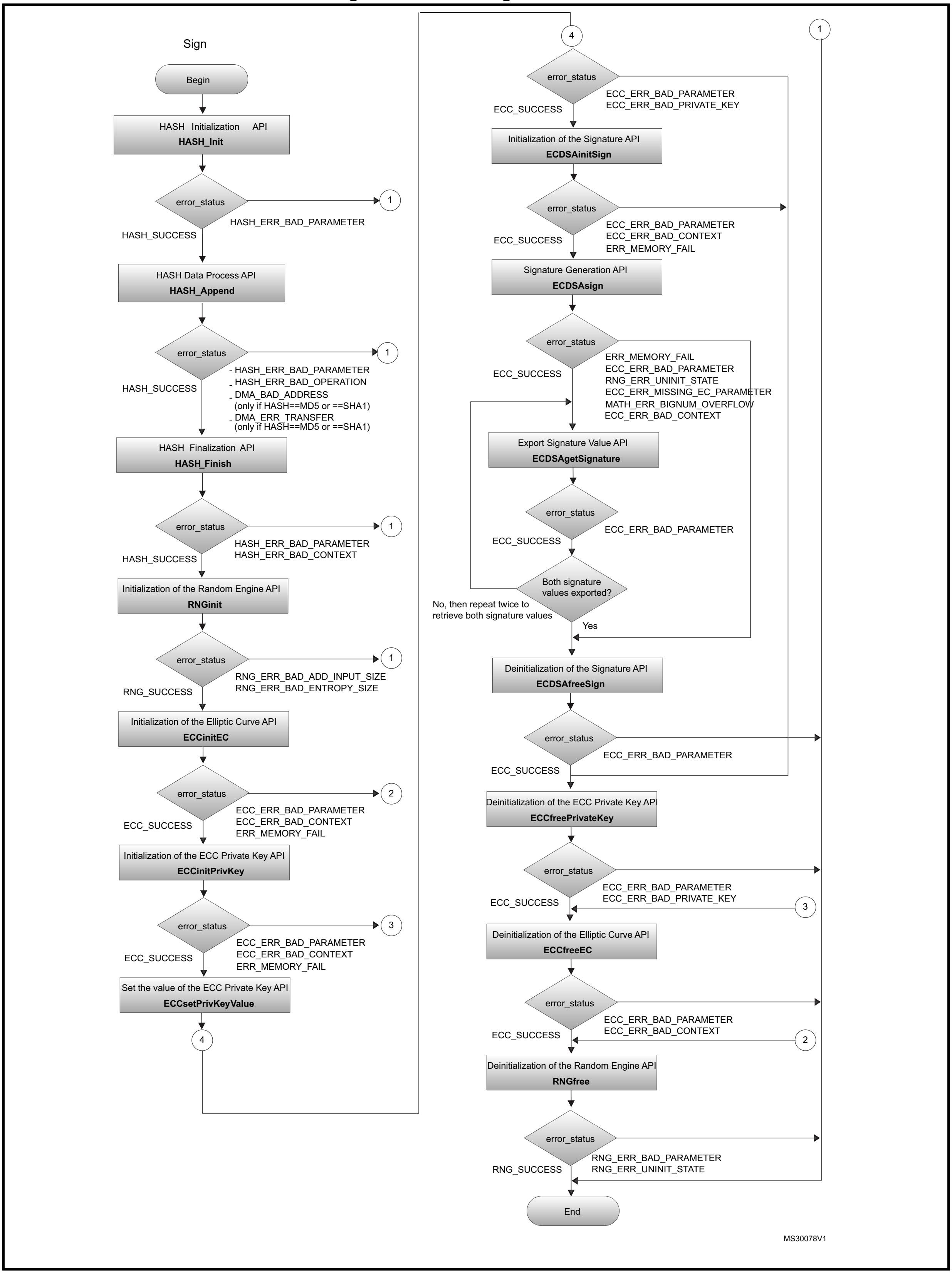


Figure 17. ECC Verify flowchart

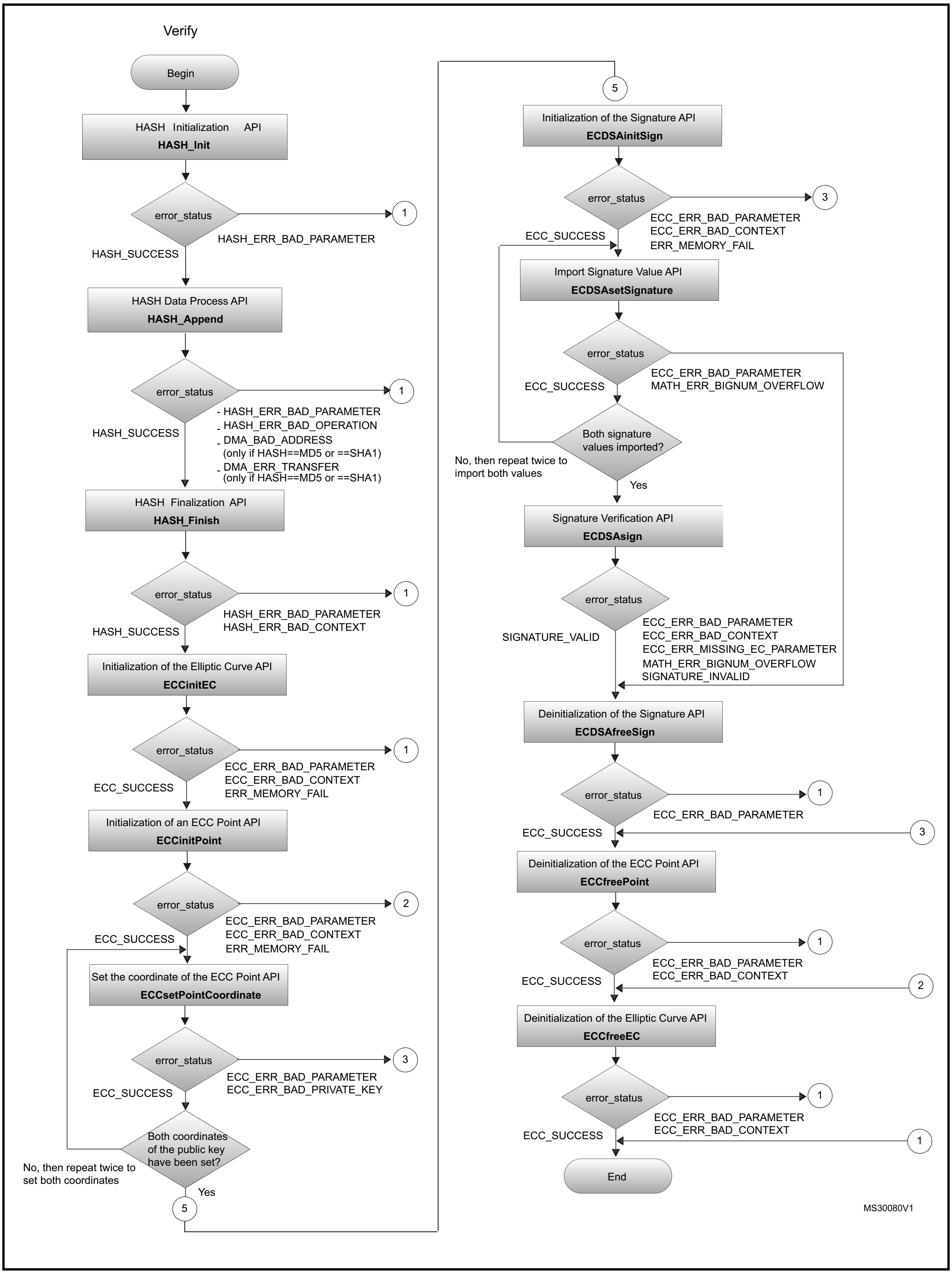
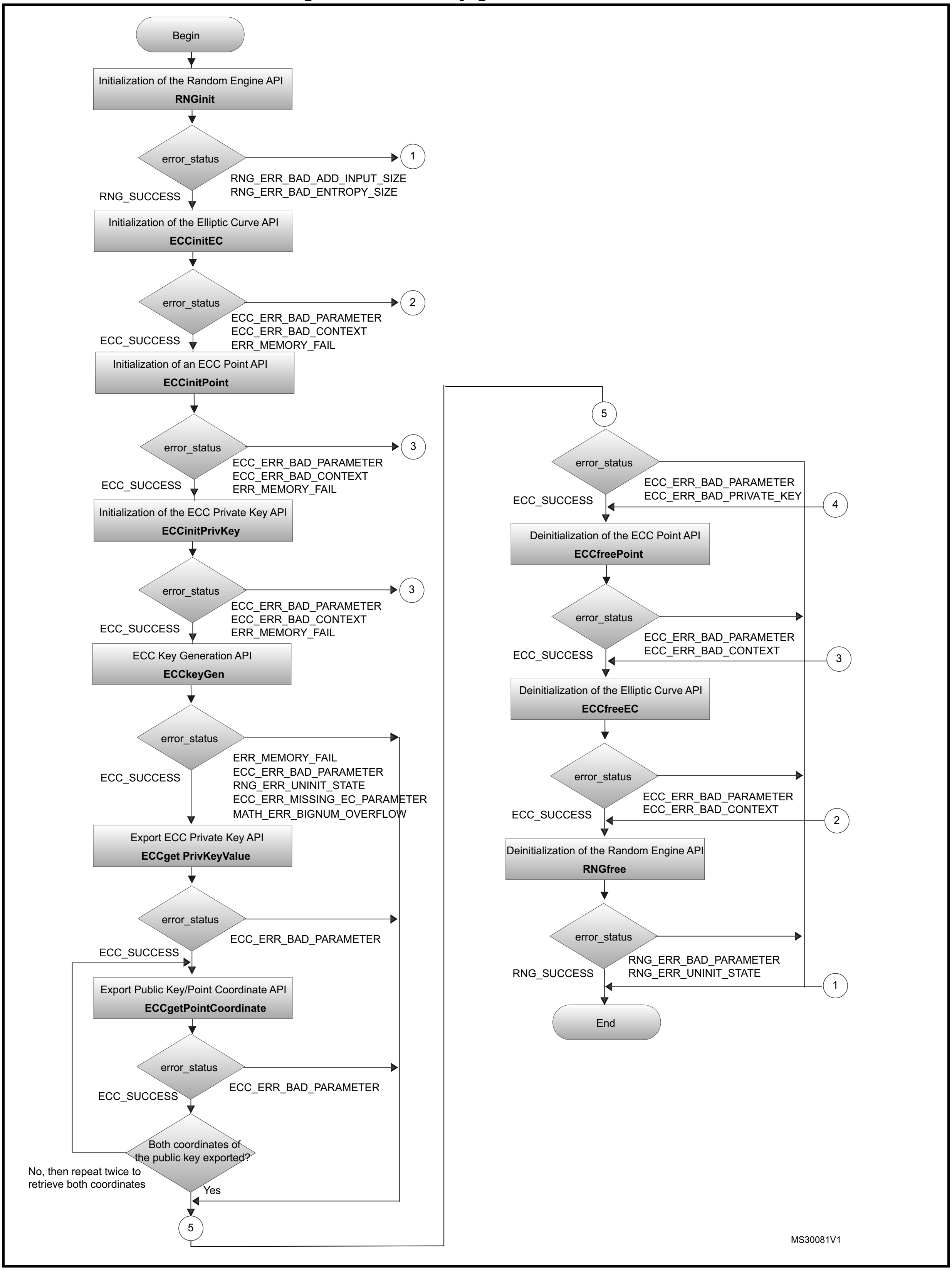


Figure 18. ECC key generator flowchart



9.2.1 ECCinitEC function

This is the first EC operation performed; it loads elliptic curve domain parameters.

Table 97. ECCinitEC function

Function name	**ECCinitEC**
Prototype	`int32_t ECCinitEC(EC_stt *P_pECctx, membuf_stt *P_pMemBuf)`
Behavior	Initialize the elliptic curve parameters into a EC_stt structure
Parameter	– [in,out] *P_pECctx: EC_stt context with parameters of ellliptic curve used – [in,out] *P_pMemBuf: Pointer to membuf_stt structure that will be used to store the Ellitpic Curve internal values
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: P_pECctx == NULL – ECC_ERR_BAD_CONTEXT: Some values inside P_pECctx are invalid – ERR_MEMORY_FAIL: Not enough memory

Note: *1. Not every parameter needs to be loaded. It depends on the operation:*
- Every operation requires at least "a" and "p" and "n".
- Set Generator requires "Gx" and "Gy"
- Verification of the validity of a public key requires "b"

2. P_pMemBuf must be initialized before calling this function. See membuf_stt.

3. This function keeps some values stored in membuf_stt.pmBuf, so on exiting membuf_stt.mUsed won't be set to zero. The caller can use the same P_pMemBuf also for other functions. The memory is freed when ECCfreeEC is called.

EC_stt data structure

Structure used to store the parameters of the elliptic curve actually selected.
Elliptic Curve equation over GF(p): y^2=x^3+ax+b mod(p). Structure that keeps the Elliptic Curve Parameters.

Table 98. EC_stt data structure

Field name	**Description**
const uint8_t * pmA	Pointer to parameter "a"
int32_t mAsize	Size of parameter "a"
