UM1924
User manual

Legacy STM32 cryptographic library

Introduction

The legacy STM32 cryptographic library package (X-CUBE-CRYPTO-V3) is no more maintained.

Refer to new STM32 cryptographic library package (www.st.com/en/product/x-cube-cryptolib) to have an up to date version of the package, supporting all STM32 Series

This user manual describes the APIs of the legacy STM32 cryptographic library (X-CUBE-CRYPTO-V3) that supports the following cryptographic algorithms:

- AES-128, AES-192, AES-256 bits, supporting the following modes
  - ECB, CBC, CTR, CFB, OFB, CCM, GCM, CMAC, KEY WRAP and XTS
- ARC4
- DES, TripleDES, supporting the following modes:
  - ECB, CBC
- HASH functions with HMAC, supporting the following modes:
  - MD5, SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512
- ChaCha20
- Poly1305
- CHACHA20-POLY1305
- Random engine based on DRBG-AES-128
- RSA with PKCS#1v1.5 for:
  - Signature/verification,
  - Encryption/decryption
- ECC (elliptic curve cryptography):
  - Key generation, scalar multiplication (the base for ECDH) and ECDSA
- ED25519
- Curve25519

These cryptographic algorithms run in all STM32 Series with the firmware implementation. For dedicated devices some algorithms are supported with hardware acceleration to optimize the performance and the footprint usage.

The legacy STM32 cryptographic library software is classified ECCN 5D002.

Contents

List of tables

List of figures

1 Acronyms and definitions

This document applies to Arm®-based devices.

arm

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Table 1. List of terms

Acronym	Definition
AEAD	Authenticated encryption with associated data
AES	Advanced encryption standard
ARC4	Alleged RC4
CBC	Cipher block chaining
CCM	Counter with CBC-MAC
CFB	Cipher feedback
CMAC	Cipher-based message authentication code
CTR	Counter
DES	Data encryption standard
ECB	Electronic codebook
ECC	Elliptic curve cryptography
ECDSA	Elliptic curve digital signature algorithm
FIPS	Federal information processing standard
GCM	Galois/counter mode
HMAC	keyed-hash message authentication code
HKDF	Hash-based key derivation function
MAC	Message authentication code
MD5	Message digest 5
NIST	National institute of standards and technology
OFB	Output feedback
RNG	Random number generation
RSA	Ronald Rivest, Adi Shamir and Leonard Adleman
SHA	Secure Hash algorithm
TDES	Triple DES
XTS	XEX-based tweakable codebook mode with cipher-text stealing

2 Terminology

Cryptography is a fundamental block for implementing information security. It deals with algorithms that process data in order to grant certain properties, depending on the application needs:

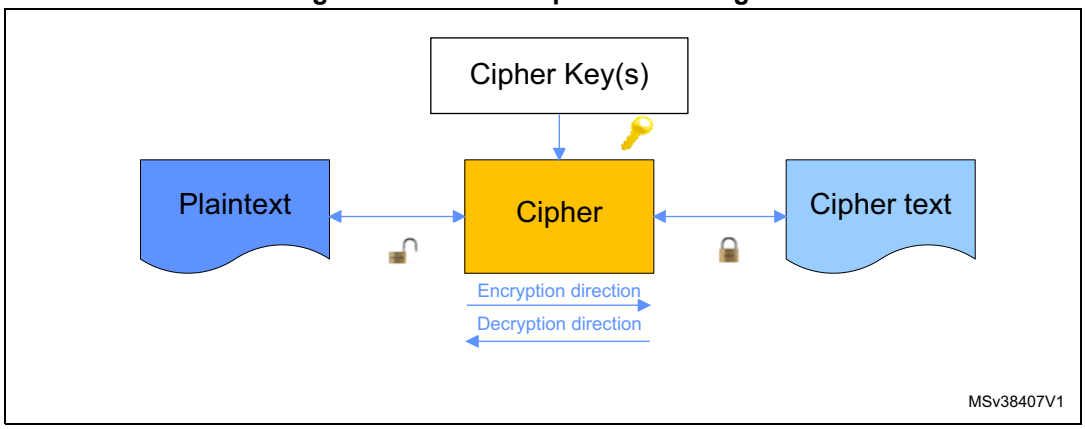
1. Data integrity services address the unauthorized or accidental modification of data. This includes data insertion, deletion, and modification. To ensure data integrity, a system must be able to detect unauthorized data modification. The goal is for the receiver of the data to verify that the data has not been altered.
2. Confidentiality services restrict access to the content of sensitive data to only those individuals who are authorized to view the data. Confidentiality measures prevent the unauthorized disclosure of information to unauthorized individuals or processes.
3. Identification and authentication services establish the validity of a transmission, message, and its originator. The goal is for the receiver of the data to determine its origin.
4. Non-repudiation services prevent an individual from denying that previous actions had been performed. The goal is to ensure that the recipient of the data is assured of the sender's identity.

These services can be used by one of the below cryptography branches:

1. **Encryption** is a branch of cryptography science. It converts clear-data / raw-data, called *Plaintext*, to unreadable data, called *Cipher-text*, using secret key(s), called *Cipher key(s)*, to perform cryptographic operations.

   The cipher text can be decrypted into human readable (clear text) form only using a secret key.

Figure 1. Common cipher block diagram



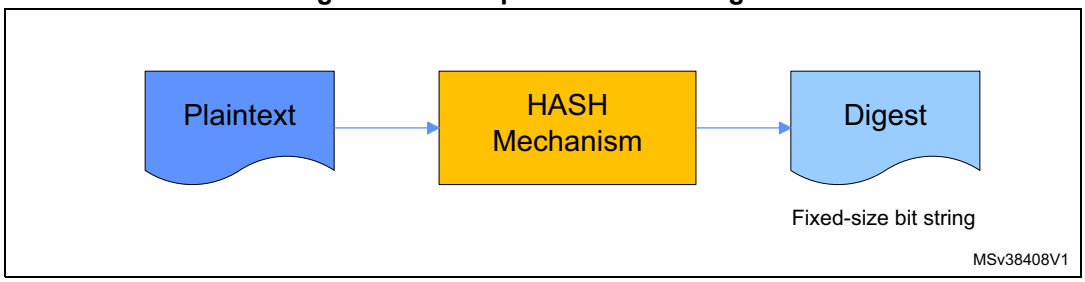
This operation can be based on:

a) symmetric cipher: cipher that uses a single key for encryption and decryption;

b) asymmetric cipher: cipher that uses two keys, one for encryption and the other for decryption.

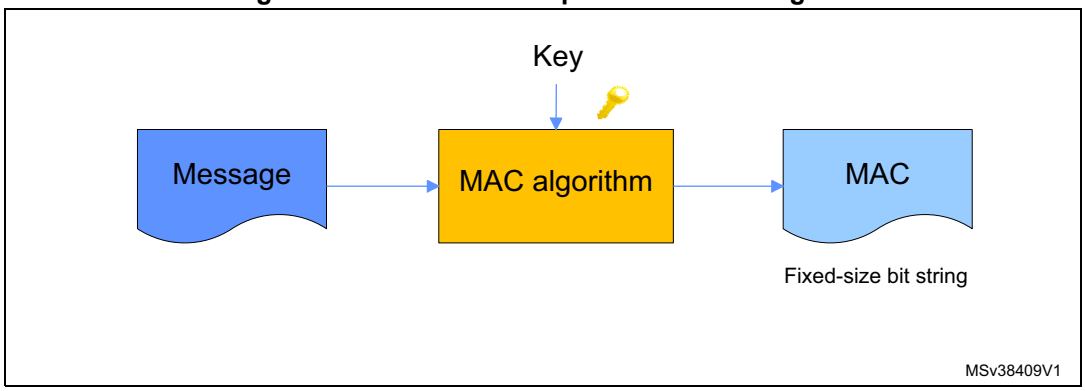
2. A cryptographic **Hash function** is a type of one-way function with additional properties. This deterministic algorithm takes an input data block of arbitrary size and returns a fixed size bit string, called digest. Hash functions are built in such a way that it is computationally infeasible to:
   a) find the data value given the “digest” of an unknown data,
   b) find two messages that hash to the same digest.

Figure 2. Hash operation block diagram



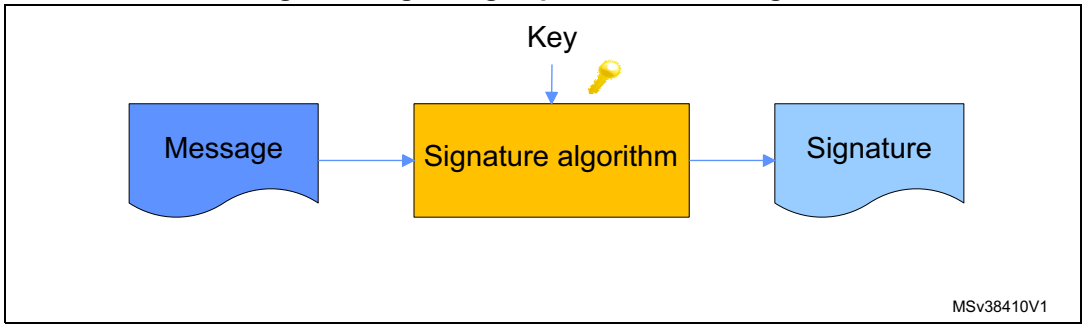
3. A **MAC (message authentication code)** requires two inputs: a message and a secret key known only by the originator of the message and its intended recipient(s). This allows the recipient(s) of the message to verify its integrity and ensure that the message’s sender has the shared secret key. If a sender doesn’t know the secret key, the hash value would then be different, allowing the recipient to understand that the message was not from the original sender.

Figure 3. Authentication operation block diagram



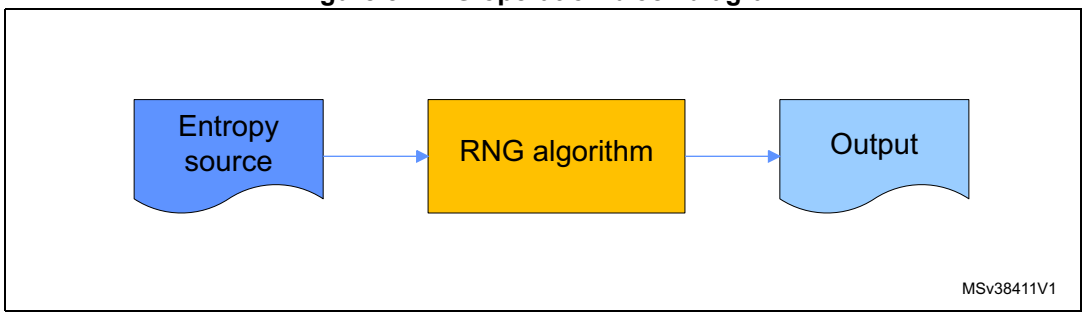
4. **Digital signature** is a mechanism by which a message is authenticated, i.e. proving that a message is effectively coming from a given sender, pretty much like a signature on a paper document. This is based on asymmetric cryptography: the private key is used to sign message, while only the public key is required to verify signatures. Therefore only the owner of the private key can compute signatures, while several other parties can verify them.

Figure 4. Digital sign operation block diagram



5. **RNG**: as listed above, the secure aspect of cryptography algorithms is based on the impossibility of guessing the key. Therefore, the key must be random. To perform this generation an RNG algorithm can be used: it receives as input a short entropy string and produces a cryptographically strong random output of much larger size.

Figure 5. RNG operation block diagram



Conclusion

Each of the above branches of the cryptography standard needs to be used according to the application level requirement. *Table 2* summarizes cryptography branch application.

Table 2. Summary of cryptography branch applications

Application	Data integrity	Confidentiality	Identification and authentication	Non-repudiation
Symmetric key encryption	NO	YES	NO	NO
Secure Hash Functions	YES	NO	NO	NO
MAC	YES	NO	YES	NO
Digital signatures	YES	NO	YES	YES

3 Legacy STM32 cryptographic library package presentation

The cryptographic library package includes a set of cryptographic algorithms based on:

1. firmware implementation ready to use in all STM32 Series
2. hardware acceleration for some/dedicated STM32 Series (for more details please refer to *Section 3.4.2*) in order to enhance performance and memory usage.

Note: The algorithms supported by cryptographic hardware peripherals are not included in this package. To use them, refer to the dedicate STM32Cube hardware abstraction layer (HAL) driver.

Almost all the test vectors used in the examples provided in this package are taken from NIST/FIPS standard documents.

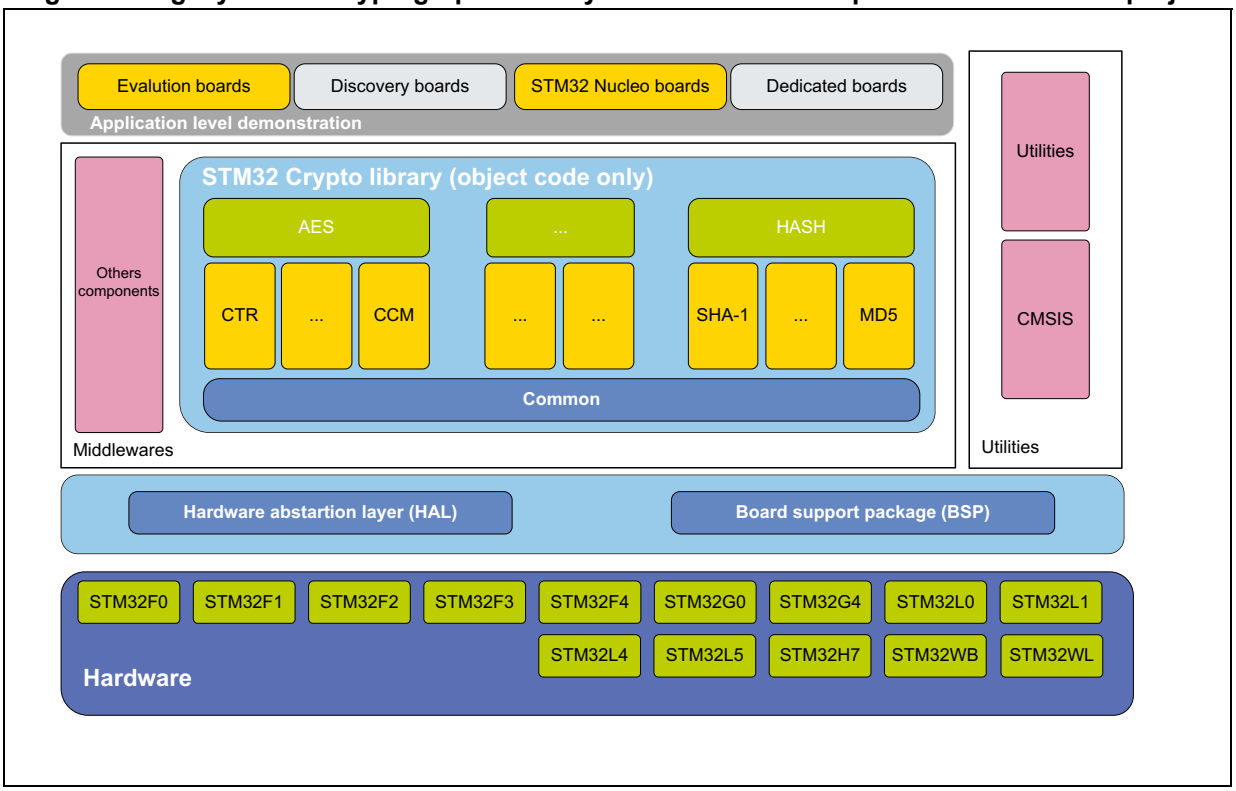
3.1 Licensing information

- Legacy STM32 cryptographic library files are provided in object format and licensed. This package cannot be used except in compliance with the License, which is available at *www.st.com*.
- Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

3.2 Architecture

The legacy STM32 cryptographic library package is based on the STM32 Cube architecture, *Figure 6* shows how the legacy STM32 cryptographic library is structured and how it can be implemented in a complete project.

Figure 6. Legacy STM32 cryptographic library architecture and implementation in a full project



Note: The CRC is used by the legacy STM32 cryptographic firmware library. When using Cryptolib APIs, CRC must be activated and set to 0xFFFFFFFF (set the CRC INIT register to 0xFFFFFFFF). Otherwise API results are not valid. On MCU with TrustZone®, the CRC peripheral must be assigned to non-secure.

As shown in *Figure 6*, the legacy STM32 cryptographic library is part of the **Middleware** layer and supports all STM32 microcontroller Series. It is based on modular architecture, allowing new algorithms to 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 optimize execution speed.