const uint8_t * pmB	Pointer to parameter "b"
int32_t mBsize	Size of parameter "b"
const uint8_t * pmP	Pointer to parameter "p"
int32_t mPsize	Size of parameter "p"
const uint8_t * pmN	Pointer to parameter "n"
int32_t mNsize	Size of parameter "n"
const uint8_t * pmGx	Pointer to x coordinate of generator point
int32_t mGxsize	Size of x coordinate of generator point
const uint8_t * pmGy	Pointer to y coordinate of generator point
int32_t mGysize	Size of y coordinate of generator point
void * pmInternalEC	Pointer to internal structure for handling the parameters

9.2.2 ECCfreeEC function

Table 99. ECCfreeEC function

Function name	ECCfreeEC
Prototype	`int32_t ECCfreeEC(EC_stt *P_pECctx, membuf_stt *P_pMemBuf)`
Behavior	De-initialize an EC_stt context
Parameter	– [in,out] *P_pECctx: Pointer to the EC_stt structure containing the curve parameters to be freed – [in,out] *P_pMemBuf :Pointer to the membuf_stt structure that holds the Ellitpic Curve internal values
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: P_pECctx == NULL – ECC_ERR_BAD_CONTEXT: Some values inside P_pECctx are invalid

9.2.3 ECCinitPoint function

Table 100. ECCinitPoint function

Function name	ECCinitPoint
Prototype	`int32_t ECCinitPoint(ECpoint_stt **P_ppECPnt, const EC_stt *P_pECctx, membuf_stt *P_pMemBuf)`
Behavior	Initialize an ECC point
Parameter	– [out] **P_ppECPnt: The point that will be initialized – [in] *P_pECctx: The EC_stt containing the Elliptic Curve Parameters – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that will be used to store the Ellitpic Curve Point internal values
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: P_pECctx == NULL – ECC_ERR_BAD_CONTEXT: Some values inside P_pECctx are invalid – ERR_DYNAMIC_ALLOCATION_FAILED: Not enough memory.