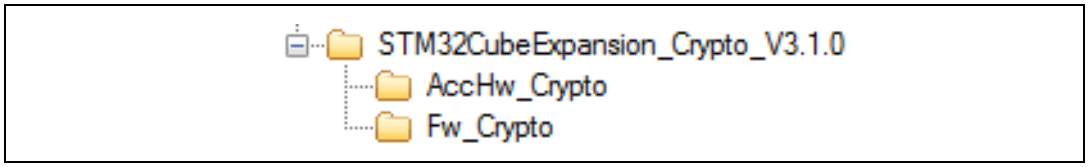
In the **Application** layer, the legacy STM32 cryptographic package provides a full set of examples (up to 31) covering all the available algorithms. All these examples also come with template projects for the most common development tools. Even without the appropriate hardware evaluation board, this layer allows the user to rapidly get started with a brand new legacy STM32 cryptographic library.

3.3 Package organization

The library is supplied as a zip file. The extraction of the zip file generates one folder, STM32CubeExpansion_Crypto_V3.1.0, which contains two subfolders:

- **AccHw_Crypto**: the legacy STM32 cryptographic hardware acceleration library package
- **Fw_Crypto**: the legacy STM32 cryptographic firmware library package.

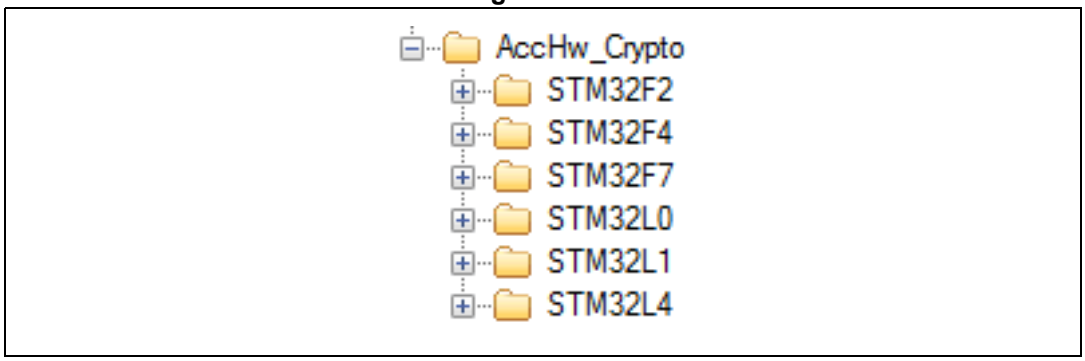
Figure 7. Legacy STM32 cryptographic library package organization



3.3.1 Legacy STM32 cryptographic hardware acceleration library package

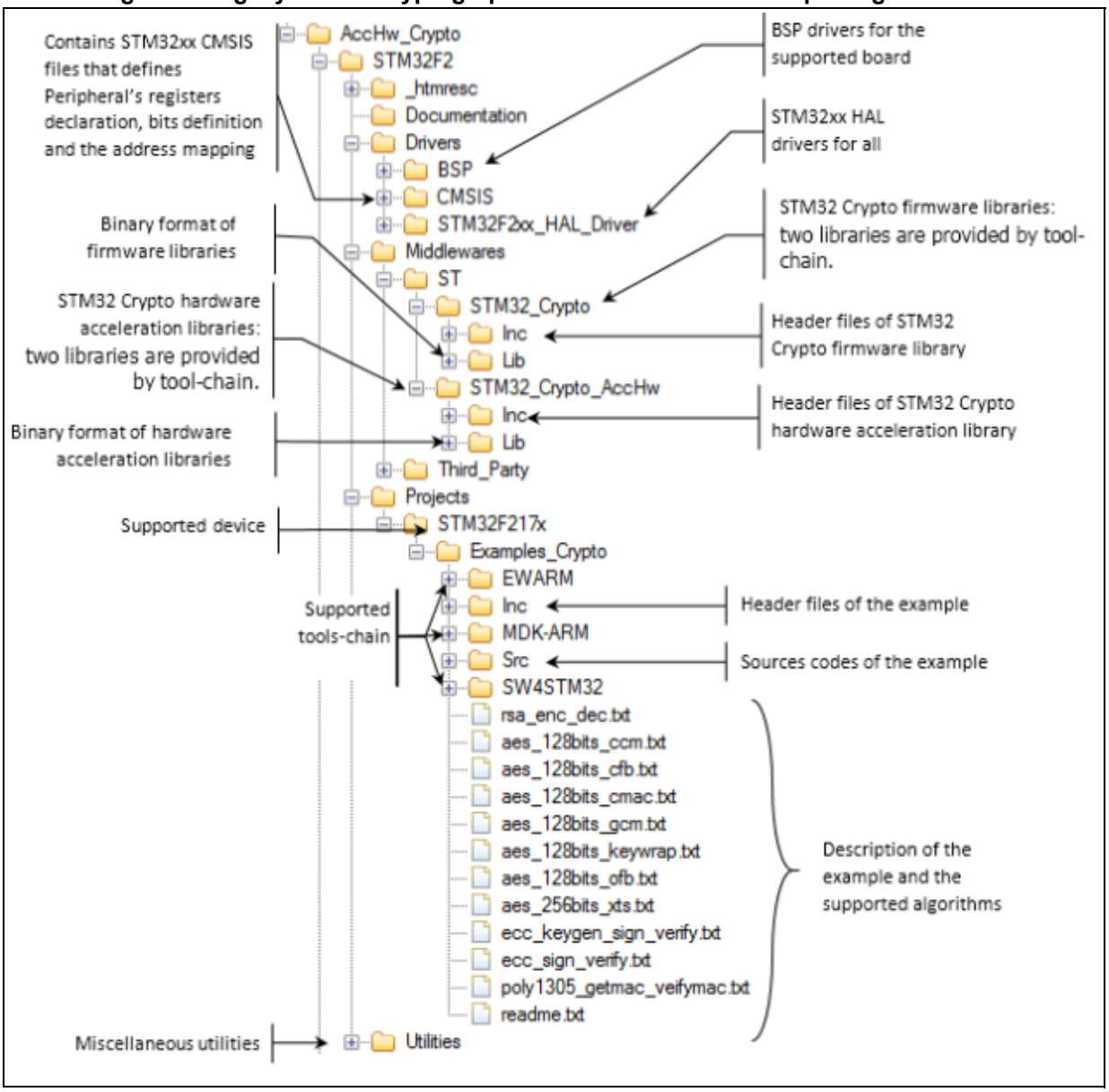
This package contains the subfolders described in *Figure 8*.

Figure 8. Legacy STM32 cryptographic hardware acceleration library package organization



The legacy STM32 cryptographic hardware acceleration library package consists of one folder for each of STM32 Series supporting cryptographic peripherals. Each folder is structured as shown in *Figure 9*.

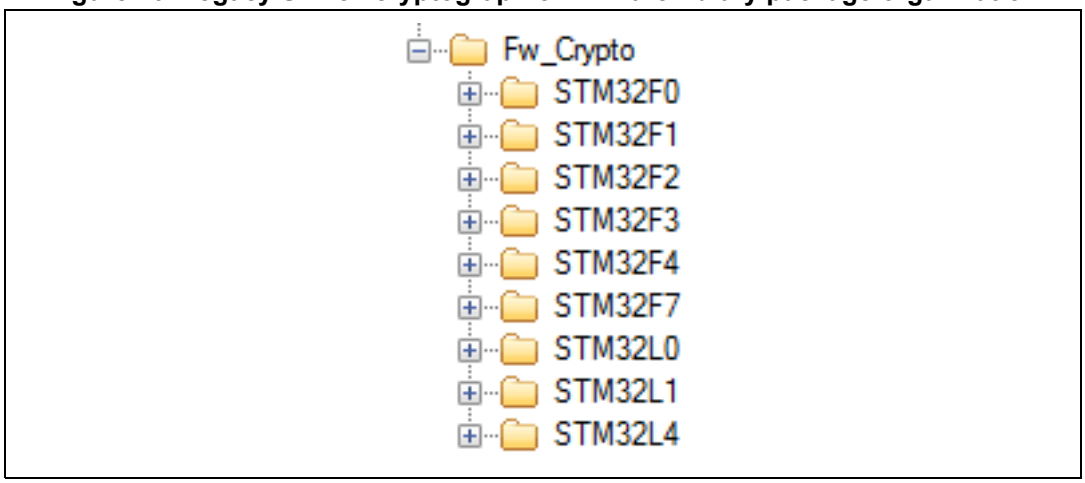
Figure 9. Legacy STM32 cryptographic hardware acceleration package structure



3.3.2 Legacy STM32 cryptographic firmware library package

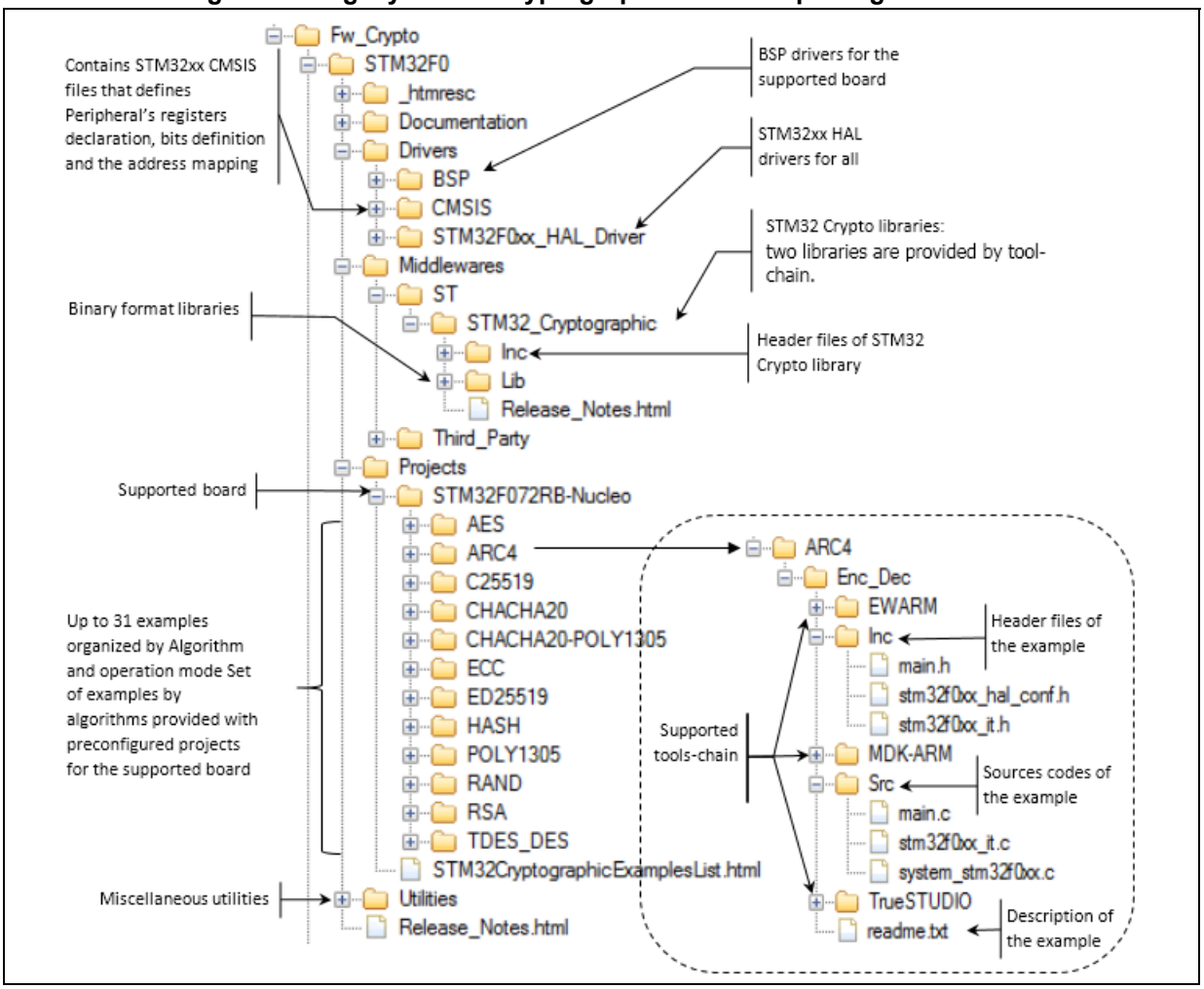
This package contains the subfolders described in *Figure 10*.

Figure 10. Legacy STM32 cryptographic firmware library package organization



The legacy STM32 cryptographic firmware library package consists of one folder for each STM32 Series, Each folder is structured as shown in *Figure 11*.

Figure 11. Legacy STM32 cryptographic firmware package structure



3.4 Binaries delivered within legacy STM32 cryptographic library

3.4.1 Legacy STM32 cryptographic firmware library

The legacy STM32 cryptographic firmware library is distributed by STMicroelectronics as an object code library linked to STM32 products.

The library is compiled for Arm® Cortex®-M0, -M0+, -M3, -M4, -M7 and -M33 cores.

Note that the library is compiler-dependent (IAR™, Arm®, GCC) and is compiled with two optimization levels (high size, high speed).

The libraries are located under “Middleware \ST\ STM32_Cryptographic\Lib” with the following naming convention:

STM32CryptographicV3.0.0_CMx_C_O, where:

- x: the CMx core class (CM0, CM0PLUS, CM3, CM33, CM4 or CM7)
- C: compiler (IAR™, Keil®, GCC)
- O: specify the compiler optimization
- In case of GCC IDE:
  - <empty>: high size optimization
  - ot: high speed optimization
  - fpu: FPU enabled
- In case of IAR™ IDE:
  - <empty>: high size optimization
  - ot: high speed optimization
  - nsc: the option No Size constraints is enabled
  - fpu: FPU enabled
- In case of Keil® IDE:
  - <empty>: high size optimization
  - ot: high speed optimization
  - slsm: the option Split Load and Store Multiple is enabled
  - o1elfspf: the option One ELF Section per Function is enabled
  - fpu: FPU enabled

Table 3 summarizes the legacy STM32 cryptographic firmware libraries provided in this package.

Table 3. Legacy STM32 cryptographic firmware libraries

Series	IDE	Optimization	Library name
STM32F0	IAR™ V8	High speed	STM32CryptographicV3.1.5_CM0_IARv8_otnsc.a
		High size	STM32CryptographicV3.1.5_CM0_IARv8.a
	Keil®	High speed	STM32CryptographicV3.1.5_CM0_KEIL_otslsm1elfspf.lib
		High size	STM32CryptographicV3.1.5_CM0_KEIL_slsm1elfspf.lib
	GCC	High speed	STM32CryptographicV3.1.5_CM0_GCC_ot.a
		High size	STM32CryptographicV3.1.5_CM0_GCC.a
STM32L0	IAR™ V8	High speed	STM32CryptographicV3.1.5_CM0PLUS_IARv8_otnsc.a
		High size	STM32CryptographicV3.1.5_CM0PLUS_IARv8.a
	Keil®	High speed	STM32CryptographicV3.1.5_CM0PLUS_KEIL_otslsm1elfspf.lib
		High size	STM32CryptographicV3.1.5_CM0PLUS_KEIL_slsm1elfspf.lib
	GCC	High speed	STM32CryptographicV3.1.5_CM0PLUS_GCC_ot.a
		High size	STM32CryptographicV3.1.5_CM0PLUS_GCC.a
STM32F1 STM32F2 STM32L1	IAR™	High speed	STM32CryptographicV3.0.0_CM3_IAR_otnsc.a
		High size	STM32CryptographicV3.0.0_CM3_IAR.a
	IAR™ V8	High speed	STM32CryptographicV3.1.1_CM3_IARV8_otnsc.a
		High size	STM32CryptographicV3.1.1_CM3_IARV8.a
	Keil®	High speed	STM32CryptographicV3.0.0_CM3_KEIL_otslsm1elfspf.lib
		High size	STM32CryptographicV3.0.0_CM3_KEIL_slsm1elfspf.lib
	GCC	High speed	STM32CryptographicV3.0.0_CM3_GCC_ot.a
		High size	STM32CryptographicV3.0.0_CM3_GCC.a
STM32F3 STM32F4 STM32L4	IAR™	High speed	STM32CryptographicV3.0.0_CM4_IAR_otnsc.a
		High size	STM32CryptographicV3.0.0_CM4_IAR.a
	IAR™ V8	High speed	STM32CryptographicV3.1.1_CM4_IARV8_otnsc.a
		High size	STM32CryptographicV3.1.1_CM4_IARV8.a
	Keil®	High speed	STM32CryptographicV3.0.0_CM4_KEIL_otslsm1elfspf.lib
		High size	STM32CryptographicV3.0.0_CM4_KEIL_slsm1elfspf.lib
	GCC	High speed	STM32CryptographicV3.0.0_CM4_GCC_ot.a
		High size	STM32CryptographicV3.0.0_CM4_GCC.a

Table 3. Legacy STM32 cryptographic firmware libraries (continued)