ECpoint_stt data structure

Object used to store an elliptic curve point. Should be allocated and unitized by ECCinitPoint and freed by ECCfreePoint

Table 101. ECpoint_stt data structure

Field name	Description
BigNum_stt * pmX	BigNum_stt integer for pmX coordinate.
BigNum_stt * pmY	BigNum_stt integer for pmY coordinate.
BigNum_stt * pmZ	BigNum_stt integer pmZ coordinate, used in projective representations.
ECPntFlags_et mFlag	– flag=CRL_EPOINT_GENERAL: point which may have pmZ not equal to 1 – flag=CRL_EPOINT_NORMALIZED: point which has pmZ equal to 1 – flag=CRL_EPOINT_INFINITY: to denote the infinity point

9.2.4 ECCfreePoint function

Table 102. ECCfreePoint function

Function name	**ECCfreePoint**
Prototype	`int32_t ECCfreePoint(ECpoint_stt **P_pECPnt, membuf_stt *P_pMemBuf)`
Behavior	Free Elliptic curve point
Parameters	[in] *P_pECPnt The point that will be freed [in,out] *P_pMemBuf Pointer to membuf_stt structure that stores Ellitpic Curve Point internal values
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER P_pECPnt == NULL || P_pMemBuf == NULL. – ECC_ERR_BAD_CONTEXT *P_pECPnt == NULL

9.2.5 ECCsetPointCoordinate function

Table 103. ECCsetPointCoordinate function

Function name	**ECCsetPointCoordinate**

Prototype	`int32_t ECCsetPointCoordinate (`
	`ECpoint_stt * P_pECPnt,`
	`ECcoordinate_et P_Coordinate,`
	`const uint8_t * P_pCoordinateValue,`
	`int32_t P_coordinateSize );`

Behavior	Set the value of one of coordinate of an ECC point
Parameter	– [in,out] *P_pECPnt: The ECC point that will have a coordinate set – [in] P_Coordinate: Flag used to select which coordinate must be set (see ECcoordinate_et) – [in] *P_pCoordinateValue: Pointer to an uint8_t array that contains the value to be set – [in] P_coordinateSize: The size in bytes of P_pCoordinateValue
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: One of the input parameters is invalid

9.2.6 ECCgetPointCoordinate function

Table 104. ECCgetPointCoordinate function

Function name	ECCgetPointCoordinate

Prototype	`int32_t ECCgetPointCoordinate (`
	`const ECpoint_stt * P_pECPnt,`
	`ECcoordinate_et P_Coordinate,`
	`uint8_t * P_pCoordinateValue,`
	`int32_t * P_pCoordinateSize );`

Behavior	Get the value of one of coordinate of an ECC point
Parameter	– [in] *P_pECPnt: The ECC point from which extract the coordinate – [in] P_Coordinate: Flag used to select which coordinate must be retrieved (see ECcoordinate_et) – [out] *P_pCoordinateValue: Pointer to an uint8_t array that will contain the returned coordinate – [out] *P_pCoordinateSize: Pointer to an integer that will contain the size of the returned coordinate
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: One of the input parameters is invalid

Note: The Coordinate size depends only on the size of the Prime (P) of the elliptic curve. Specifically if P_pECctx->mPsize is not a multiple of 4, then the size will be expanded to be a multiple of 4. In this case P_pCoordinateValue will contain one or more leading zeros.

9.2.7 ECCgetPointFlag function

Table 105. ECCgetPointFlag function

Function name	**ECCgetPointFlag**
Prototype	`int32_t ECCgetPointFlag(const ECpoint_stt *P_pECPnt)`
Behavior	Reads the flag member of an Elliptic Curve Point structure
Parameter	– [in] *P_pECPnt The point whose flag will be returned
Return value	– ECC_ERR_BAD_PARAMETER (P_pECPnt == NULL)

9.2.8 ECCsetPointFlag function

Table 106. ECCsetPointFlag function

Function name	**ECCsetPointFlag**
Prototype	`void ECCsetPointFlag(ECpoint_stt *P_pECPnt, ECPntFlags_et P_newFlag)`
Behavior	Set the flag member of an Elliptic Curve Point structure
Parameter	– [in,out] *P_pECPnt The point whose flag will be set – [out] P_newFlag The flag value to be set

9.2.9 ECCcopyPoint function

Table 107. ECCcopyPoint function

Function name	ECCcopyPoint
Prototype	`int32_t ECCcopyPoint ( const ECpoint_stt * P_pOriginalPoint, ECpoint_stt * P_pCopyPoint ) ;`
Behavior	Copy an Elliptic Curve Point
Parameter	– [in] *P_pOriginalPoint: The point that will be copied – [out] *P_pCopyPoint: The output copy of P_OriginalPoint
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: An input is invalid (i.e. NULL or not initialized with ECCinitPoint) – MATH_ERR_BIGNUM_OVERFLOW: P_pCopyPoint not initialized with correct P_pECctx

Note: Both points must be already initialized with ECCinitPoint

9.2.10 ECCinitPrivKey function

Table 108. ECCinitPrivKey function

Function name	ECCinitPrivKey(1)
Prototype	`int32_t ECCinitPrivKey(ECCprivKey_stt **P_ppECCprivKey, const EC_stt *P_pECctx, membuf_stt *P_pMemBuf)`
Behavior	Initialize an ECC private key
Parameters	– [out] **P_ppECCprivKey: Private key that will be initialized – [in] *P_pECctx: EC_stt containing the Elliptic Curve Parameters – [in,out] *P_pMemBuf: Pointer to membuf_stt structure that will be used to store the Ellitpic Curve Private Key internal value
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: P_pECctx == NULL – ECC_ERR_BAD_CONTEXT: Some values inside P_pECctx are invalid – ERR_MEMORY_FAIL: Not enough memory.

1. This function keeps values stored in membuf_stt.pmBuf, so when exiting this function membuf_stt.mUsed is greater than it was before the call. The memory is freed when ECCfreePrivKey is called.

ECCprivKey_stt data structure

Object used to store an ECC private key. Must be allocated and unitized by ECCinitPrivKey and freed by ECCfreePrivKey.

Table 109. ECCprivKey_stt data structure

Field name	Description
BigNum_stt * pmD	BigNum Representing the Private Key.

9.2.11 ECCfreePrivKey function

Table 110. ECCfreePrivKey function

Function name	ECCfreePrivKey
Prototype	`int32_t ECCfreePrivKey(ECCprivKey_stt **P_ppECCprivKey, membuf_stt *P_pMemBuf) ;`
Behavior	Free an ECC Private Key
Parameter	– [in,out] **P_ppECCprivKey The private key that will be freed – [in,out] *P_pMemBuf Pointer to the membuf_stt structure that currently stores the Ellitpic Curve Private Key internal value
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: P_ppECCprivKey == NULL || P_pMemBuf == NULL – ECC_ERR_BAD_PRIVATE_KEY: Private Key uninitalized

9.2.12 ECCsetPrivKeyValue function

Table 111. ECCsetPrivKeyValue function

Function name	ECCsetPrivKeyValue
Prototype	`int32_t ECCsetPrivKeyValue ( ECCprivKey_stt * P_pECCprivKey, const uint8_t * P_pPrivateKey, int32_t P_privateKeySize ) ;`
Behavior	Set the value of an ECC private key object from a byte array
Parameter	– [in,out] *P_pECCprivKey: The ECC private key object to set – [in] *P_pPrivateKey: Pointer to an uint8_t array that contains the value of the private key – [in] P_privateKeySize: The size in bytes of P_pPrivateKey
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: One of the input parameters is invalid – ECC_ERR_BAD_PRIVATE_KEY Private Key uninitalized

9.2.13 ECCgetPrivKeyValue function

Table 112. ECCgetPrivKeyValue function

Function name	ECCgetPrivKeyValue(1)

Prototype	`int32_t ECCgetPrivKeyValue(const ECCprivKey_stt *P_pECCprivKey,`
	`uint8_t *P_pPrivateKey,`
	`int32_t *P_pPrivateKeySize)`

Behavior	Get the private key value from an ECC private key object
Parameter	– [in] *P_pECCprivKey: The ECC private key object to be retrieved – [in] *P_pPrivateKey: Pointer to an uint8_t array that contains the value of the private key – [in] *P_privateKeySize: Pointer to an int that will contain the size in bytes of P_pPrivateKey
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: One of the input parameters is invalid

1. The Coordinate size depends only on the size of the Order (N) of the elliptic curve. Specifically if P_pECctx->mNsize is not a multiple of 4, then the size will be expanded to be a multiple of 4. In this case P_pPrivateKey will contain one or more leading zeros.

9.2.14 ECCscalarMul function

Table 113. ECCscalarMul function

Function name	ECCscalarMul

Prototype	`int32_t ECCscalarMul(const ECpoint_stt *P_pECbasePnt,`
	`const ECCprivKey_stt *P_pECCprivKey,`
	`ECpoint_stt *P_pECresultPnt,`
	`const EC_stt *P_pECctx,`
	`membuf_stt *P_pMemBuf)`

Behavior	Computes the point scalar multiplication kP = k*P
Parameter	– [in] *P_pECbasePnt: The point that will be multiplied – [in] *P_pECCprivKey: Structure containing the scalar value of the multiplication – [out] *P_pECresultPnt: The output point, result of the multiplication – [in] *P_pECctx: Structure describing the curve parameters – *P_pMemBuf: Pointer to the membuf_stt structure that currently stores the Ellitpic Curve Private Key internal value
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: One of the inputs == NULL – MATH_ERR_BIGNUM_OVERFLOW: The P_pCopyPoint was not initialized with the correct P_pECctx – ECC_ERR_BAD_CONTEXT: P_pECctx->pmInternalEC == NULL – ECC_WARN_POINT_AT_INFINITY: The returned point is the O point for the Elliptic Curve

9.2.15 ECCsetPointGenerator function

Table 114. ECCsetPointGenerator function

Function name	ECCsetPointGenerator
Prototype	`int32_t ECCsetPointGenerator(ECpoint_stt *P_pPoint, const EC_stt *P_pECctx)`
Behavior	Writes the Elliptic Curve Generator point into a ECpoint_stt
Parameter	– [out] *P_pPoint The point that will be set equal to the generator point – [in] *P_pECctx Structure describing the curve parameters, it must contain the generator point
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER One of the inputs == NULL – ECC_ERR_BAD_CONTEXT Some values inside P_pECctx are invalid (it doesn't contain the Generator) – MATH_ERR_BIGNUM_OVERFLOW The P_pPoint was not initialized with the correct P_pECctx

Note: P_pPoint must be already initialized with ECCinitPoint.

9.2.16 ECDSAinitSign function

Table 115. ECDSAinitSign function

Function name	ECDSAinitSign(1)
Prototype	`int32_t ECDSAinitSign(ECDSAsignature_stt **P_ppSignature, const EC_stt *P_pECctx, membuf_stt *P_pMemBuf)`
Behavior	Initialize an ECDSA signature structure
Parameter	– [out] **P_ppSignature Pointer to pointer to the ECDSA structure that will be allocated and initialized – [in] *P_pECctx The EC_stt containing the Elliptic Curve Parameters – *P_pMemBuf Pointer to the membuf_stt structure that will be used to store the ECDSA signatures internal values
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: Invalid Parameter – ECC_ERR_BAD_CONTEXT: Some values inside P_pECctx or P_pMemBuf are invalid – ERR_MEMORY_FAIL: Not enough memory.

1. This function keeps some value stored in membuf_stt.pmBuf, so when exiting this function membuf_stt.mUsed will be greater than it was before the call. The memory is freed when ECDSAfreeSign is called.

9.2.17 ECDSAfreeSign function

Table 116. ECDSAfreeSign function

Function name	ECDSAfreeSign

Prototype	`int32_t ECDSAfreeSign(`
	`ECDSAsignature_stt **P_ppSignature,`
	`membuf_stt *P_pMemBuf)`

Behavior	Free an ECDSA signature structure
Parameter	– [in,out] *P_pSignature: The ECDSA signature that will be freed – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that currently stores the ECDSA signature internal values
Return value	– ECC_SUCCESS Operation Successful – ECC_ERR_BAD_PARAMETER: P_pSignature == NULL || P_pMemBuf == NULL

9.2.18 ECDSAsetSignature function

Table 117. ECDSAsetSignature function

Function name	ECDSAsetSignature

Prototype	`int32_t ECDSAsetSignature (`
	`ECDSAsignature_stt * P_pSignature,`
	`ECDSAsignValues_et P_RorS,`
	`const uint8_t * P_pValue,`
	`int32_t P_valueSize ) ;`

Behavior	Set the value of the parameters (one at a time) of an ECDSAsignature_stt
Parameter	– [out] *P_pSignature: The ECDSA signature whose one of the value will be set – [in] P_RorS: Flag selects if the parameter R or the parameter S must be set – [in] *P_pValue: Pointer to an uint8_t array containing the signature value – [in] P_valueSize: Size of the signature value
Return value	– ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: One of the input parameters is invalid – MATH_ERR_BIGNUM_OVERFLOW: signature value passed is too big for the Signature structure

9.2.19 ECDSAgetSignature function

Table 118. ECDSAgetSignature function

Function name	ECDSAgetSignature(1)

Prototype	`int32_t ECDSAgetSignature (`
	`const ECDSAsignature_stt * P_pSignature,`
	`ECDSAsignValues_et P_RorS,`
	`uint8_t * P_pValue,`
	`int32_t * P_pValueSize) ;`

Behavior	Get the values of the parameters (one at a time) of an ECDSAsignature_stt

Table 118. ECDSAgetSignature function (continued)

Function name	**ECDSAgetSignature**[1]
Parameter	– [in] *P_pSignature: The ECDSA signature from which retrieve the value – [in] P_RorS: Flag selects if the parameter R or the parameter S must be returned – [out] *P_pValue: Pointer to an uint8_t array that will contain the value – [out] *P_pValueSize: Pointer to integer that contains the size of returned value
Return value	– ECC_SUCCESS: Operation Successful ECC_ERR_BAD_PARAMETER: One of the input parameters is invalid

1. The R or S size depends on the size of the Order (N) of the elliptic curve. Specifically if P_pECctx->mNsize is not a multiple of 4, then the size is expanded to be a multiple of 4. In this case P_pValue contains one or more leading zeros.