Series	IDE	Optimization	Library name
STM32F7	IAR™	High speed	STM32CryptographicV3.0.0_CM7_IAR_otnsc.a
		High size	STM32CryptographicV3.0.0_CM7_IAR.a
	IAR™ V8	High speed	STM32CryptographicV3.1.1_CM7_IARV8_otnsc.a
		High size	STM32CryptographicV3.1.1_CM7_IARV8.a
	Keil®	High speed	STM32CryptographicV3.0.0_CM7_KEIL_otslsm1elfspf.lib
		High size	STM32CryptographicV3.0.0_CM7_KEIL_slsm1elfspf.lib
	GCC	High speed	STM32CryptographicV3.0.0_CM7_GCC.a
		High size	STM32CryptographicV3.0.0_CM7_GCC_ot.a
STM32H7	IAR™	High speed	STM32CryptographicV3.1.1_STM32H7_IARv8_otnsc.a
		High size	STM32CryptographicV3.1.1_STM32H7_IARv8.a
	Keil® V5	High speed	STM32CryptographicV3.1.1_STM32H7_KEIL_otslsm1elfspf_ARMCCv5.lib
		High size	STM32CryptographicV3.1.1_STM32H7_KEIL_slsm1elfspf_ARMCCv5.lib
	Keil® V6	High speed	STM32CryptographicV3.1.1_STM32H7_KEIL_otslsm1elfspf_ARMCCv6.lib
		High size	STM32CryptographicV3.1.1_STM32H7_KEIL_slsm1elfspf_ARMCCv6.lib
	GCC	High speed	STM32CryptographicV3.1.1_STM32H7_GCC_ot.a
		High size	STM32CryptographicV3.1.1_STM32H7_GCC.a
STM32G0	IAR™ V8	High speed	STM32CryptographicV3.1.5_CM0PLUS_IARv8_otnsc.a
		High size	STM32CryptographicV3.1.5_CM0PLUS_IARv8.a
	Keil®	High speed	STM32CryptographicV3.1.5_CM0PLUS_KEIL_ot1elfspf.lib
		High size	STM32CryptographicV3.1.5_CM0PLUS_KEIL_1elfspf.lib
	GCC	High speed	STM32CryptographicV3.1.5_CM0PLUS_GCC_ot.a
		High size	STM32CryptographicV3.1.5_CM0PLUS_GCC.a
STM32WL	IAR™ V8	High speed	STM32CryptographicV3.1.3_CM4_IARv8_otnsc.a
			STM32CryptographicV3.1.5_CM0PLUS_IARv8_otnsc.a
		High size	STM32CryptographicV3.1.3_CM4_IARv8_otnsc.a
			STM32CryptographicV3.1.5_CM0PLUS_IARv8_otnsc.a
	Keil®	High speed	STM32CryptographicV3.1.3_CM4_KEIL_ot1elfspf.lib
			STM32CryptographicV3.1.5_CM0PLUS_KEIL_ot1elfspf.lib
		High size	STM32CryptographicV3.1.3_CM4_KEIL_ot1elfspf.lib
			STM32CryptographicV3.1.5_CM0PLUS_KEIL_ot1elfspf.lib
	GCC	High speed	STM32CryptographicV3.1.3_CM4_GCC_ot.a
			STM32CryptographicV3.1.5_CM0PLUS_GCC_ot.a
		High size	STM32CryptographicV3.1.3_CM4_GCC.a
			STM32CryptographicV3.1.5_CM0PLUS_GCC.a

Table 3. Legacy STM32 cryptographic firmware libraries (continued)

Series	IDE	Optimization	Library name
STM32G4 STM32WB	IAR™ V8	High speed (with FPU)	STM32CryptographicV3.1.3_CM4_IARv8_otnsc_fpu.a
		High speed (without FPU)	STM32CryptographicV3.1.3_CM4_IARv8_otnsc.a
		High size (with FPU)	STM32CryptographicV3.1.3_CM4_IARv8_fpu.a
		High size (without FPU)	STM32CryptographicV3.1.3_CM4_IARv8.a
	Keil®	High speed (with FPU)	STM32CryptographicV3.1.3_CM4_KEIL_ot1elfspf_fpu.lib
		High speed (without FPU)	STM32CryptographicV3.1.3_CM4_KEIL_ot1elfspf.lib
		High size (with FPU)	STM32CryptographicV3.1.3_CM4_KEIL_1elfspf_fpu.lib
		High size (without FPU)	STM32CryptographicV3.1.3_CM4_KEIL_1elfspf.lib
	GCC	High speed (with FPU)	STM32CryptographicV3.1.3_CM4_GCC_ot_fpu.a
		High speed (without FPU)	STM32CryptographicV3.1.3_CM4_GCC_ot.a
		High size (with FPU)	STM32CryptographicV3.1.3_CM4_GCC_fpu.a
		High size (without FPU)	STM32CryptographicV3.1.3_CM4_GCC.a

Table 3. Legacy STM32 cryptographic firmware libraries (continued)

Series	IDE	Optimization	Library name
STM32L5	IAR™ V8	High speed (with FPU)	STM32CryptographicV3.1.3_CM33_IARv8_otnsc_fpu.a
		High speed (without FPU)	STM32CryptographicV3.1.3_CM33_IARv8_otnsc.a
		High size (with FPU)	STM32CryptographicV3.1.3_CM33_IARv8_fpu.a
		High size (without FPU)	STM32CryptographicV3.1.3_CM33_IARv8.a
	Keil®	High speed (with FPU)	STM32CryptographicV3.1.3_CM33_KEIL_ot1elfspf_fpu.lib
		High speed (without FPU)	STM32CryptographicV3.1.3_CM33_KEIL_ot1elfspf.lib
		High size (with FPU)	STM32CryptographicV3.1.3_CM33_KEIL_1elfspf_fpu.lib
		High size (without FPU)	STM32CryptographicV3.1.3_CM33_KEIL_1elfspf.lib
	GCC	High speed (with FPU)	STM32CryptographicV3.1.3_CM33_GCC_ot_fpu.a
		High speed (without FPU)	STM32CryptographicV3.1.3_CM33_GCC_ot.a
		High size (with FPU)	STM32CryptographicV3.1.3_CM33_GCC_fpu.a
		High size (without FPU)	STM32CryptographicV3.1.3_CM33_GCC.a

3.4.2 Legacy STM32 cryptographic hardware acceleration library

The legacy STM32 cryptographic acceleration hardware library is distributed by STMicroelectronics as an object code library.

Note that the library is compiler-dependent (IAR™, Arm®, GCC) and is compiled with two optimization levels (High size, High speed).

The libraries are located under “Middleware\ST\STM32_Crypto_AccHw\Lib” with the following naming convention:

STM32AccHwCryptoV3.1.0_Xy_C_O.a, where:

1. Xy: the STM32 Series supported(F2, F4, F7, L0, L1, L4)
2. C: compiler (IAR™, Keil®, GCC)
3. O: specify the compiler optimization
   - In case of GCC IDE:
     - <empty>: high size optimization
     - ot: high speed optimization
   - In case of IAR™ IDE:
     - <empty>: high size optimization

  - ot: high speed optimization
  - nsc: the option No Size constraints is enabled
- In case of Keil® IDE:
  - <empty>: high size optimization
  - ot: high speed optimization
  - slsm: the option Split Load and Store Multiple is enabled
  - o1elfspf: the option One ELF Section per Function is enabled

Table 4 summarizes the legacy STM32 cryptographic acceleration hardware libraries provided in this package and the algorithms supported by each library.

Table 4. Legacy STM32 cryptographic hardware acceleration libraries

Series	Devices	Supported algorithms	Peripheral used
STM32F2	STM32F20x	– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1	
	STM32F21x	– AES: CFB, OFB, XTS, CCM, GCM, CMAC, KeyWrap Key size: 128, 192, 256 bit	Cryptographic accelerator
		– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1.5	
STM32F4	STM32F405/407	– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
	STM32F427/429	– RSA: encryption/decryption functions with PKCS#1v1	
	STM32F415x/417x	– AES: CFB, OFB, XTS, CCM, GCM, CMAC, KeyWrap Key size: 128, 192, 256 bit	Cryptographic accelerator
		– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1.5	
	STM32F437x/439x	– AES: CFB, OFB, XTS, CCM, GCM, CMAC, KeyWrap Key size: 128, 192, 256 bit	Cryptographic accelerator
		– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1.5	
STM32F7	STM32F745x/746x	– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1.5	
	STM32F756xx	– AES: CFB, OFB, XTS, CCM, GCM, CMAC, KeyWrap Key size: 128, 192, 256 bit	Cryptographic accelerator
		– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1.5	
STM32L0	STM32L05x	– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1.5	
	STM332L06x	– AES: CFB, OFB, XTS, CCM, GCM, CMAC, KeyWrap Key size: 128 bit	AES hardware accelerator
		– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1.5	

Table 4. Legacy STM32 cryptographic hardware acceleration libraries (continued)

Series	Devices	Supported algorithms	Peripheral used
STM32L1	STM32L16x	– AES: CFB, OFB, XTS, CCM, GCM, CMAC, KeyWrap Key size: 128 bit	AES hardware accelerator
STM32L4	STM32L471xx	– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1.5	
	STM32L486xx	– AES: CFB, OFB, XTS, CCM, KeyWrap Key size: 128, 256 bit	AES hardware accelerator
		– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1.5	

Note: The user can use the HAL driver for the algorithms supported by the cryptographic peripheral (such as DES-ECB for STM32F2 Series).

4 AES algorithm

4.1 AES description

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

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

- ECB
- CBC
- CTR
- CFB
- OFB
- XTS
- CCM
- GCM
- CMAC
- KEY WRAP

These modes run with all STM32 microcontrollers using a software algorithm implementation.

For AES library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For the modes supported with hardware acceleration, refer to *Section 3.4.2: Legacy STM32 cryptographic hardware acceleration library*.

For AES library performances and memory requirements, refer to:

- the legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\FW_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.
- The legacy STM32 cryptographic hardware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\AccHw_Crypto\STM32XY\Documentation” where XY indicates the STM32.

4.2 AES library functions (ECB, CBC, CTR, CFB and OFB)

Table 5 describes the AES functions of the legacy STM32 cryptographic firmware library.

Table 5. AES algorithm functions of the legacy STM32 cryptographic firmware library

Function name	Description
`AES_AAA_Encrypt_Init`	Loads the key and ivec(1), performs key schedule
`AES_AAA_Encrypt_Append`	Launches cryptographic operation, can be called several times

Table 5. AES algorithm functions of the legacy STM32 cryptographic firmware library

Function name	Description
AES_AAA_Encrypt_Finish	AES encryption finalization of AAA mode
AES_AAA_Decrypt_Init	Loads the key and ivec(1), if needed 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

1. ivec: Initialization vector.

AAA represents the mode of operation of the AES algorithm, it is can be ECB, CBC, CTR, CFB or OFB.

Table 6 describes the AES functions of th legacy STM32 cryptographic hardware acceleration library.

Table 6. AES algorithm functions of the legacy STM32 cryptographic hardware acceleration library

Function name	Description
AccHw_AES_AAA_Encrypt_Init	Loads the key and ivec, performs key schedule
AccHw_AES_AAA_Encrypt_Append	Launches cryptographic operation, can be called several times
AccHw_AES_AAA_Encrypt_Finish	AES encryption finalization of AAA mode
AccHw_AES_AAA_Decrypt_Init	Loads the key and ivec, eventually performs key schedule
AccHw_AES_AAA_Decrypt_Append	Launches cryptographic operation, can be called several times
AccHw_AES_AAA_Decrypt_Finish	AES decryption finalization of AAA mode

As an example, to use CFB mode for AES algorithm from the firmware library the functions in *Table 7* must be used, while those in *Table 8* are the ones to use CFB for the hardware acceleration library.

Table 7. AES CFB algorithm functions of the legacy STM32 cryptographic firmware library

Function name	Description
AES_CFB_Encrypt_Init	Loads the key and ivec(1), performs key schedule
AES_CFB_Encrypt_Append	Launches cryptographic operation, can be called several times
AES_CFB_Encrypt_Finish	Possible final output
AES_CFB_Decrypt_Init	Loads the key and ivec(1), performs key schedule, initializes hardware
AES_CFB_Decrypt_Append	Launches cryptographic operation, can be called several times
AES_CFB_Decrypt_Finish	Possible final output

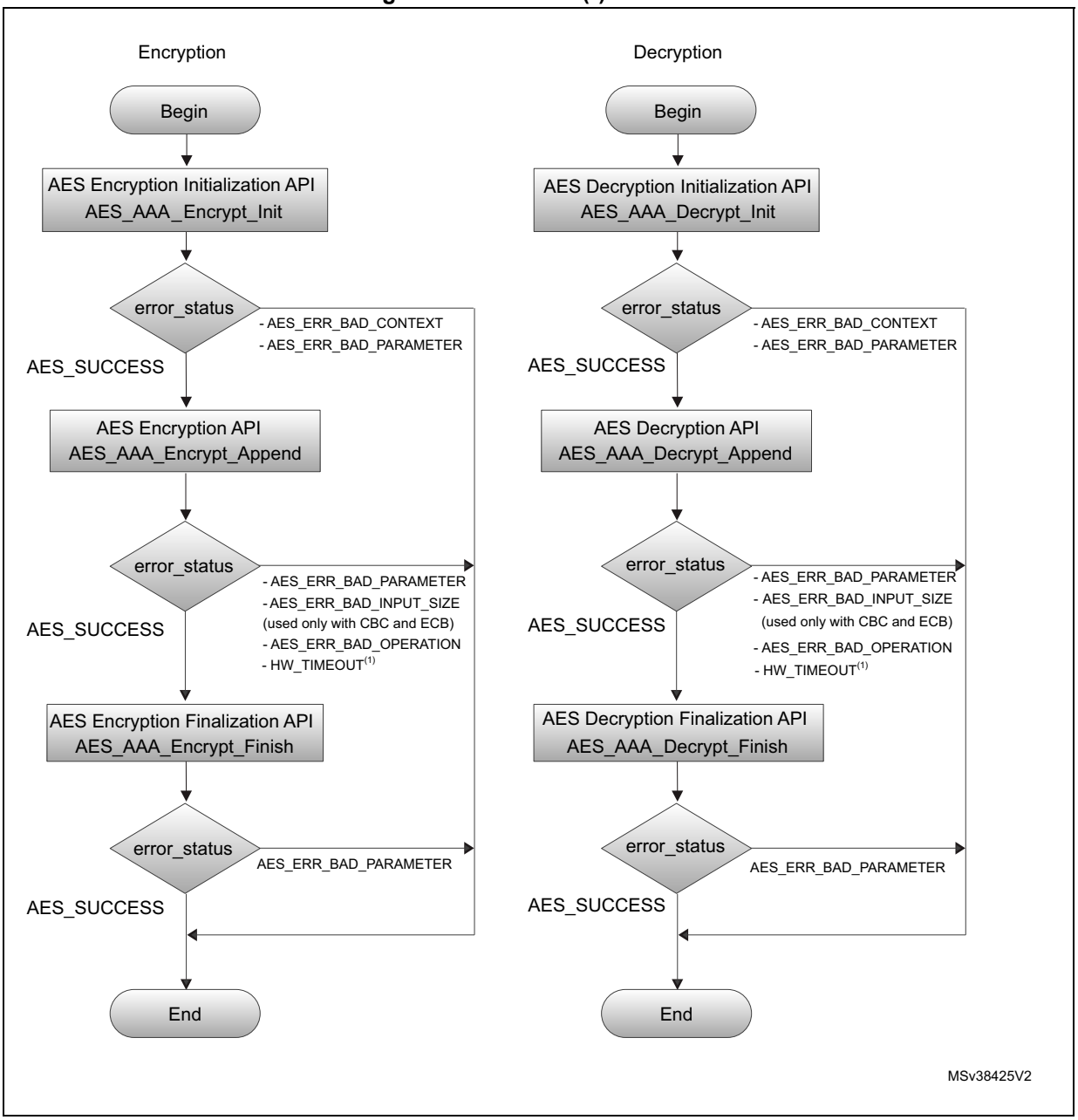
1. ivec: Initialization vector.

Table 8. AES CFB algorithm functions of the legacy STM32 cryptographic hardware acceleration library

Function name	Description
AccHw_AES_CFB_Encrypt_Init	Loads the key and ivec, performs key schedule
AccHw_AES_CFB_Encrypt_Append	Launches cryptographic operation, can be called several times
AccHw_AES_CFB_Encrypt_Finish	Possible final output
AccHw_AES_CFB_Decrypt_Init	Loads the key and ivec, performs key schedule, init hw. and so on
AccHw_AES_CFB_Decrypt_Append	Launches cryptographic operation, can be called several times
AccHw_AES_CFB_Decrypt_Finish	Possible final output

Figure 12 describes the AES_AAA algorithm.

Figure 12. AES AAA(*) flowchart



1. Used only with the algorithms featuring hardware acceleration.

4.2.1 AES_AAA_Encrypt_Init function

Table 9. AES_AAA_Encrypt_Init

Function name	AES_AAA_Encrypt_Init
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.