9.2.20 ECDSAverify function

Table 119. ECDSAverify function

Function name	**ECDSAverify**

Prototype	`int32_t ECDSAverify(const uint8_t *P_pDigest,`
	`int32_t P_digestSize,`
	`const ECDSAsignature_stt *P_pSignature,`
	`const ECDSAverifyCtx_stt *P_pVerifyCtx,`
	`membuf_stt *P_pMemBuf)`

Behavior	ECDSA signature verification with a digest input
Parameter	– [in] *P_pDigest: The digest of the signed message – [in] P_digestSize: The mSize in bytes of the digest – [in] *P_pSignature: The public key that will verify the signature – [in] *P_pVerifyCtx: The ECDSA signature that will be verified – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that will be used to store the internal values required by computation
Return value	– ERR_MEMORY_FAIL: There's not enough memory ECC_ERR_BAD_PARAMETER ECC_ERR_BAD_CONTEXT ECC_ERR_MISSING_EC_PARAMETER MATH_ERR_BIGNUM_OVERFLOW SIGNATURE_INVALID SIGNATURE_VALID

Note: This function requires that:
- P_pVerifyCtx.pmEC points to a valid and initialized EC_stt structure
- P_pVerifyCtx.pmPubKey points to a valid and initialized public key ECpoint_stt structure

9.2.21 ECCvalidatePubKey function

Table 120. ECCvalidatePubKey function

Function name	ECCvalidatePubKey(1)
Prototype	`int32_t ECCvalidatePubKey(const ECpoint_stt *P_pECCpubKey, const EC_stt *P_pECCctx, membuf_stt *P_pMemBuf)`
Behavior	Checks the validity of a public key.
Parameter	– [in] *pECCpubKey: The public key to be checked – [in] *P_pECCctx: Structure describing the curve parameters – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that will be used to store the internal values required by computation
Return value	– ECC_SUCCESS: pECCpubKey is a valid point of the curve – ECC_ERR_BAD_PUBLIC_KEY: pECCpubKey is not a valid point of the curve – ECC_ERR_BAD_PARAMETER: One of the input parameter is NULL – ECC_ERR_BAD_CONTEXT One of the values inside P_pECCctx is invalid – ERR_MEMORY_FAIL: Not enough memory.

1. This function does not check that PubKey * group_order == infinity_point. This is correct assuming that the curve's cofactor is 1.

9.2.22 ECCkeyGen function

Table 121. ECCkeyGen function

Function name	ECCkeyGen

Prototype	`int32_t ECCkeyGen(ECCprivKey_stt *P_pPrivKey,`
	`ECpoint_stt *P_pPubKey,`
	`RNGstate_stt *P_pRandomState,`
	`const EC_stt *P_pECCctx,`
	`membuf_stt *P_pMemBuf)`

Behavior	Generate an ECC key pair.
Parameters	– [out] *P_pPrivKey: Initialized object that will contain the generated private key – [out] *P_pPubKey: Initialized point that will contain the generated public key – [in] *P_pRandomState: The random engine current state – [in] *P_pECCctx: Structure describing the curve parameters. This must contain the values of the generator – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that will be used to store the internal values required by computation
Return value	– ECC_SUCCESS: Key Pair generated Successfully – ERR_MEMORY_FAIL: There's not enough memory – ECC_ERR_BAD_PARAMETER: One of input parameters is not valid – RNG_ERR_UNINIT_STATE: Random engine not initialized. – ECC_ERR_MISSING_EC_PARAMETER: P_pECCctx must contain a, p, n, Gx,Gy – MATH_ERR_BIGNUM_OVERFLOW: P_pPubKey was not properly initialized

Note: *P_pPrivKey and P_pPubKey must be already initialized with respectively ECCinitPrivKey and ECCinitPoint P_pECCtx must contain the value of the curve's generator.*

9.2.23 ECDSAsign function

Table 122. ECDSAsign function

Function name	**ECDSAsign**

Prototype	`int32_t ECDSAsign(const uint8_t *P_pDigest,`
	`int32_t P_digestSize,`
	`const ECDSAsignature_stt *P_pSignature,`
	`const ECDSAsignCtx_stt *P_pSignCtx,`
	`membuf_stt *P_pMemBuf)`

Behavior	ECDSA Signature Generation.
Parameter	– [in] *P_pDigest: The message digest that will be signed – [in] P_digestSize: The size in bytes of the P_pDigest – [out] *P_pSignature: Pointer to an initialized signature structure that will be contain the result of the operaion – [in] *P_pSignCtx: Pointer to an initialized ECDSAsignCtx_stt structure – [in,out] *P_pMemBuf Pointer to the membuf_stt structure that will be used to store the internal values required by computation
Return value	– ECC_SUCCESS: Key Pair generated Successfully – ERR_MEMORY_FAIL: There's not enough memory – ECC_ERR_BAD_PARAMETER: One of input parameters is not valid – RNG_ERR_UNINIT_STATE: Random engine not initialized. – MATH_ERR_BIGNUM_OVERFLOW: P_pPubKey was not properly initialized – ECC_ERR_BAD_CONTEXT: Some values inside P_pSignCtx are invalid – ECC_ERR_MISSING_EC_PARAMETER: P_pSignCtx must contain a, p, n, Gx, Gy

Note: This function requires that:

- P_pSignCtx.pmEC points to a valid and initialized EC_stt structure
- P_pSignCtx.pmPrivKey points to a valid and initialized private key ECCprivKey_stt structure
- P_pSignCtx.pmRNG points to a valid and initialized Random State RNGstate_stt structure

9.3 ECC example

```
/* Initialize the EC_stt structure with the known values. We also initialize
to NULL and zero the unknown parameter */
   ECparams.mAsize = sizeof(ecc_160_a);
   ECparams.pmA = ecc_160_a;
   ECparams.mPsize = sizeof(ecc_160_p);
   ECparams.pmP = ecc_160_p;
   ECparams.pmN = ecc_160_n;
   ECparams.mNsize = sizeof(ecc_160_n);
   ECparams.pmB = NULL;
   ECparams.mBsize = 0;
   ECparams.pmGx = NULL;
   ECparams.mGxsize = 0;
   ECparams.pmGy = NULL;
   ECparams.mGysize = 0;

   /* Call the Elliptic Curve initialization function */
   retval = ECCinitEC(&ECparams);
   if (retval != 0)
   {
     printf("Error! ECCinitEC returned %d\n", retval);
     return(-1);
   }
   /* Initialize the point that will contain the generator point */
   retval = ECCinitPoint(&G, &ECparams);
   if (retval != 0)
   {
     printf("Error! ECCinitPoint returned %d\n", retval);
     return(-1);
   }

   /* Set the coordinates of the generator point inside G */
   rertval = ECCsetPointGenerator(G, &ECparams);
   if (retval != 0)
   {
     printf("Error! ECCsetPointGenerator returned %d\n", retval);
     return(-1);
   }

   /* Init the point the will keep the result of the scalar multiplication
*/
   retval = ECCinitPoint(&PubKey, &ECparams);
   if (retval != 0)
   {
     printf("Error! ECCinitPoint returned %d\n", retval);
```

```
      return(-1);
    }

    /* Initialize the private key object */
    retval = ECCinitPrivKey(&privkey, &ECparams);
    if (retval != 0)
    {
      printf("Error! ECCinitPrivKey returned %d\n", retval);
      return(-1);
    }
    /* Set the private key object */
    retval = ECCsetPrivKeyValue(privkey, ecc_160_privkey,
sizeof(ecc_160_privkey));
    if (retval != 0)
    {
      printf("Error! ECCsetPrivKeyValue returned %d\n", retval);
      return(-1);
    }

    /* All ECCscalarMul parameters are initalized and set, proceed. */
    retval = ECCscalarMul(G, privkey, PubKey, &ECparams);
    if (retval != 0 )
    {
      printf("ECCscalarMul returned %d\n",retval);
      return(-1);
    }
    /* Now PubKey contains the result point, we can get its coordinates
through */
    ECCgetPointCoordinate(KP, E_ECC_POINT_COORDINATE_X, pubKeyX, &Xsize);
    ECCgetPointCoordinate(KP, E_ECC_POINT_COORDINATE_Y, pubKeyY, &Ysize);

    /* Finally we free everything we initialized */
    ECCfreePrivKey(privkey);
    ECCfreePoint(G);
    ECCfreePoint(PubKey);
    ECCfreeEC(&ECparams);
  }
```

10 STM32 encryption library settings

The flexibility of the Cryptographic library allows the user to select just the algorithm and the modes needed, and the necessary object code will be generated. Customization leads to a very small code size.

10.1 Configuration parameters

Table 123 describes the configuration parameters used to build the STM32 cryprtographic library. These parameters are defined in the file inc\config.h

Table 123. Library build options

Configuration type	Configuration parameter name	Description
Endianness	`CRL_ENDIANNESS = 1`	Specifies the memory representation of the platform: – CRL_ENDIANNESS = 1 for LITTLE ENDIAN – CRL_ENDIANNESS = 2 for BIG ENDIAN
MISALIGNED read/write operations	`CRL_CPU_SUPPORT_MISALIGNED`	When set to 1 this flag improves the performance of AES when used through high level functions.
Encryption/ Decryption capability	`INCLUDE_ENCRYPTION`	Includes the Encryption functionalities. Remove it if only decryption is needed
	`INCLUDE_DECRYPTION`	Includes the Decryption functionalities. Remove it if only encryption is needed
Symmetric Key Algorithms	`INCLUDE_DES`	Permits DES functions in the library
	`INCLUDE_TDES`	Permits TripleDES (TDES) functions in the library
	`INCLUDE_AES128`	Permits AES functions with key size of 128 bits in the library, if it's NOT defined then aes128.c is not needed
	`INCLUDE_AES192`	Permits AES functions with key size of 192 bits in the library. If it's NOT defined then aes192.c is not needed
	`INCLUDE_AES256`	Permits AES functions with key size of 256 bits in the library. If it's NOT defined then aes256.c is not needed
	`INCLUDE_ARC4`	Enables the ARC4 algorithm
Symmetric Key Modes of operations	`INCLUDE_ECB`	Enables AES high level functions for ECB mode are included in the library
	`INCLUDE_CBC`	Enables AES high level functions for CBC mode in the library
	`INCLUDE_CTR`	Enables AES high level functions for CTR mode in the library
	`INCLUDE_GCM`	Enables AES high level functions for GCM mode in the library
	`INCLUDE_KEY_WRAP`	Enables AES-KWRAP function in the library
	`INCLUDE_CCM`	Enables AES-CCM function in the library
	`INCLUDE_CMAC`	Enables AES-CMAC function in the library

Table 123. Library build options (continued)

Configuration type	Configuration parameter name	Description
Public Key Algorithms	`INCLUDE_RSA`	Enables RSA functions for signature generation/validation in the library
	`INCLUDE_ECC`	Enables RSA functions
HASH Algorithms	`INCLUDE_MD5`	Permits MD5 functions in the library
	`INCLUDE_SHA1`	Permits SHA-1 functions in the library
	`INCLUDE_SHA224`	Permits SHA-224 functions in the library
	`INCLUDE_SHA256`	Permits SHA-256 functions in the library
	`INCLUDE_HMAC`	Enables HMAC for the selected hash algorithms
Deterministic Random Bit Generator	`INCLUDE_DRBG_AES128`	Enables the **D**eterministic **R**andom **B**it **G**enerator (**DRBG**) feature. **Requires AES128 with Encryption capabilities**
	`CRL_RANDOM_REQUIRE_RESEED`	CRL_RANDOM_REQUIRE_RESEED implements the request for reseed when using the DRBG too many times for security standards
AES Algorithm version	`CRL_AES_ALGORITHM = 1`	Selects the AES algorithm version with 522 bytes of look-up tables, slower than version 2
	`CRL_AES_ALGORITHM = 2`	Selects the AES algorithm version with 2048 bytes of look-up tables, faster than version 1.
RSA Window size	`RSA_WINDOW_SIZE = 4`	Speeds up RSA operation with private key at expense of RAM memory. It can't be less than one, and memory grows according to the formula: **MemoryRequired = 2^(RSA_WINDOW_SIZE - 1) * (20 + RSAKeySizeInBytes)** Suggested values are 3 or 4. Entering a value of 7 or more will be probably worst than using 6.
AES GCM GF(2^128) Table Precomputations	`CRL_GFMUL = 2`	Specifies algorithm used for polynomial multiplication in AES-GCM. This also defines the size of the precomputed table made to speed up the multiplication. There are two types of table, one is based on the value of the key and so needs to be generated at running (through AES_GCM_keyschedule), the other is constant and is defined (if included here) in privkey.h. There are 3 possible choices: – 0 = Without any tables. No space required. Slower version. – 1 = Key-dependent table for *Poly(y) 0000<y<1111 and constant table for *x^4(256 key-dependent bytes - 32 constant bytes). – 2 = 4 key-dependent tables for *Poly(y^(2^(32*i))) and 4 key-dependent tables for *Poly((y*x^4)^(2^(32*i))) with 0000<y<1111 and 0<i<4 and constant tables for *x^8 and for *x^4 (2048 key-dependent bytes - 544 constant bytes).

10.2 STM32_CryptoLibraryVersion

To get information about the STM32 Cryptographic Library setting and version, call the STM32_CryptoLibraryVersion() function in the application layer.

Table 124. STM32_CryptoLibraryVersion

Function name	STM32_CryptoLibraryVersion
Prototype	`void TM32_CryptoLibraryVersion( STM32CryptoLibVer_TypeDef * LibVersion)`
Behavior	Get the `STM32 Cryptographic Library setting`
Parameter	– [in,out] *STM32CryptoLibVer_TypeDef: Pointer to structure that will be used to store the internal library setting
Return value	– None

11 Cryptographic library performance and memory requirements

This section provides a performance evaluation of the cryptographic library for the STM32 microcontroller series. In particular this analysis targets the STM32F4xx family, as the series STM32F41x includes some cryptographic accelerators, specifically it includes:

- One CRYP Accelerator, capable of encryption/decryption with:
  - AES in ECB, CBC, CTR and KEYWRAP with all three key sizes (128, 192, 256 bit)
  - DES and TDES in ECB and CBC
- One HASH Accelerator, capable of MD5 and SHA-1 HASH and HMAC operations
- One RNG (Random Number Generator)

The tests were conducted on STM32F41x with CPU running at a frequency of 168 MHz and using RealView Microcontroller Development Kit (MDK-ARM) toolchain V4.70 ST-Link.