AAA is ECB, CBC, CFB, or OFB

AESAAActx_stt data structure

Structure type for public key.

Table 10. AESAAActx_stt data structure(1)

Field name	Description
`uint32_t mContextId`	Unique ID of this context. Not used in current version
`SKflags_et(2) mFlags`	32 bit mFlags, used to perform key schedule. Default value is E_SK_DEFAULT. See *Table 11: SKflags_et mFlags* for details.
`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 (default CRL_AES_BLOCK)
`uint32_t amIv[4]`	Temporary result/IV
`uint32_t mKeySize`	Key length in bytes. The following predefined values, supported by each library, can be used instead of the size of the key: – CRL_AES128_KEY – CRL_AES192_KEY – CRL_AES256_KEY
`uint32_t amExpKey [CRL_AES_MAX_EXPKEY_SIZE]`	Expanded AES key

1. In case of using the hardware library this structure is "AccHw_AESAAActx_stt"
2. In case of using the hardware library this structure is "AccHw_SKflags"

SKflags_et mFlags

Enumeration of allowed flags in a context for symmetric key operations with the firmware implementation.

Table 11. *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_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
E_SK_SET_COUNTER	User Flag: With ChaCha20 this is used to indicate a value for the counter, used to process non contiguous blocks (i.e. jump ahead)

AccHw_SKflags_et mFlags

Enumeration of allowed flags in a context for symmetric key operations with the hardware implementation.

Table 12. AccHw_SKflags_et mFlags

Field name	Description
AccHw_E_SK_DEFAULT	User Flag: No flag specified. This is the default value for this flag
AccHw_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.
AccHw_E_SK_REDSERVED	reserved
AccHw_E_SK_FINAL_APPEND	User Flag: Must be set in some modes before final Append call occurs.
AccHw_E_SK_OPERATION_COMPLETED	Internal Flag: Checks that the Finish function has been called.
AccHw_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.

Table 12. AccHw_SKflags_et mFlags (continued)

Field name	Description
AccHw_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
E_SK_SET_COUNTER	User Flag: With ChaCha20 this is used to indicate a value for the counter, used to process non contiguous blocks (i.e. jump ahead)

4.2.2 AES_AAA_Encrypt_Append function

Table 13. AES_AAA_Encrypt_Append

Function name	AES_AAA_Encrypt_Append
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. – HW_TIMEOUT (Used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

AAA is ECB, CBC, CTR, CFB or OFB.

Note: This function can be called several times, provided 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, Cipher-text Stealing is activated.*
- *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 14. AES_AAA_Encrypt_Finish

Function name	AES_AAA_Encrypt_Finish
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.

AAA is ECB, CBC, CTR, CFB or OFB.

Note: This function does not write output data, and can hence be skipped. It is kept for API compatibility.

4.2.4 AES_AAA_Decrypt_Init function

Table 15. AES_AAA_Decrypt_Init

Function name	AES_AAA_Decrypt_Init
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.

AAA is ECB, CBC, CTR, CFB or OFB

Note: 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 16. AES_AAA_Decrypt_Append

Function name	AES_AAA_Decrypt_Append
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) – HW_TIMEOUT (Used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

AAA is ECB, CBC, CTR, CFB or OFB.

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

In CBC mode and in case of a call where P_inputSize is greater than 16 and not multiple of 16, Cipher-text Stealing is activated.

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

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 17. AES_AAA_Decrypt_Finish

Function name	AES_AAA_Decrypt_Finish
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

Table 17. AES_AAA_Decrypt_Finish (continued)

Function name	AES_AAA_Decrypt_Finish
Parameter	– [in,out] *P_pAESAAActx: AES AAA context – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that contains 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

AAA is ECB, CBC, CTR, CFB or OFB.

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

This function does not write output data and can hence be skipped. It is kept for API compatibility.

4.3 AES GCM library functions

Table 18 describes the AES GCM library of the legacy STM32 cryptographic firmware library.

Table 18. AES GCM algorithm functions of firmware library

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 creates 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 is checked

Table 19 describe the AES GCM functions of the legacy STM32 cryptographic hardware acceleration library.

Table 19. AES GCM algorithm functions of hardware acceleration library

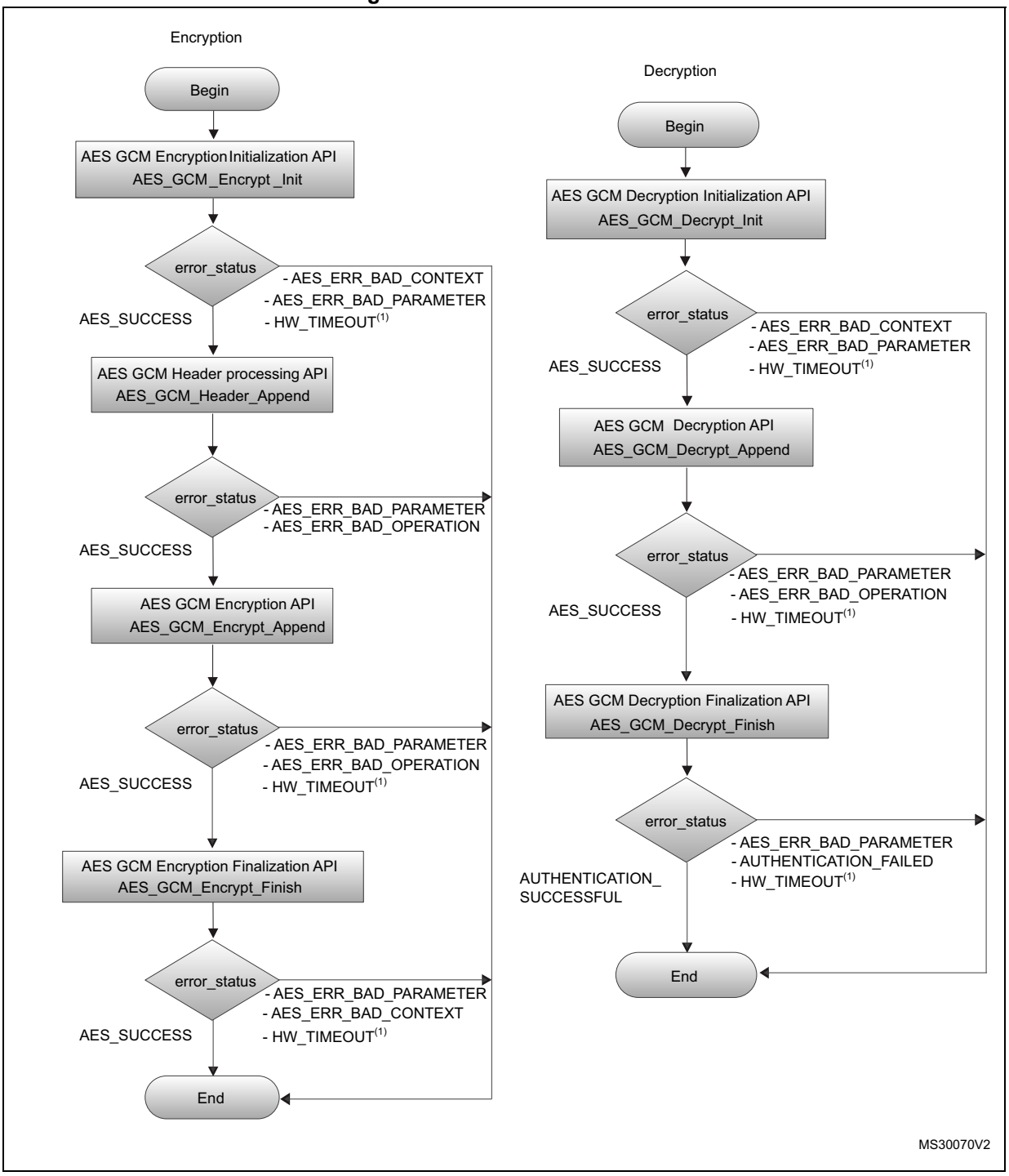
Function name	Description
AccHw_AES_GCM_Encrypt_Init	Initialization for AES GCM encryption
AccHw_AES_GCM_Header_Append	Header processing function
AccHw_AES_GCM_Encrypt_Append	AES GCM encryption function
AccHw_AES_GCM_Encrypt_Finish	AES GCM finalization during encryption, this creates the Authentication TAG

Table 19. AES GCM algorithm functions of hardware acceleration library (continued)

Function name	Description
AccHw_AES_GCM_Decrypt_Init	Initialization for AES GCM decryption
AccHw_AES_GCM_Decrypt_Append	AES GCM decryption function
AccHw_AES_GCM_Decrypt_Finish	AES GCM finalization during decryption, the Authentication TAG is checked

The flowcharts in *Figure 13* describe the AES_GCM algorithm.

Figure 13. AES GCM flowchart



1. Used only with the algorithms featuring hardware acceleration.

4.3.1 AES_GCM_Encrypt_Init function

Table 20. 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 – HW_TIMEOUT (Used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

AESGCMctx_stt data structure

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

Table 21. AESGCMctx_stt data structure(1)

File name	Description
uint32_t mContextId	Unique ID of this AES-GCM context. Not used in current implementation.
SKflags_et(2) mFlags	32 bit mFlags, used to perform key schedule. Default value is E_SK_DEFAULT. See *Table 11: SKflags_et mFlags* for details.
*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 (only 12 is supported value). This must be set by the caller prior to calling Init.
uint32_t amIv[4]	This is the current IV value.
int32_t mKeySize	Key length in bytes, must be set by the caller prior to calling Init. The following predefined values, supported by each library, can be used instead of the size of the key: – CRL_AES128_KEY – CRL_AES192_KEY – CRL_AES256_KEY
*const uint8_t * pmTag*	Pointer to Authentication TAG. Must be set in decryption, and this TAG is verified.
int32_t mTagSize	Size of the Tag to return (must be a valid value between 1 and 16). Must be set by the caller prior to calling Init.

Table 21. AESGCMctx_stt data structure(1)

File name	Description
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

1. In case of using the hardware library this structure is "AccHw_AESGCMctx_stt".
2. In case of using the hardware library this structure is "AccHw_SKflags".

4.3.2 AES_GCM_Header_Append function

Table 22. 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 23. AES_GCM_Encrypt_Append

Function name	AES_GCM_Encrypt_Append
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 contains 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 – HW_TIMEOUT (Used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: This function can be called several 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 24. AES_GCM_Encrypt_Finish

Function name	AES_GCM_Encrypt_Finish
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 creates the Authentication TAG

Table 24. AES_GCM_Encrypt_Finish (continued)

Function name	AES_GCM_Encrypt_Finish
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 – HW_TIMEOUT (Used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

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

4.3.5 AES_GCM_Decrypt_Init function

Table 25. 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. – HW_TIMEOUT (Used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

4.3.6 AES_GCM_Decrypt_Append function

Table 26. 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 – HW_TIMEOUT (Used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

4.3.7 AES_GCM_Decrypt_Finish function

Table 27. 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 is checked

Table 27. AES_GCM_Decrypt_Finish

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 is 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 – HW_TIMEOUT (Used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

4.4 AES KeyWrap library functions

Table 28 describes the AES_KeyWrap function of the legacy STM32 cryptographic firmware library.

Table 28. AES KeyWrap algorithm functions of firmware library

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 is checked

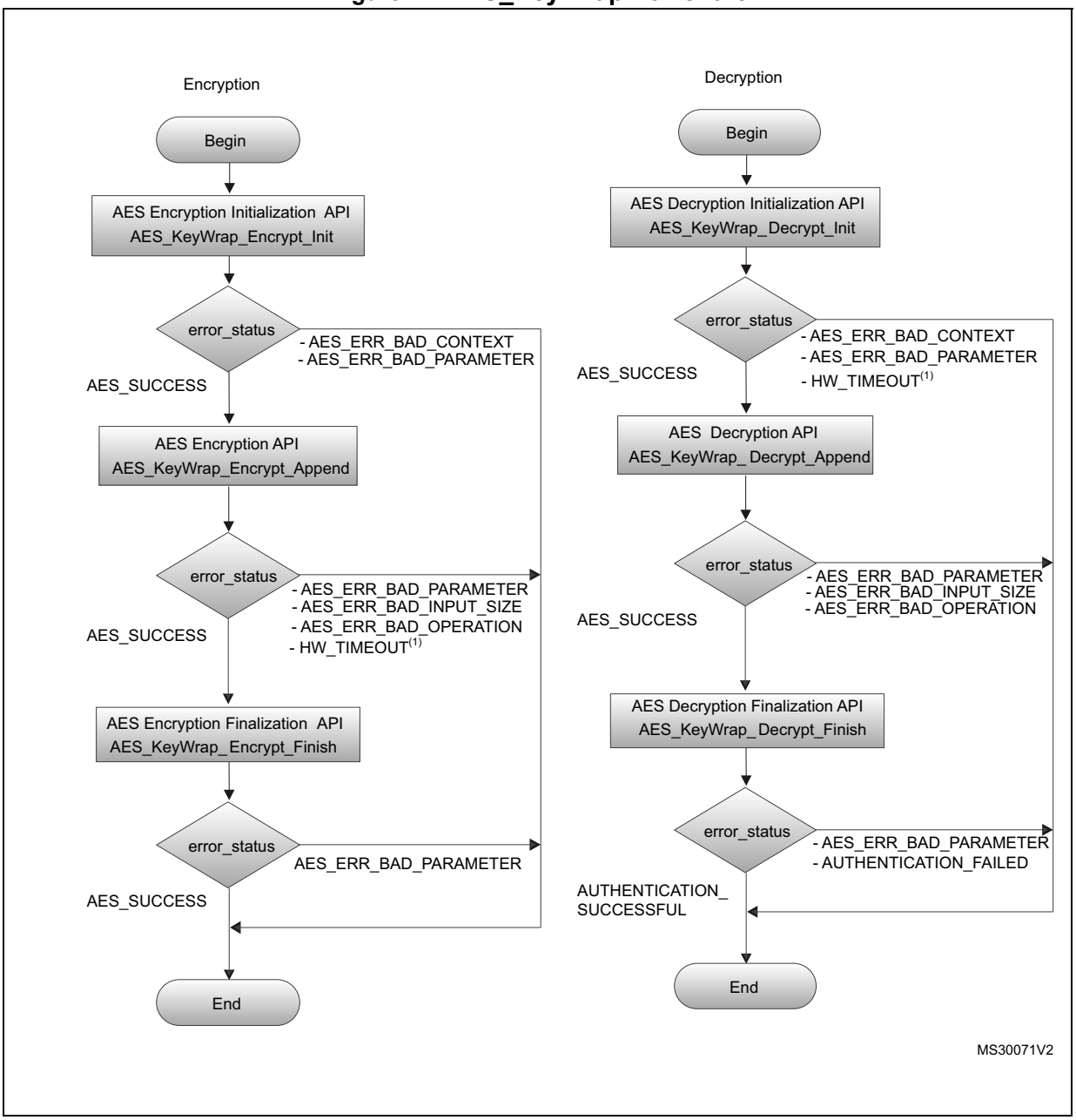
Table 29 describes the AES KEYWRAP functions of the legacy STM32 cryptographic hardware acceleration library.

Table 29. AES KeyWrap algorithm functions of hardware accelaration library

Function name	Description
AccHw_AES_KeyWrap_Encrypt_Init	Initialization for AES KeyWrap Encryption
AccHw_AES_KeyWrap_Encrypt_Append	AES KeyWrap Wrapping function
AccHw_AES_KeyWrap_Encrypt_Finish	AES KeyWrap Finalization
AccHw_AES_KeyWrap_Decrypt_Init	Initialization for AES KeyWrap Decryption
AccHw_AES_KeyWrap_Decrypt_Append	AES KeyWrap UnWrapping function
AccHw_AES_KeyWrap_Decrypt_Finish	AES KeyWrap Finalization during Decryption, the authentication is checked

The flowcharts in *Figure 14* describe the AES_KeyWrap algorithm.

Figure 14. AES_KeyWrap flowchart



1. Used only with the algorithms featuring hardware acceleration.

4.4.1 AES_KeyWrap_Encrypt_Init function

Table 30. 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 bit 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: 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 *Table 10: AESAAActx_stt data structure*.

4.4.2 AES_KeyWrap_Encrypt_Append function

Table 31. AES_KeyWrap_Encrypt_Append

Function name	AES_KeyWrap_Encrypt_Append
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

Table 31. AES_KeyWrap_Encrypt_Append (continued)

Function name	AES_KeyWrap_Encrypt_Append
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
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 bit – HW_TIMEOUT (Used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

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

P_pOutputBuffer must be at least 8 bytes longer than P_pInputBuffer.

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

4.4.3 AES_KeyWrap_Encrypt_Finish function

Table 32. AES_KeyWrap_Encrypt_Finish

Function name	AES_KeyWrap_Encrypt_Finish

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 is zero)
Return value	– AES_SUCCESS: Operation successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer

Note: This function does not write output data and can hence be skipped. It is kept for API compatibility.

4.4.4 AES_KeyWrap_Decrypt_Init function

Table 33. 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 bit 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. – HW_TIMEOUT (Used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: 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 34. 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

Table 34. AES_KeyWrap_Decrypt_Append (continued)

Function name	AES_KeyWrap_Decrypt_Append
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 bit and at maximum 264

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

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

P_pOutputBuffer must be at least 8 bytes smaller than P_pInputBuffer.

4.4.6 AES_KeyWrap_Decrypt_Finish function

Table 35. 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 is checked
Parameter	– [in,out] *P_pAESKWctx: AES KeyWrap context – [out] *P_pOutputBuffer: Won't be used – [out] *P_pOutputSize: Contains 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 36 describes the AES CMAC functions of the legacy STM32 cryptographic firmware library.

Table 36. AES CMAC algorithm functions of firmware library

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-CMAC decryption Data Processing
AES_CMAC_Decrypt_Finish	AES Finalization of CMAC Mode

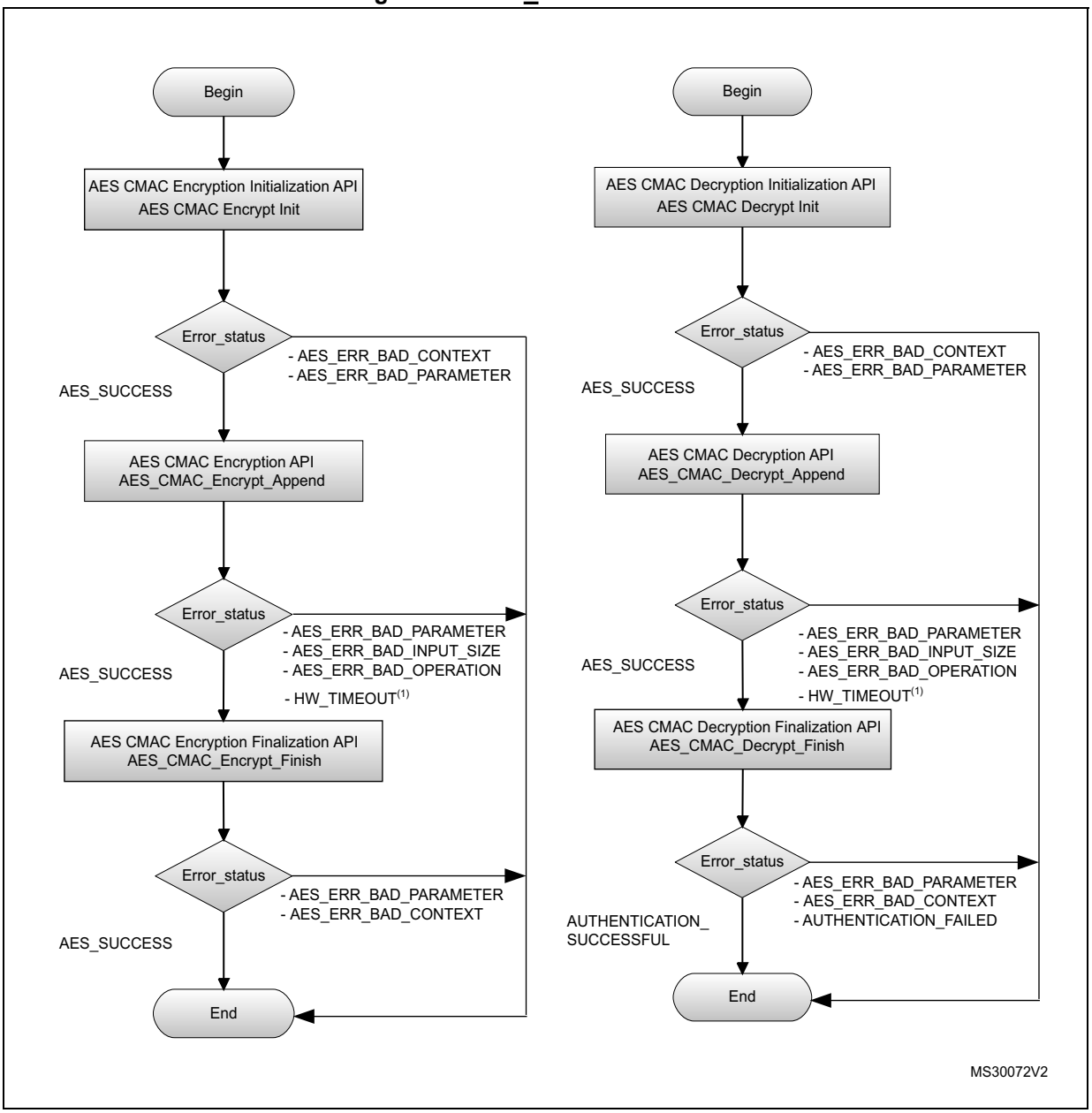
Table 37 describes the AES CMAC functions of the legacy STM32 cryptographic hardware acceleration library.

Table 37. AES CMAC algorithm functions of hardware acceleration library

Function name	Description
AccHw_AES_CMAC_Encrypt_Init	Initialization for AES-CMAC for Authentication TAG Generation
AccHw_AES_CMAC_Encrypt_Append	AES Encryption in CMAC Mode
AccHw_AES_CMAC_Encrypt_Finish	AES Finalization of CMAC Mode
AccHw_AES_CMAC_Decrypt_Init	Initialization for AES-CMAC for Authentication TAG Verification
AccHw_AES_CMAC_Decrypt_Append	AES-CMAC decryption Data Processing
AccHw_AES_CMAC_Decrypt_Finish	AES Finalization of CMAC Mode

The flowcharts in *Figure 15* describe the AES_CMAC algorithm.

Figure 15. AES_CMAC flowchart



1. Used only with the algorithms featuring hardware acceleration.

4.5.1 AES_CMAC_Encrypt_Init function

Table 38. 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

AESCMACctx_stt data structure

Table 39. AESCMACctx_stt data structure(1)

Field name	Description
`uint32_t mContextId`	Unique ID of this AES-GCM Context. Not used in this implementation.
`SKflags_et(2) mFlags`	32 bit mFlags, used to perform keyschedule. Default value is E_SK_DEFAULT. See *Table 11: SKflags_et mFlags* for details.
`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`	Key length in bytes, must be set by the caller prior to calling Init. The following predefined values, supported by each library, can be used instead of the size of the key: – CRL_AES128_KEY – CRL_AES192_KEY – CRL_AES256_KEY
`uint32_t amExpKey[CRL_AES_MAX_EXPKEY_SIZE]`	Key length in bytes, must be set by the caller prior to calling Init. The following predefined values, supported by each library, can be used
`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 (should be a valid value between 1 and 16). Must be set by the caller prior to calling Init

1. In case of using the hardware library this structure is "AccHw_AESCMACctx_stt"
2. In case of using the hardware library this structure is "AccHw_SKflags"

4.5.2 AES_CMAC_Encrypt_Append function

Table 40. AES_CMAC_Encrypt_Append

Function name	AES_CMAC_Encrypt_Append
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. – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: This function can be called several 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 41. AES_CMAC_Encrypt_Finish

Function name	AES_CMAC_Encrypt_Finish
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.

4.5.4 AES_CMAC_Decrypt_Init function

Table 42. 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.

4.5.5 AES_CMAC_Decrypt_Append function

Table 43. 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. – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: This function can be called several 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 44. 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 45 describes the AES CCM functions of the legacy STM32 cryptographic firmware library.

Table 45. AES CCM algorithm functions of firmware library

Function name	Description
`AES_CCM_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 creates 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 is checked.

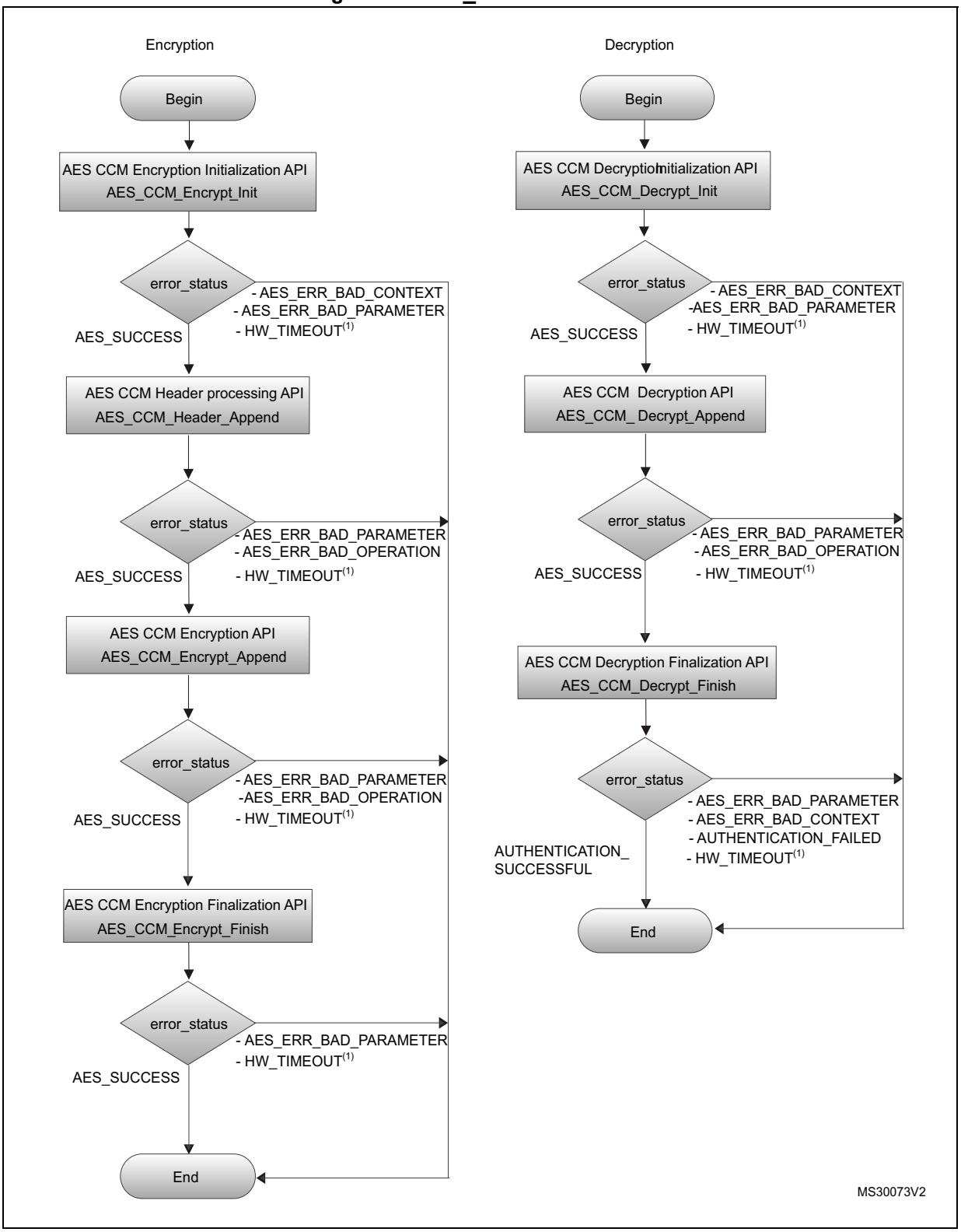
Table 46 describe the AES CCM functions of the legacy STM32 cryptographic hardware acceleration library.

Table 46. AES CCM algorithm functions of hardware acceleration library

Function name	Description
AccHw_AES_CCM_Encrypt_Init	Initialization for AES CCM encryption.
AccHw_AES_CCM_Header_Append	Header processing Function.
AccHw_AES_CCM_Encrypt_Append	AES CCM encryption function.
AccHw_AES_CCM_Encrypt_Finish	AES CCM finalization during encryption, this creates the Authentication TAG.
AccHw_AES_CCM_Decrypt_Init	Initialization for AES CCM decryption.
AccHw_AES_CCM_Decrypt_Append	AES CCM decryption function.
AccHw_AES_CCM_Decrypt_Finish	AES CCM finalization during decryption, the authentication TAG is checked.

The flowcharts in *Figure 16* describe the AES_CCM algorithm.

Figure 16. AES_CCM flowcharts



1. Used only with the algorithms featuring hardware acceleration.

4.6.1 AES_CCM_Encrypt_Init function

Table 47. 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, – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: 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 is used to output the authentication TAG.

AESCCMctx_stt data structure

Table 48. AESCCMctx_stt data structure(1)

Field name	Description
`uint32_t mContextId`	Unique ID of this AES-CCM Context. Not used in current implementation.
`SKflags_et(2) mFlags`	32 bit mFlags, used to perform keyschedule. Default value is E_SK_DEFAULT. See *Table 11: SKflags_et mFlags* for details.
`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

Table 48. AESCCMctx_stt data structure(1)

Field name	Description
int32_t mKeySize	Key length in bytes, must be set by the caller prior to calling Init. The following predefined values, supported by each library, can be used instead of the size of the key: – CRL_AES128_KEY – CRL_AES192_KEY – CRL_AES256_KEY
*const uint8_t * pmTag*	Pointer to Authentication TAG. This value must be set in decryption, and this TAG is verified.
int32_t mTagSize	Size of the Tag to return. This must be set by the caller prior to calling Init. Possible values are {4,6,8,10,12,14,16}
int32_t mAssDataSize	Size of the associated data (i.e. Header or any data that is authenticated but not encrypted) to be processed yet. This must be set by the caller prior to calling Init
int32_t mPayloadSize	Size of the payload data (i.e. Data that is authenticated and encrypted) 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

1. In case of using the hardware library this structure is "AccHw_AESCCMctx_stt"
2. In case of using the hardware library this structure is "AccHw_SKflags"

4.6.2 AES_CCM_Header_Append function

Table 49. AES_CCM_Header_Append

Function name	AES_CCM_Header_Append

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 – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: This function can be called several 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 50. AES_CCM_Encrypt_Append

Function name	AES_CCM_Encrypt_Append

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

Table 50. AES_CCM_Encrypt_Append

Function name	AES_CCM_Encrypt_Append
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 – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: This function can be called several 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 51. 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 creates 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 – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

4.6.5 AES_CCM_Decrypt_Init function

Table 52. 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. – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: 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 is checked by AES_CCM_Decrypt_Finish.

4.6.6 AES_CCM_Decrypt_Append function

Table 53. AES_CCM_Decrypt_Append

Function name	AES_CCM_Decrypt_Append

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

Table 53. AES_CCM_Decrypt_Append

Function name	AES_CCM_Decrypt_Append
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 contains 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. – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: This function must not process the TAG which is part of the ciphertext according to CCM standard.

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

4.6.7 AES_CCM_Decrypt_Finish function

Table 54. 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 is checked
Parameter	– [in,out] *P_pAESCCMctx: AES CCM context – [out] *P_pOutputBuffer: Won't be used – [out] *P_pOutputSize: Contains 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 – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

4.7 AES XTS library functions

Table 55 describes the AES XTS functions of the legacy STM32 cryptographic firmware library.