11.1 Symmetric key algorithms performance results

In this section we provide performance results for:

- DES in ECB and CBC
- TDES in ECB and CBC
- AES-128 in ECB, CBC and CTR and CMAC modes.
- AES-192 in ECB, CBC and CTR and CMAC modes.
- AES-256 in ECB, CBC and CTR and CMAC modes.
- ARC4

AES modes CTR and CMAC do not have a proper decryption mode like ARC4. In these cases decryption works exactly like encryption.

To calculate the number of cycles needed to perform each operation mode:
Cycles = Init key cycle + Init message cycle + Process block of data cycle * number of blocks

The code size required by these algorithms is shown in *Table 126 on page 120*.

11.1.1 Software optimized for speed

Table 125 shows the clock cycles needed by each algorithm to process a block of data.

Table 125. Performance of symmetric key encryption algo. optimized for speed

Algorithm mode	Operation	Init key	Init message	Process block of data(1)
DES-ECB	Encryption	19 539	205	1 553
	Decryption	19 542	219	1 554
DES-CBC	Encryption	19 548	390	1 556
	Decryption	19 548	402	1 578
TDES-ECB	Encryption	58 638	215	4 569
	Decryption	58 629	200	4 565
TDES-CBC	Encryption	58 650	469	4 569
	Decryption	58 650	395	4 587
AES-128-CBC	Encryption	639	622	1 622
	Decryption	2 928	630	1 644
AES-192-ECB	Encryption	630	316	1 885
	Decryption	3 411	311	1 936
AES-192-CBC	Encryption	636	735	1 909
	Decryption	3 432	702	1 975
AES-256-ECB	Encryption	837	340	2 183
	Decryption	4 131	316	2 204
AES-256-CBC	Encryption	843	632	2 180
	Decryption	4 155	694	2 243
AES-128-CTR	Encryption	624	673	1 628
	Decryption	621	689	1 628
AES-128-CMAC	Encryption	636/	639	1 575
	Decryption	618	525	1 575
AES-192-CTR	Encryption	621	676	1 911
	Decryption	618	691	1 911
AES-192-CMAC	Encryption	632	719	1 859
	Decryption	616	608	1 859
AES-256-CTR	Encryption	828	730	2 180
	Decryption	825	746	2 180
AES-256-CMAC	Encryption	840	758	2 141
	Decryption	816	649	2 141
ARC4	Encryption	0	6 059	25
	Decryption	0	6 059	25

1. Block of data represent :8 bytes for DES and TDES, 16 for AES, 1 for ARC4

Table 126. Code size required by symmetric key encryption algo

Algorithm mode	Code size (byte)	Constant data size (byte)
DES/TDES ECB,CBC	3 842	6 040
AES(128,192,256) ECB,CBC	8 068	6 040
AES(128,192,256) CTR	4 896	6 040
AES(128,192,256) CMAC	5 796	6 040
ARC4	686	0

11.1.2 Hardware enhanced

Table 127 shows the performance calculated for symmetric key encryption algorithms with hardware acceleration. The code size required by these algorithms is shown in *Table 128*. All AES modes except CMAC are shown as associated because the hardware supports all of them, so removing one would not significantly decrease the code size.

Table 127. Symmetric key encrypt. algo. performance with HW acceleration

Algorithm mode	Operation	Init key	Init message	Process block of data[(1)]
DES-ECB	Encryption	0	601	28
	Decryption	0	607	28
DES-CBC	Encryption	0	799	28
	Decryption	0	787	28
TDES-ECB	Encryption	0	616	59
	Decryption	0	631	59
TDES-CBC	Encryption	0	818	59
	Decryption	0	817	59
AES-128-ECB	Encryption	0	702	34
	Decryption	0	819	34
AES-128-CBC	Encryption	0	1 170	34
	Decryption	0	1 281	34
AES-192-ECB	Encryption	0	726	34
	Decryption	0	849	34
AES-192-CBC	Encryption	0	1 197	34
	Decryption	0	1 311	34
AES-256-ECB	Encryption	0	728	34
	Decryption	0	854	34
AES-256-CBC	Encryption	0	1 205	34
	Decryption	0	1322	34
AES-128-CTR	Encryption	0	1 085	34
AES-128-CTR	Decryption	0	1 107	34
AES-128-CMAC	Encryption	0	1 079	128
AES-128-CMAC	Decryption	0	982	128
AES-192-CTR	Encryption	0	1 112/	34
AES-192-CTR	Decryption	0	1 131	34
AES-192-CMAC	Encryption	0	1 104	128
AES-192-CMAC	Decryption	0	1 002	128
AES-256-CTR	Encryption	0	1 120	34
AES-256-CTR	Decryption	0	1 142	34
AES-256-CMAC	Encryption	0	1 096	128
AES-256-CMAC	Decryption	0	1 002	128

1. Block of data represent: 8 bytes for DES and TDES, 16 for AES, 1 for ARC4

Table 128. Code size for symmetric key encryption algo. with HW acceleration

Algorithm mode	Code size (byte)	Constant data size (byte)
DES/TDES ECB,CBC	1 984	0
AES(128,192,256) ECB,CBC,CTR	2 868	0
AES(128,192,256) CMAC	2 244	0

11.2 Authenticated encryption algorithms performance results

11.2.1 Software optimized for speed

Below are the required clock cycles for each mode and key length.

Table 129. Clock cycles for authenticated encryption algorithms optimized for speed

Algorithm mode	Operation	Init key	Init message	Block of header (16 bytes)	Block of payload (16 bytes)
AES-128-GCM	Encryption	12 570	3 368	1 314	3 043
	Decryption	12 570	3 410	1 314	3071
AES-192-GCM	Encryption	12 762	3 692	1 314	3 318
	Decryption	12 762	3 795	1 314	3 345
AES-256-GCM	Encryption	13 245	4 092	1 315	3 607
	Decryption	13 248	4 120	1 315	3 634
AES-128-CCM	Encryption	606	4 167	1 585	3 158
	Decryption	621	4 070	1 585	3 136
AES-192-CCM	Encryption	600	4 875	1 871	3 724
	Decryption	609	4 727	1 871	3 699
AES-256-CCM	Encryption	807	5 509	2 157	4 289
	Decryption	819	5 245	2 157	4 270

To process a message of 16 bytes of header and 32 bytes of payload with AES-128 in GCM mode and software optimized for speed, would require: 12 570 + 3 368 + 1 314 x 1 + 3 043 x 2 = 23 338 clock cycles

The required sizes for the algorithms are shown below. The Context size is the amount of RAM memory required to store a context of the Mode. It is listed here because in the case of GCM, the amount is significant.

Table 130. Code size for authenticated encryption algorithms optimized for speed

Algorithm mode	Code size (byte)	Constant data size (byte)	Context size (byte)
AES(128,192,256) CCM	6502	6040	332
AES(128,192,256) GCM	6488	6040	2360

11.2.2 Hardware enhanced

For each version of algorithm with hardware acceleration, *Table 131* shows the clock cycles required for each operation.