Table 55. AES XTS algorithm functions of firmware library

Function name	Description
AES_XTS_Encrypt_Init	Initialization for AES Encryption in XTS Mode
AES_XTS_Encrypt_Append	AES Encryption in XTS Mode
AES_XTS_Encrypt_Finish	Finalization for AES Encryption in XTS Mode
AES_XTS_Decrypt_Init	Initialization for AES Decryption in XTS Mode
AES_XTS_Decrypt_Append	AES Decryption in XTS Mode
AES_XTS_Decrypt_Finish	Finalization for AES Decryption in XTS Mode

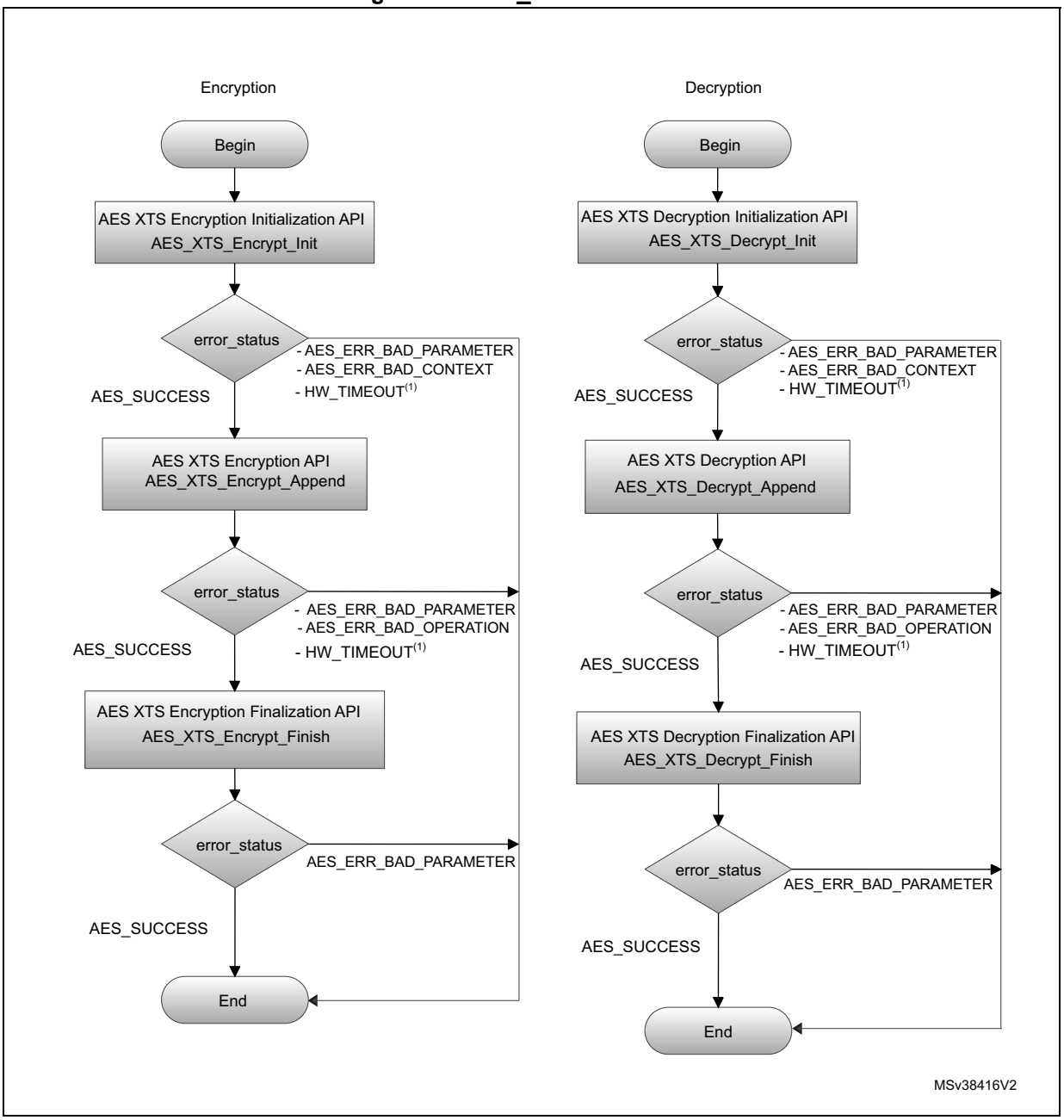
Table 56 describes the AES XTS functions of the legacy STM32 cryptographic hardware acceleration library.

Table 56. AES XTS algorithm functions of hardware acceleration library

Function name	Description
AccHw_AES_XTS_Encrypt_Init	Initialization for AES Encryption in XTS Mode
AccHw_AES_XTS_Encrypt_Append	AES Encryption in XTS Mode
AccHw_AES_XTS_Encrypt_Finish	Finalization for AES Encryption in XTS Mode
AccHw_AES_XTS_Decrypt_Init	Initialization for AES Decryption in XTS Mode
AccHw_AES_XTS_Decrypt_Append	AES Decryption in XTS Mode
AccHw_AES_XTS_Decrypt_Finish	Finalization for AES Decryption in XTS Mode

The flowcharts in *Figure 17* describe the AES_XTS algorithm.

Figure 17. AES_XTS flowcharts



1. Used only with the algorithms featuring hardware acceleration.

4.7.1 AES_XTS_Encrypt_Init function

Table 57. AES_XTS_Encrypt_Init

Function name	AES_XTS_Encrypt_Init
Prototype	`int32_t AES_XTS_Encrypt_Init(AESXTSctx_stt *P_pAESXTSctx, const uint8_t *P_pKey, const uint8_t *P_pTweak)`
Behavior	Initialization for AES XTS encryption
Parameter	– [in, out] *P_pAESXTSctx AES XTS context. – [in] *P_pKey Buffer with an XTS key (whose size is either 32 or 64 bytes). – [in] *P_pTweak Buffer with the Tweak value (it is assumed to be of 16 bytes).
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. – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

AESXTSctx_stt data structure

Table 58. AESXTSctx_stt data structure(1)

Field name	Description
`uint32_t mContextId`	Unique ID of this AES-CCM Context. Not used in current implementation.
`SKflags_et(2) mFlags`	32 bit mFlags, used to perform key schedule. Default value is E_SK_DEFAULT. See *Table 11: SKflags_et mFlags* for details.
`const uint8_t * pmKey`	Pointer to original XTS Key buffer
`const uint8_t *pmTweak`	Pointer to original Tweak buffer. The Tweak (16 bytes long) is "The 128-bit value used to represent the logical position of the data being encrypted or decrypted with XTS-AES".
`int32_t mTweakSize`	Size of the Tweak in bytes
`uint32_t amTweak[4]`	Temporary result/Tweak
`int32_t mKeySize`	Should be set with half of the size of the XTS key prior to calling this function.The following predefined values, supported by each library, can be used instead of the size of the key: – CRL_AES128_KEY – CRL_AES192_KEY – CRL_AES256_KEY

Table 58. AESXTSctx_stt data structure(1) (continued)

Field name	Description
uint32_t amExpKey [CRL_AES_MAX_EXPKEY_SIZE]	Expanded AES Key 1
uint32_t amExpKey2 [CRL_AES_MAX_EXPKEY_SIZE	Expanded AES Key 2

1. In case of using the hardware library this structure is "AccHw_AESXTSctx_stt"
2. In case of using the hardware library this structure is "AccHw_SKflags"

4.7.2 AES_XTS_Encrypt_Append function

Table 59. AES_XTS_Encrypt_Append

Function name	AES_XTS_Encrypt_Append
Prototype	*int32_t AES_XTS_Encrypt_Append (* *AESXTSctx_stt *P_pAESXTSctx,* *const uint8_t *P_pInputBuffer,* *int32_t P_inputSize,* *uint8_t *P_pOutputBuffer,* *int32_t *P_pOutputSize)*
Behavior	AES XTS Encryption function
Parameter	– [in,out] *P_pAESXTSctx AES XTS, 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 contains 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. – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: This function can be called several times, provided that P_inputSize is a multiple of 16. It can be called one last time with an input size > 16 and not multiple of 16, in this case the Ciphertext Stealing of XTS is used.

4.7.3 AES_XTS_Encrypt_Finish function

Table 60. AES_XTS_Encrypt_Finish

Function name	AES_XTS_Encrypt_Finish
Prototype	`int32_t AES_XTS_Encrypt_Finish ( AESXTSctx_stt *P_pAESXTSctx, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)`
Behavior	AES encryprion finalization of XTS Mode
Parameter	– [in,out] *P_pAESXTSctx AES XTS, already initialized, context – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that contains 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

Note: This function does not write output data and can hence be skipped. It is kept for API compatibility.

4.7.4 AES_XTS_Decrypt_Init function

Table 61. AES_XTS_Decrypt_Init

Function name	*AES_XTS_Decrypt_Init*
Prototype	`int32_t AES_XTS_Decrypt_Init(AESXTSctx_stt *P_pAESXTSctx, const uint8_t *P_pKey, const uint8_t *P_pTweak)`
Behavior	Initialization for AES XTS Decryption
Parameter	– [in,out] *P_pAESXTSctx AES XTS context – [in] *P_pKey Buffer with an XTS key (whose size is either 32 or 64 bytes) – [in] *P_pTweak Buffer with the Tweak value (it is assumed to be of 16 bytes)
Return value	– int32_t AES_XTS_Decrypt_Init(AESXTSctx_stt *P_pAESXTSctx, const uint8_t *P_pKey, const uint8_t *P_pTweak) – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

4.7.5 AES_XTS_Decrypt_Append function

Table 62. AES_XTS_Decrypt_Append

Function name	AES_XTS_Decrypt_Append

Prototype	`int32_t AES_XTS_Decrypt_Append (`
	`AESXTSctx_stt *P_pAESXTSctx,`
	`const uint8_t *P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t *P_pOutputBuffer,`
	`int32_t *P_pOutputSize)`

Behavior	AES descryprion function
Parameter	– [in,out] *P_pAESXTSctx AES XTS, 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 contains 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 – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: This function can be called multiple times, provided that P_inputSize is a multiple of 16. It can be called one last time with an input size > 16 and not multiple of 16, in this case the Ciphertext Stealing of XTS is used.

4.7.6 AES_XTS_Decrypt_Finish function

Table 63. AES_XTS_Decrypt_Finish function

Function name	AES_XTS_Decrypt_Finish function

Prototype	`int32_t AES_XTS_Decrypt_Finish (`
	`AESXTSctx_stt *P_pAESXTSctx,`
	`uint8_t *P_pOutputBuffer,`
	`int32_t *P_pOutputSize)`

Behavior	AES descryprion finalization of XTS Mode

Table 63. AES_XTS_Decrypt_Finish function

Function name	AES_XTS_Decrypt_Finish function
Parameter	– [in,out] *P_pAESXTSctx AES XTS, already initialized, context. – [out] *P_pOutputBuffer: Output buffer. – [out] *P_pOutputSize: Pointer to integer that contains 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.

Note: This is a wrapper for AES_XTS_Encrypt_Finish as the XTS Mode is equal in encryption and decryption.

4.8 AES CBC enciphering and deciphering example

The following code gives a simple example of how to use AES-CBC of the legacy STM32 cryptographic firmware library.

```
#include "main.h"
const uint8_t Plaintext[PLAINTEXT_LENGTH] ={0x6b, 0xc1, 0xbe, 0xe2, 0x2e,
0x40};
/* Key to be used for AES encryption/decryption */
uint8_t Key[CRL_AES192_KEY] ={0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64,
0x52,0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,0x62, 0xf8, 0xea, 0xd2,
0x52, 0x2c, 0x6b, 0x7b};
/* Initialization Vector */
uint8_t IV[CRL_AES_BLOCK] ={0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
/* Buffer to store the output data */
uint8_t OutputMessage[PLAINTEXT_LENGTH];
/* Size of the output data */
uint32_t OutputMessageLength = 0;
int main(void)
{
  AESCBCctx_stt AESctx;
  uint32_t error_status = AES_SUCCESS;
  int32_t outputLength = 0;
  /* Set flag field to default value */
  AESctx.mFlags = E_SK_DEFAULT;
  /* Set key size to 24 (corresponding to AES-192) */
  AESctx.mKeySize = 24;
  /* Set iv size field to IvLength*/
  AESctx.mIvSize = IvLength;
  /* Initialize the operation, by passing the key.
   * Third parameter is NULL because CBC doesn't use any IV */
  error_status = AES_CBC_Encrypt_Init(&AESctx, AES192_Key,
InitializationVector );
  /* check for initialization errors */
  if (error_status == AES_SUCCESS)
  {
    /* Encrypt Data */
    error_status =
AES_CBC_Encrypt_Append(&AESctx,InputMessage,InputMessageLength,
OutputMessage,&outputLength);
    if (error_status == AES_SUCCESS)
    {
      /* Write the number of data written*/
      *OutputMessageLength = outputLength;
      /* Do the Finalization */
      error_status = AES_CBC_Encrypt_Finish(&AESctx, OutputMessage +
*OutputMessageLength, &outputLength);
```

```
        *OutputMessageLength += outputLength;
      }
    }

    return error_status;
  }
```

5 ARC4 algorithm

5.1 ARC4 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 XOR-ed with the generated key sequence. The algorithm is serial, as it requires successive exchanges of state entries based on the key sequence.

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

This algorithm can run with all STM32 microcontrollers using a software algorithm implementation.

For ARC4 library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For ARC4 library performance and memory requirements, refer to the legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\Fw_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.

5.2 ARC4 library functions

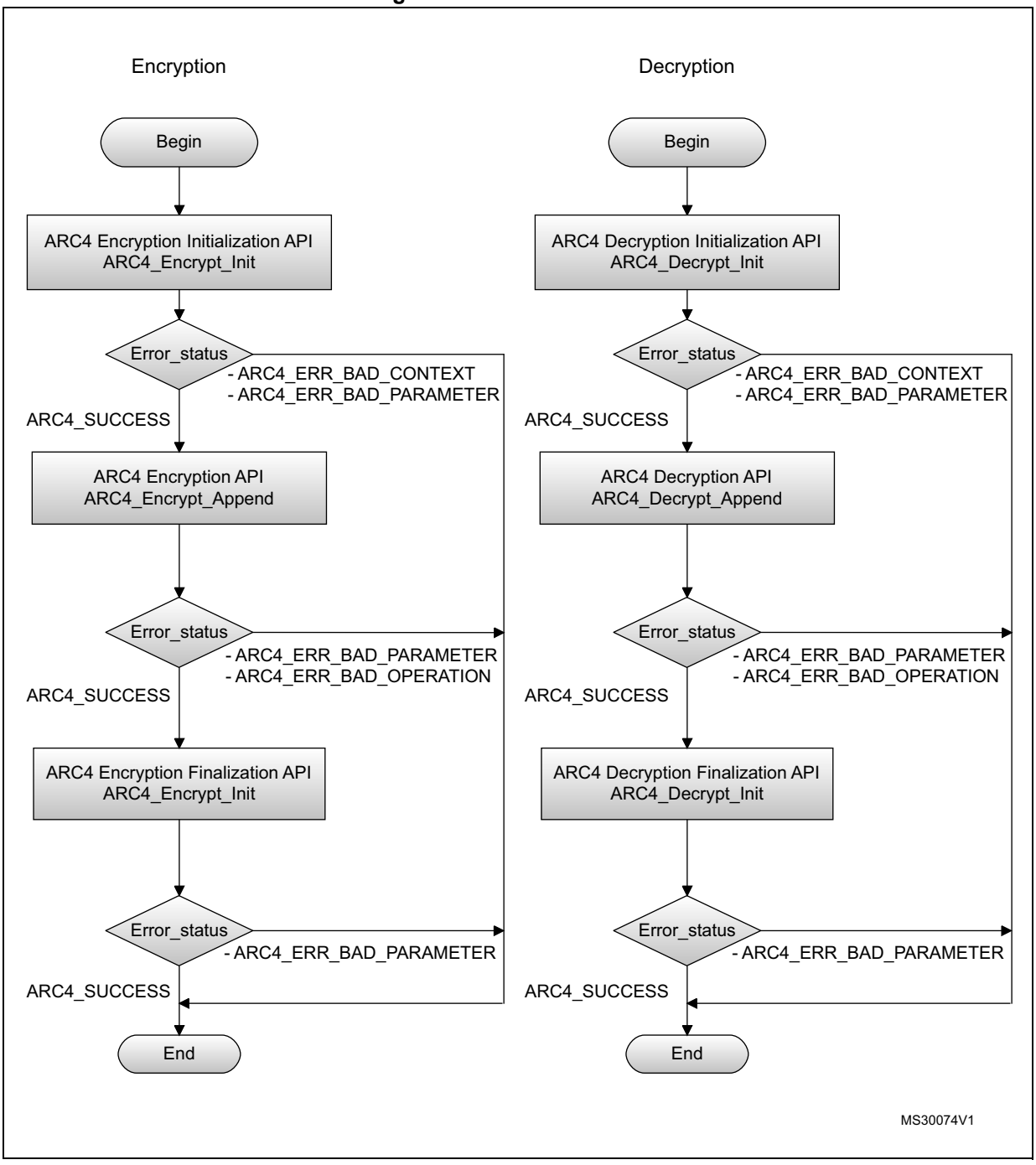
Table 64 describes the ARC4 functions of the firmware cryptographic library.

Table 64. ARC4 algorithm functions of firmware library

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 flowcharts provided in *Figure 18* describe the ARC4 algorithm.

Figure 18. ARC4 flowcharts



5.2.1 ARC4_Encrypt_Init function

Table 65. 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.
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.

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

The IV is not used in ARC4, so the value of P_pIv is not checked or used.

5.2.2 ARC4_Encrypt_Append function

Table 66. 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 contains 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 several times.

ARC4ctx_stt data structure

Structure describing an ARC4 content.

Table 67. 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 *Table 11: 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 68. 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 contains 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 69. 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.
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.

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

The IV is not used in ARC4, so the value of P_pIv is not checked or used.

5.2.5 ARC4_Decrypt_Append function

Table 70. 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 contains 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 several times.

5.2.6 ARC4_Decrypt_Finish function

Table 71 describes ARC4_Decrypt_Finish function.

Table 71. 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 contains 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 does not write output data and can hence be skipped. It is kept for API compatibility.

5.3 ARC4 example

The following code give a simple example how to use ARC4 of the legacy STM32 cryptographic firmware library.

```
#include "main.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 CHACHA20 algorithm

6.1 CHACHA20 description

CHACHA20 is a symmetric cipher algorithm, it is a variant of salsa20 designed by Dan J. Bernstein. This implementation of ChaCha supports only keys of 128 or 256 bit and it make 20 round for 512 bit of Plaintext by altering Column and diagonal round of the input matrix to produce 512 bit of cipher text.

This algorithm can run with all the STM32 Series using a software algorithm implementation.

For CHACHA20 library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For CHACHA20 library performances and memory requirements, refer to the legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\Fw_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.

6.2 CHACHA20 library functions

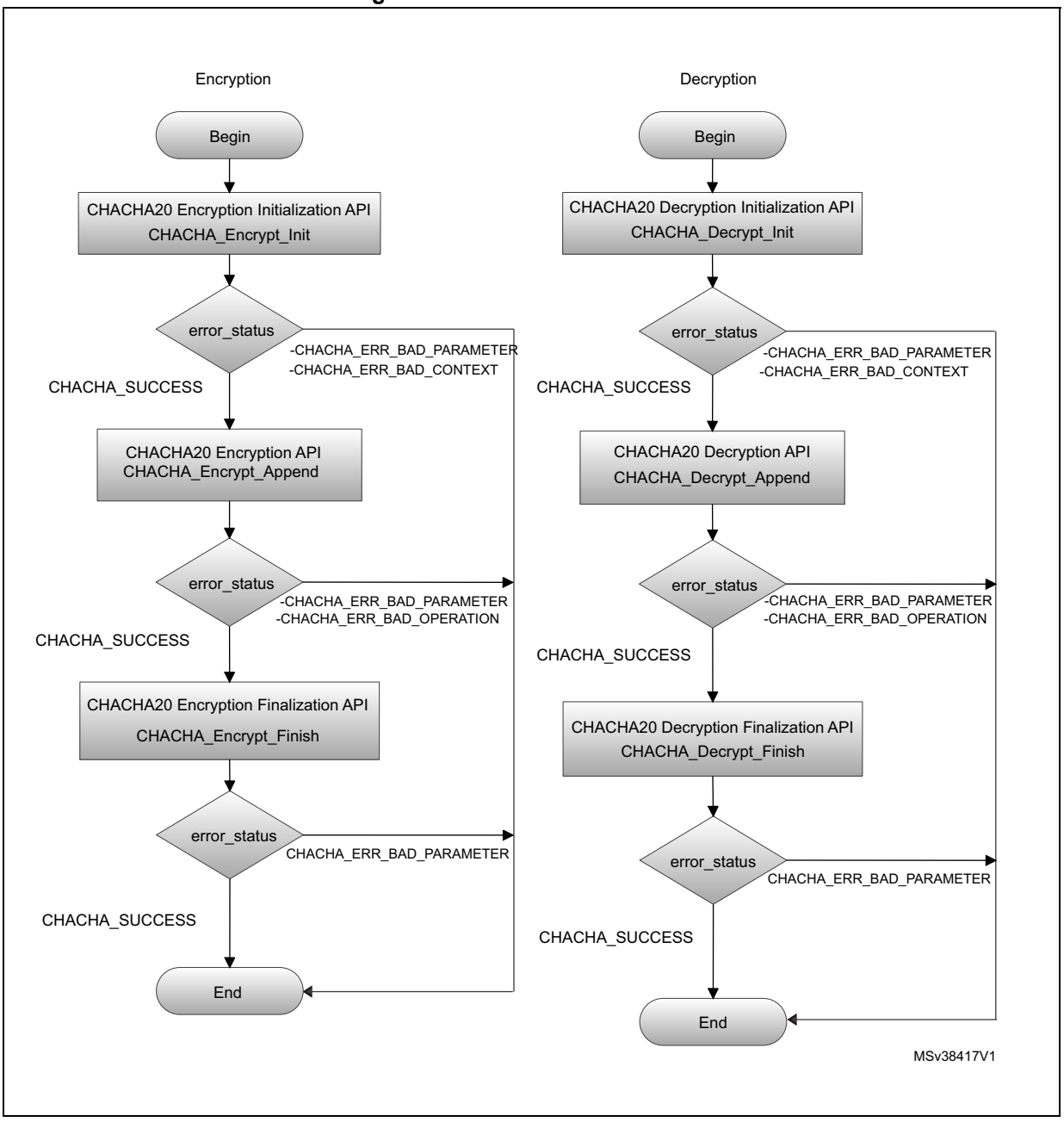
Table 72 describes the CHACHA20 functions of the firmware cryptographic library.

Table 72. CHACHA20 algorithm functions of the firmware library

Function name	Description
`CHACHA_Encrypt_Init`	Initialization for CHACHA Encryption
`CHACHA_Encrypt_Append`	CHACHA Encryption
`CHACHA_Encrypt_Finish`	CHACHA Encryption Finalization
`CHACHA_Decrypt_Init`	Initialization for CHACHA Decryption
`CHACHA_Decrypt_Append`	CHACHA Decryption
`CHACHA_Decrypt_Finish`	CHACHA Decryption Finalization

The flowcharts in *Figure 19* describe the CHACHA20 algorithms.

Figure 19. CHACHA20 flowcharts



6.2.1 CHACHA_Encrypt_Init

Table 73. CHACHA_Encrypt_Init

Function name	CHACHA_Encrypt_Init
Prototype	`int32_t CHACHA_Encrypt_Init (CHACHActx_stt *P_pCHACHActx, const uint8_t *P_pKey, const uint8_t *P_pIv)`
Behavior	Initialization for CHACHA Encryption
Parameter	– [in, out] *P_pCHACHActx Chacha context – [in] *P_pKey Buffer with the Key – [in] *P_pIv Buffer with the IV
Return value	– CHACHA_SUCCESS Operation successful – CHACHA_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer – CHACHA_ERR_BAD_CONTEXT Context was not initialized with valid values

Note: P_pCHACHActx.mKeySize (see CHACHActx_stt) must be set with the byte size of the key prior to calling this function. This implementation of ChaCha supports keys of 128 or 256 bit only.

P_pCHACHActx.mIvSize allows two values:

- *8: in this case Nonce is 64 bit, counter is 64 bit (as in the original specification).*
- *12: in this case Nonce is 96 bit, Counter is 32 bit.*

CHACHActx_stt data structure

Table 74. CHACHActx_stt data structure

Function name	Description
`CHACHA_Encrypt_Init`	Initialization for CHACHA Encryption
`CHACHA_Encrypt_Append`	CHACHA Encryption
`CHACHA_Encrypt_Finish`	CHACHA Encryption Finalization
`CHACHA_Decrypt_Init`	Initialization for CHACHA Decryption
`CHACHA_Decrypt_Append`	CHACHA Decryption
`CHACHA_Decrypt_Finish`	CHACHA Decryption Finalization

6.2.2 CHACHA_Encrypt_Append

Table 75. CHACHA_Encrypt_Append

Function name	CHACHA_Encrypt_Append

Prototype	`int32_t CHACHA_Encrypt_Append (CHACHActx_stt *P_pCHACHActx,`
	`const uint8_t *P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t *P_pOutputBuffer,`
	`int32_t *P_pOutputSize);`

Behavior	CHACHA encryption
Parameter	– [in, out] *P_pCHACHActx ChaCha, 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 contains the size of written output data, expressed in bytes.
Return value	– CHACHA_SUCCESS Operation successful. – CHACHA_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer. – CHACHA_ERR_BAD_OPERATION Append can't be called after a Final.

Note: This function can be called several times, provided that P_inputSize is a multiple of 64. A single, final, call with P_inputSize not multiple of 64 is allowed

6.2.3 CHACHA_Encrypt_Finish

Table 76. CHACHA_Encrypt_Finish

Function name	CHACHA_Encrypt_Finish

Prototype	`int32_t CHACHA_Encrypt_Finish (CHACHActx_stt *P_pCHACHActx,`
	`uint8_t *P_pOutputBuffer,`
	`int32_t *P_pOutputSize)`

Behavior	CHACHA Encryption Finalization
Parameter	– [in, out] *P_pCHACHActx ChaCha, already initialized, context. – [out] *P_pOutputBuffer Output buffer. – [out] *P_pOutputSize Pointer to integer that contains the size of written output data, expressed in bytes.
Return value	– CHACHA_SUCCESS Operation successful. – CHACHA_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer.

Note: This function does not write output data and can hence be skipped. It is kept for API compatibility.

6.2.4 CHACHA_Decrypt_Init

Table 77. CHACHA_Decrypt_Init

Function name	CHACHA_Decrypt_Init
Prototype	`int32_t CHACHA_Decrypt_Init (CHACHActx_stt *P_pCHACHActx, const uint8_t *P_pKey, const uint8_t *P_pIv)`
Behavior	Initialization for CHACHA Decryption
Parameter	– [in, out] *P_pCHACHActx ChaCha context. – [in] *P_pKey Buffer with the Key. – [in] *P_pIv Buffer with the IV.
Return value	– CHACHA_SUCCESS Operation successful. – CHACHA_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer. – CHACHA_ERR_BAD_CONTEXT Context was not initialized with valid values, see the note below.

Note: P_pCHACHActx.mKeySize (see CHACHActx_stt) must be set with the byte size of the key prior to calling this function. This implementation of ChaCha supports keys of 128 or 256 bit only.

P_pCHACHActx.mIvSize allows two values:

- *8: in this case Nonce is 64 bit, counter is 64 bit (as in the original specification)*
- *12: in this case Nonce is 96 bit, Counter is 32 bit*

This function is just a wrapper for CHACHA_Encrypt_Init.

6.2.5 CHACHA_Decrypt_Append

Table 78. CHACHA_Decrypt_Append

Function name	CHACHA_Decrypt_Append

Prototype	`int32_t CHACHA_Decrypt_Append (CHACHActx_stt *P_pCHACHActx,`
	`const uint8_t *P_pInputBuffer,`
	`int32_t P_inputSize,`
	`uint8_t *P_pOutputBuffer,`
	`int32_t *P_pOutputSize`

Behavior	CHACHA decryption

Table 78. CHACHA_Decrypt_Append (continued)

Function name	CHACHA_Decrypt_Append
Parameter	– [in, out] *P_pCHACHActx ChaCha, 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 contains the size of written output data, expressed in bytes.
Return value	– CHACHA_SUCCESS Operation successful. – CHACHA_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer. – CHACHA_ERR_BAD_OPERATION Append can't be called after a Final.

Note: This function can be called several times, provided that P_inputSize is a multiple of 64. A single, final, call with P_inputSize not multiple of 64 is allowed.

This function is just a wrapper for CHACHA_Encrypt_Append.

6.2.6 CHACHA_decrypt_Finish

Table 79. CHACHA_decrypt_Finish

Function name	CHACHA_decrypt_Finish

Prototype	`int32_t CHACHA_Decrypt_Finish (CHACHActx_stt *P_pCHACHActx,`
	`  uint8_t  *P_pOutputBuffer,`
	`  int32_t  *P_pOutputSize)`

Behavior	CHACHA Decryption Finalization
Parameter	– [in, out] *P_pCHACHActx ChaCha, already initialized, context. – [out] *P_pOutputBuffer Output buffer. – [out] *P_pOutputSize Pointer to integer that contains the size of written output data, expressed in bytes.
Return value	– CHACHA_SUCCESS Operation successful. – CHACHA_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer.

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

This function is just a wrapper for CHACHA_Encrypt_Finish.

6.3 CHACHA20 example

The following code give a simple example how to use CHACHA20 of the legacy STM32 cryptographic firmware library.

```
#include "main.h"
/* Key to be used for encryption/decryption */
const uint8_t Key[CRL_CHACHA256_KEY] ={0x00, 0x11, 0x22, 0x33, 0x44, 0x55,
0x66, 0x77,
0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff,0xff, 0xee, 0xdd, 0xcc, 0xbb,
0xaa, 0x99, 0x88,0x77, 0x66, 0x55, 0x44, 0x33, 0x22, 0x11, 0x00};
/* Initialization Vector */
const uint8_t IV [CRL_CHACHA_NONCE] = {0x0f, 0x1e, 0x2d, 0x3c, 0x4b, 0x5a};
/* Test Vectors have zero input, we need to replicate */
uint8_t Plaintext[PLAINTEXT_LENGTH] = {0};
/* Buffer to store the output data */
uint8_t OutputMessage[PLAINTEXT_LENGTH];
/* Size of the output data */
uint32_t OutputMessageLength = 0;
int main(void)
{
  int32_t error_status;
  int32_t outputLength; /* Will store the size of written data */
  CHACHActx_stt chacha20ctx_st; /* Will keep the ChaCha20 Context */
  /* First Initialize the Context */
  chacha20ctx_st.mFlags = E_SK_DEFAULT; /* Default Flags */
  chacha20ctx_st.mIvSize = 8; /* IV of 8 bytes */
  chacha20ctx_st.mKeySize = 16; /* Key of 16 bytes */
  /* Call Init */
  error_status = CHACHA_Encrypt_Init (&chacha20ctx_st, Key,
InitializationVector);
  /* check for initialization errors */
  if (error_status == CHACHA_SUCCESS)
  {
    /* Call Append. It will process inout and generate output */
    error_status = CHACHA_Encrypt_Append (&chacha20ctx_st, InputMessage,
InputMessageLength, OutputMessage, &outputLength);
    if (error_status == CHACHA_SUCCESS )
    {
s      *OutputMessageLength = outputLength;
/* Call Finish. For chacha20, this is useless, as all output has been
already generated */
      error_status = CHACHA_Encrypt_Finish (&chacha20ctx_st, OutputMessage
+ *OutputMessageLength, &outputLength);
      /* add data written to the information to be returned */
      *OutputMessageLength += outputLength;
```