Table 131. Clock cycles for authenticated encryption algorithms & HW acceleration

Algorithm mode	Operation	Init Key	Init Message	Block of header (16 bytes)	Block of Payload (16 bytes)
AES-128-GCM	Encryption	10 374	2 417	1 314	1 468
	Decryption	10 374	2 454	1 314	1 492
AES-192-GCM	Encryption	10 314	2 434	1 314	1 469
	Decryption	10 314	2 482	1 314	1 492
AES-256-GCM	Encryption	10 302	2 437	1 315	1 468
	Decryption	10 299	2 479	1 315	1 492
AES-128-CCM	Encryption	0	1 543	136	260
	Decryption	0	1 356	136	239
AES-192-CCM	Encryption	0	1 573	136	260
	Decryption	0	1 386	136	239
AES-256-CCM	Encryption	0	1 564	136	260
	Decryption	0	1 380	136	239

Table 132 shows the required sizes for the algorithms. The Context size is the amount of RAM memory required to store a context of the Mode. It is listed here because in the case of GCM the amount is significant.

Table 132. Code size for authenticated encryption algorithm & HW acceleration

Algorithm mode	Code size (byte)	Constant data size (byte)	Context size (byte)
AES-128-192-256-GCM	3 538	880	2360
AES-128-192-256-CCM	3 886	0	332

11.3 AES key wrap results

11.3.1 Software optimized for speed

Table 133 shows the results of AES Key Wrap/Unwrap using all the three AES supported key sizes for software optimized for speed.

Table 133. AES Key Wrap/Unwrap in software

Algorithm	Mode	Wrapping 128 bits	Wrapping 192 bits	Wrapping 256 bits
AES-128-KW	Key wrap	23 976	27 537	31 083
	Key unwrap	23 364	27 456	31 485
AES-192-KW	Key wrap	--	38 484	43 761
	Key unwrap	--	38 685	44 334
AES-256-KW	Key wrap	--	--	56 511
	Key unwrap	--	--	56 976

Table 134. Code size for AES key wrap/unwrap in software(1)

Algorithm	Code size (byte)	Constant data size (byte)
AES-128-KW	7 274	6 040
AES-192-KW	7 274	6 040
AES-256-KW	7 274	6 040

1. Note that Key Wrap needs to allocate a memory whose size is equal to the input size plus 8 bytes.

11.3.2 Hardware enhanced

Table 135. AES key wrap/unwrap with HW acceleration

Algorithm	Mode	Wrapping 128 bits	Wrapping 192 bits	Wrapping 256 bits
AES-128-KW	Key wrap	6 462	6 489	6 492
	Key unwrap	3 486	3 519	3 522
AES-192-KW	Key wrap	--	7 143	7 146
	Key unwrap	--	4 134	4 137
AES-256-KW	Key wrap	--	--	7 800
	Key unwrap	--	--	4 752

Table 136. Code size for AES key wrap/unwrap with HW acceleration(1)

Algorithm	Code size (byte)	Constant data size (byte)
AES-128-192-256-KW	1578	0

1. Note that Key Wrap, even with HW acceleration, needs to allocate a memory whose size is equal to the input size plus 8 bytes.

11.4 HASH and HMAC algorithm results

11.4.1 Software optimized for speed

Table 137. Clock cycles for HASH and HMAC algorithms optimized for speed

Algorithm	Init message	Block of data (64 bytes)	Finalization
MD5	175	909	1 608
SHA-1	250	2 466	3 063
SHA-224	230	3 352	3 906
SHA-256	210	3 352	3 948
HMAC-MD5	2 001	909	4 344
HMAC-SHA-1	3 813	2 466	8 823
HMAC-SHA-224	4 708	3 352	11 340
HMAC-SHA-256	4 789	3 352	11 403

Table 138 shows the required sizes for the algorithms. SHA-224 and SHA-256 are shown together because they share the same core function, thus leaving only one of them provide just a small improvement in code size reduction.

Table 138. Clock cycles for HASH and HMAC algorithms with SW acceleration

Algorithm	Code size (byte)	Constant data size (byte)
MD5	2684	6040
SHA-1	1692	
SHA-224, SHA-256	2098	6040
MD5, HMAC-MD5	3264	6040
SHA-1, HMAC-SHA-1	2208	0
SHA-224, SHA-256, HMAC-SHA-224, HMAC-SHA-256	3485	6040

11.4.2 Hardware enhanced

Table 139. Clock cycles required by HASH/HMAC algorithms with HW acceleration

Algorithm	Init message	Block of data (64 bytes)	Finalization
MD5	330	119	374
SHA-1	308	135	419
HMAC-MD5	496	119	596
HMAC-SHA-1	489	135	686

Table 140. Code size required by HASH/HMAC algorithms

Algorithm	Code size (byte)	Constant data size (byte)
MD5, SHA-1	1166	0
SSHA-224 + SHA-256	2098	880
MD5, SHA-1, HMAC-MD5, HMAC-SHA-1	2282	0
SHA-224, SHA-256, HMAC-SHA-224, HMAC-SHA-256	3458	880

11.5 RSA results

RSA operates with different key sizes, and different exponents. The time required by the operation depends on these values.

In this section, we provide the results for the three most common public key exponents, which are 3, 17 and 65537. Considered key sizes are 1024 and 2048 bit.

The following table shows RSA algorithm performance with speed optimization.

Table 141. RSA performance with optimization for speed

Key size	Exponent	Clock cycles
1024	3	1 213 793
1024	17	1 284 982
1024	65537	1 573 079
1024	Private Key	30 627 432
2048	3	4 839 035
2048	17	5 109 399
2048	65537	6 195 481
2048	Private Key	228 068 226

The following table shows the required code size and heap, note that dynamically allocated memory is a requirement, because the private key operation is optimized with pre-calculations, which impacts performance and heap usage.

Code size is independent from the key size or the exponent used.

Table 142. Code size required by RSA algorithms

Key Size	Code size (byte)	Constant data size (byte)	Heap size (byte)
1024	6654	0	2132
2048	6654	0	4052

11.6 ECC results

Table 143 shows required clock cycles for ECC operations executed on all the NIST approved prime curves. The results are provided for software compiled with speed optimization.

Table 143. Number of cycles for ECC operations with for speed optimization

Operations	ECC-192	ECC-224	ECC-256	ECC-384	ECC-521
Init Key Generation	7 400 421	9 849 334	12 713 277	29 180 298	62 531 611
Signanture	7 720 020	10 414 487	13 102 239	29 673 252	64 664 144
Verification	14 716 374	19 558 528	24 702 099	58 986 725	124 393 892

Table 144 shows the required code size and heap memory (includes DRBG-AES-128, required for ECDSA Signature Generation). This data groups together all three functionalities and the required DRBG.

Code size is independent from the key size or the exponent used.

Table 144. Code size for ECC operations with speed optimization

Curve	Code size (byte)	Constant data size (byte)	Heap size (byte)
ECC-192	15960	6040	1424
ECC-224	15960	6040	1564
ECC-256	15960	6040	1704
ECC-384	15960	6040	2264
ECC-521	15960	6040	2964

12 Revision history

Table 145. Document revision history

Date	Revision	Changes
13-Oct-2008	1	Initial release.
11-Jul-2011	2	Added support for new algorithms. Added support for STM32F1, F2 and L1.
23-Aug-2013	3	Added support for STM32F4, F0 and F3.
13-Sep-2013	4	Publishing scope changed to Public. Added part number STM32-CRYP-LIB.