```
    }
  }
  return error_status;
}
```

7 CHACHA20-POLY1305

7.1 CHACHA20-POLY1305 description

CHACHA20-POLY1305 is an AEAD algorithm that combines the following two primitives:

- ChaCha20 for Encryption (for more details about ChaCha20 algorithm please refer ChaCha20 section).
- Poly1305 for Athentication (for more details about Poly1305 algorithm please refer Poly1305 section).

This algorithm can run on all STM32 microcontrollers using a software algorithm implementation.

For CHACHA20 library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For CHACHA20 library performances and memory requirements, refer to the legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\Fw_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.

7.2 CHACHA20-POLY1305 library functions

Table 80 describes the CHACHA20-POLY1305 functions of the firmware cryptographic library.

Table 80. ChaCha20-Poly1305 algorithm functions of the firmware library

Function name	Description
`ChaCha20Poly1305_Encrypt_Init`	Initialization for ChaCha20-Poly1305 AEAD Encryption Algorithm
`ChaCha20Poly1305_Encrypt_Append`	ChaCha20-Poly1305 AEAD Encryption processing function
`ChaCha20Poly1305_Header_Append`	ChaCha20-Poly1305 AAD (Additional Authenticated Data) processing function
`ChaCha20Poly1305_Encrypt_Finish`	ChaCha20-Poly1305 TAG generation function
`ChaCha20Poly1305_Decrypt_Init`	Initialization for ChaCha20-Poly1305 AEAD Decryption Algorithm
`ChaCha20Poly1305_Decrypt_Append`	ChaCha20-Poly1305 AEAD Decryption processing function
`ChaCha20Poly1305_Decrypt_Finish`	ChaCha20-Poly1305 Authentication TAG verification function

The flowcharts in *Figure 20* and *Figure 21* describe the CHACHA20-POLY1305 algorithms.

Figure 20. CHACHA20-Poly1305 encryption flowchart

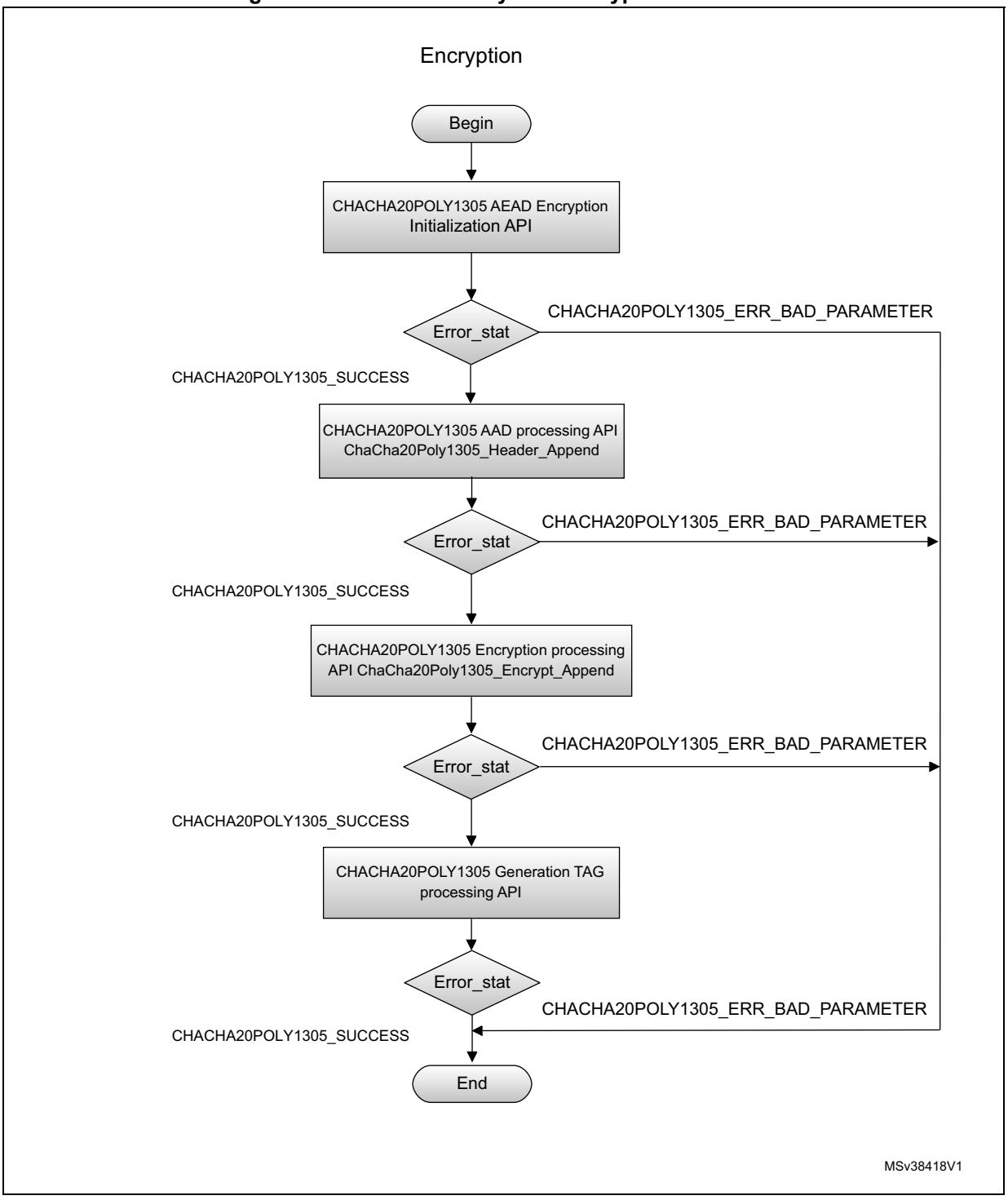
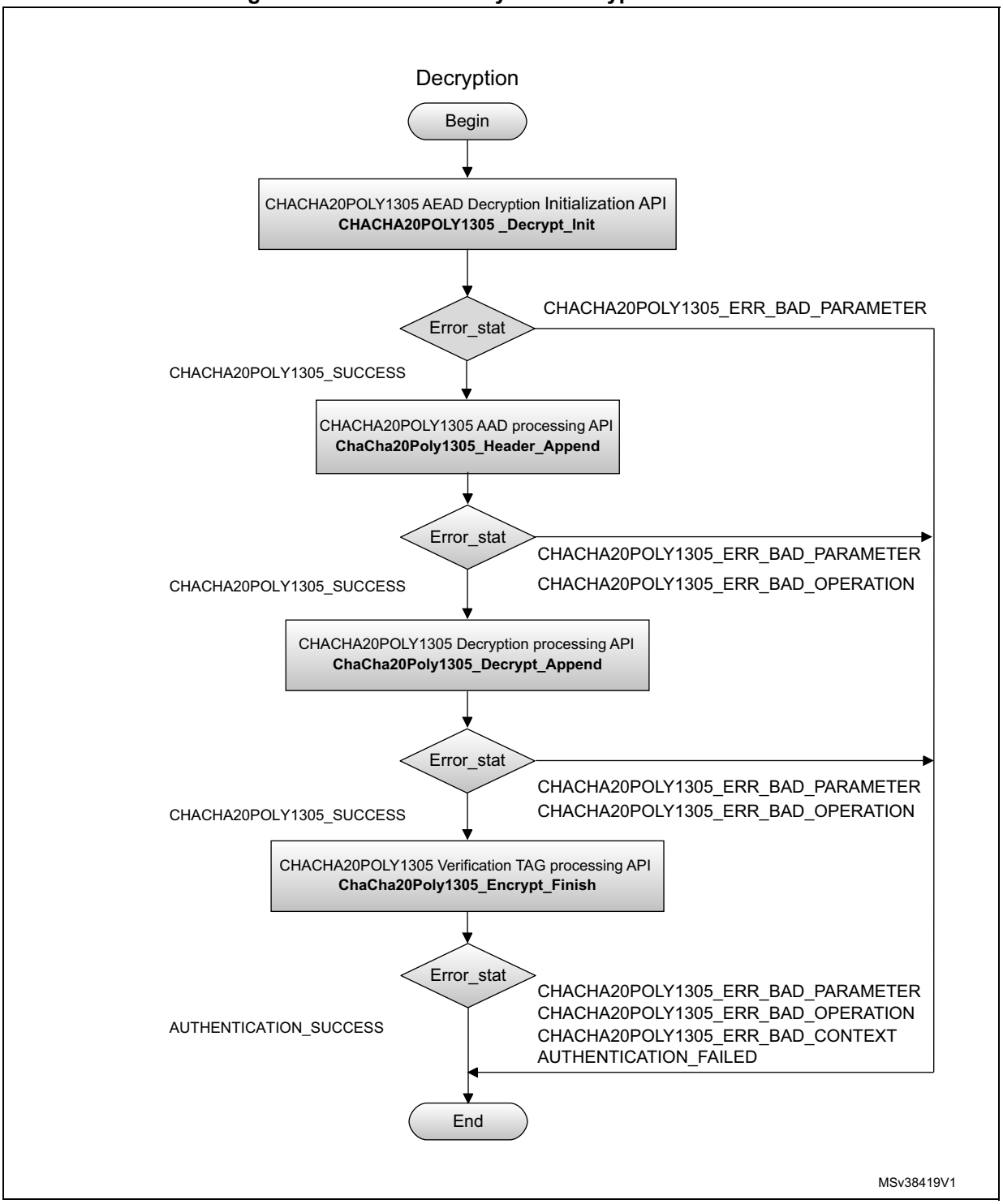


Figure 21. CHACHA20-Poly1305 decryption flowchart



7.2.1 ChaCha20Poly1305_Encrypt_Init

Table 81. ChaCha20Poly1305_Encrypt_Init

Function name	ChaCha20Poly1305_Encrypt_Init
Prototype	`int32_t ChaCha20Poly1305_Encrypt_Init (ChaCha20Poly1305ctx_stt *P_pChaCha20Poly1305ctx, const uint8_t *P_pKey, const uint8_t *P_pNonce)`
Behavior	Initialization for ChaCha20-Poly1305 AEAD Encryption Algorithm
Parameter	– [in,out] *P_pChaCha20Poly1305ctx ChaCha20-Poly1305 context. – [in] *P_pKey ChaCha20-Poly1305 32 byte key. – [in] *P_pNonce ChaCha20-Poly1305 12 byte Nonce (Number used Once).
Return value	– CHACHA20POLY1305_SUCCESS Operation successful. – CHACHA20POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer.

Note: Output TAG consists of 16 bytes.

7.2.2 ChaCha20Poly1305_Encrypt_Append

Table 82. ChaCha20Poly1305_Encrypt_Append

Function name	ChaCha20Poly1305_Encrypt_Append
Prototype	`int32_t ChaCha20Poly1305_Encrypt_Append (ChaCha20Poly1305ctx_stt *P_pChaCha20Poly1305ctx, const uint8_t *P_pInputBuffer, int32_t P_inputSize, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)`
Behavior	ChaCha20-Poly1305 AEAD Encryption processing function
Parameter	– [in,out] *P_pChaCha20Poly1305ctx ChaCha20-Poly1305 context (Initialized with \ref ChaCha20Poly1305_Encrypt_Init). – [in] *P_pInputBuffer Data to be authenticated and encrypted. – [in] P_inputSize Size (in bytes) of data at P_pInputBuffer. – [out] *P_pOutputBuffer Output Buffer. – [out] *P_pOutputSize Pointer to integer that contains the size of written output data.
Return value	– POLY1305_SUCCESS Operation successful. – POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer. – POLY1305_ERR_BAD_OPERATION Call to \ref ChaCha20Poly1305_Encrypt_Append is not allowed.

Note: This function can be called several times with P_inputSize multiple of 64 bytes. A single, last, call can be made with any value for P_inputSize.

7.2.3 ChaCha20Poly1305_Header_Append

Table 83. ChaCha20Poly1305_Header_Append

Function name	ChaCha20Poly1305_Header_Append
Prototype	`int32_t ChaCha20Poly1305_Header_Append (ChaCha20Poly1305ctx_stt *P_pChaCha20Poly1305ctx, const uint8_t *P_pInputBuffer, int32_t P_inputSize)`
Behavior	ChaCha20-Poly1305 AEAD Encryption processing function
Parameter	– [in,out] *P_pChaCha20Poly1305ctx ChaCha20-Poly1305 context (Initialized with ChaCha20Poly1305_Encrypt_Init or ChaCha20Poly1305_Decrypt_Init). – [in] *P_pInputBuffer Data to be authenticated but not encrypted/decrytped. – [in] P_inputSize Size (in bytes) of data at P_pInputBuffer.
Return value	– CHACHA20POLY1305_SUCCESS Operation successful. – CHACHA20POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer. – CHACHA20POLY1305_ERR_BAD_OPERATION Call to \ref ChaCha20Poly1305_Header_Append is not allowed.

Note: This function can be called several times with P_inputSize multiple of 16 bytes. A single, last, call can be made with any value for P_inputSize.

7.2.4 ChaCha20Poly1305_Encrypt_Finish

Table 84. ChaCha20Poly1305_Encrypt_Finish

Function name	ChaCha20Poly1305_Encrypt_Finish
Prototype	`int32_t ChaCha20Poly1305_Encrypt_Finish (ChaCha20Poly1305ctx_stt *P_pChaCha20Poly1305ctx, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)`
Behavior	ChaCha20-Poly1305 TAG generation function
Parameter	– [in,out] *P_pChaCha20Poly1305ctx ChaCha20-Poly1305 context. – [in] *P_pOutputBuffer where the TAG is written. – [in] *P_pOutputSize contains the size of the written tag (=16).
Return value	– CHACHA20POLY1305_SUCCESS Operation successful. – CHACHA20POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer. – CHACHA20POLY1305_ERR_BAD_OPERATION Call to \ref ChaCha20Poly1305_Header_Append is not allowed.

Note: This function writes a 16-byte TAG.

7.2.5 ChaCha20Poly1305_Decrypt_Init

Table 85. ChaCha20Poly1305_Decrypt_Init

Function name	ChaCha20Poly1305_Decrypt_Init
Prototype	`int32_t ChaCha20Poly1305_Decrypt_Init (ChaCha20Poly1305ctx_stt *P_pChaCha20Poly1305ctx, const uint8_t *P_pKey, const uint8_t *P_pNonce)`
Behavior	Initialization for ChaCha20-Poly1305 AEAD Decryption Algorithm
Parameter	– [in,out] *P_pChaCha20Poly1305ctx ChaCha20-Poly1305 context. – [in] *P_pKey ChaCha20-Poly1305 32 byte key. – [in] *P_pNonce ChaCha20-Poly1305 12 byte Nonce (Number used Once).
Return value	– CHACHA20POLY1305_SUCCESS Operation successful. – CHACHA20POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer.

Note: P_pChaCha20Poly1305ctx.pmTag must be written by the caller before calling ChaCha20Poly1305_Decrypt_Finish.

This function is just a wrapper for ChaCha20Poly1305_Encrypt_Init.

7.2.6 ChaCha20Poly1305_Decrypt_Append

Table 86. ChaCha20Poly1305_Decrypt_Append

Function name	ChaCha20Poly1305_Decrypt_Append
Prototype	`int32_t ChaCha20Poly1305_Decrypt_Append (ChaCha20Poly1305ctx_stt *P_pChaCha20Poly1305ctx, const uint8_t *P_pInputBuffer, nt32_t P_inputSize, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)`
Behavior	Initialization for ChaCha20-Poly1305 AEAD Decryption Algorithm
Parameter	– [in,out] *P_pChaCha20Poly1305ctx ChaCha20-Poly1305 context (Initialized with \ref ChaCha20Poly1305_Encrypt_Init). – [in] *P_pInputBuffer Data to be authenticated and decrypted. – [in] P_inputSize Size (in bytes) of data at P_pInputBuffer. – [out] *P_pOutputBuffer Output Buffer. – [out] *P_pOutputSize Pointer to integer that contains the size of written output data.
Return value	– CHACHA20POLY1305_SUCCESS Operation successful. – CHACHA20POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer.

Note: This function can be called several times with P_inputSize multiple of 64 bytes. A single last call can be made with any value for P_inputSize.

7.2.7 ChaCha20Poly1305_Decrypt_Finish

Table 87. ChaCha20Poly1305_Decrypt_Finish

Function name	ChaCha20Poly1305_Decrypt_Finish
Prototype	`int32_t ChaCha20Poly1305_Decrypt_Finish (ChaCha20Poly1305ctx_stt *P_pChaCha20Poly1305ctx, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)`
Behavior	ChaCha20-Poly1305 Authentication TAG verification function
Parameter	– [in,out] *P_pChaCha20Poly1305ctx ChaCha20-Poly1305 context (Initialized with \ref ChaCha20Poly1305_Encrypt_Init). – [in] *P_pInputBuffer Data to be authenticated and decrypted. – [in] P_inputSize Size (in bytes) of data at P_pInputBuffer. – [out] *P_pOutputBuffer Output Buffer. – [out] *P_pOutputSize Pointer to integer that contains the size of written output data.
Return value	– CHACHA20POLY1305_SUCCESS Operation successful. – CHACHA20POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer.

Note: This function requires that P_pChaCha20Poly1305ctx->pmTag is set by the caller and points to the TAG to be checked.

The caller should just check the return value to be equal (or different) to AUTHENTICATION_SUCCESSFUL and accept (reject) the message/tag pair accordingly.

7.3 CHACHA20-POLY1305 example

The following code give a simple example how to use CHACHA20-POLY1305 of the legacy STM32 cryptographic firmware library.

```
#include "main.h"
const uint8_t key[] ={0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88,
0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f,0x90, 0x91, 0x92, 0x93, 0x94, 0x95,
0x96, 0x97, 0x98,0x99, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f};
const uint8_t nonce[] ={0x07, 0x00, 0x00, 0x00, 0x40, 0x41, 0x42, 0x43,
0x44, 0x45};
const uint8_t input[] ={0x4c, 0x61, 0x64, 0x69, 0x65, 0x73, 0x20, 0x61,
0x6e, 0x64};
const uint8_t aad[] ={0x50, 0x51, 0x52, 0x53, 0xc0, 0xc1, 0xc2, 0xc3, 0xc4,
0xc5};
uint8_t outputBuffer[265];
int main()
{
/* ChaCha20Poly1305, error status and output length */
ChaCha20Poly1305ctx_stt ctx;
/* Default value for error status */
uint32_t error_status = CHACHA20POLY1305_SUCCESS;
/* Integer to store size of written data */
int32_t outputLength = 0;
/* Set flag field to default value */
  ctx.mFlags = E_SK_DEFAULT;
  /* Call the Init */
  error_status = ChaCha20Poly1305_Encrypt_Init(&ctx, Key, Nonce);
  if (error_status == CHACHA20POLY1305_SUCCESS)
  {
    /* Process the AAD */
    error_status = ChaCha20Poly1305_Header_Append(&ctx, AAD, AADSize);
    if (error_status == CHACHA20POLY1305_SUCCESS)
    {
      /* Encrypt Message */
      error_status = ChaCha20Poly1305_Encrypt_Append(&ctx, InputMessage,
InputMessageSize, Output, &outputLength);
      if (error_status == CHACHA20POLY1305_SUCCESS)
      {
        /* Generate authentication tag */
        error_status = ChaCha20Poly1305_Encrypt_Finish(&ctx, Output +
outputLength, &outputLength);
      }
    }
  }

  return error_status;
}
```

8 Curve 25519 algorithm

8.1 Curve 25519 description

Curve25519 is a state-of-the-art Diffie-Hellman function, it can be used to exchange messages between two users.

Using 32 byte secret key, Curve25519 generates a 32 byte public key. Then given the user 32-byte secret key and the other user 32-byte public key, the curve25519 algorithm generates a 32-byte secret key shared by the two users, to be used to authenticate and encrypt messages between them.

This algorithm can run with all STM32 microcontrollers using a software algorithm implementation.

For Curve25519 library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For Curve25519 library performance and memory requirements, refer to the legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\Fw_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.

8.2 Curve 25519 library functions

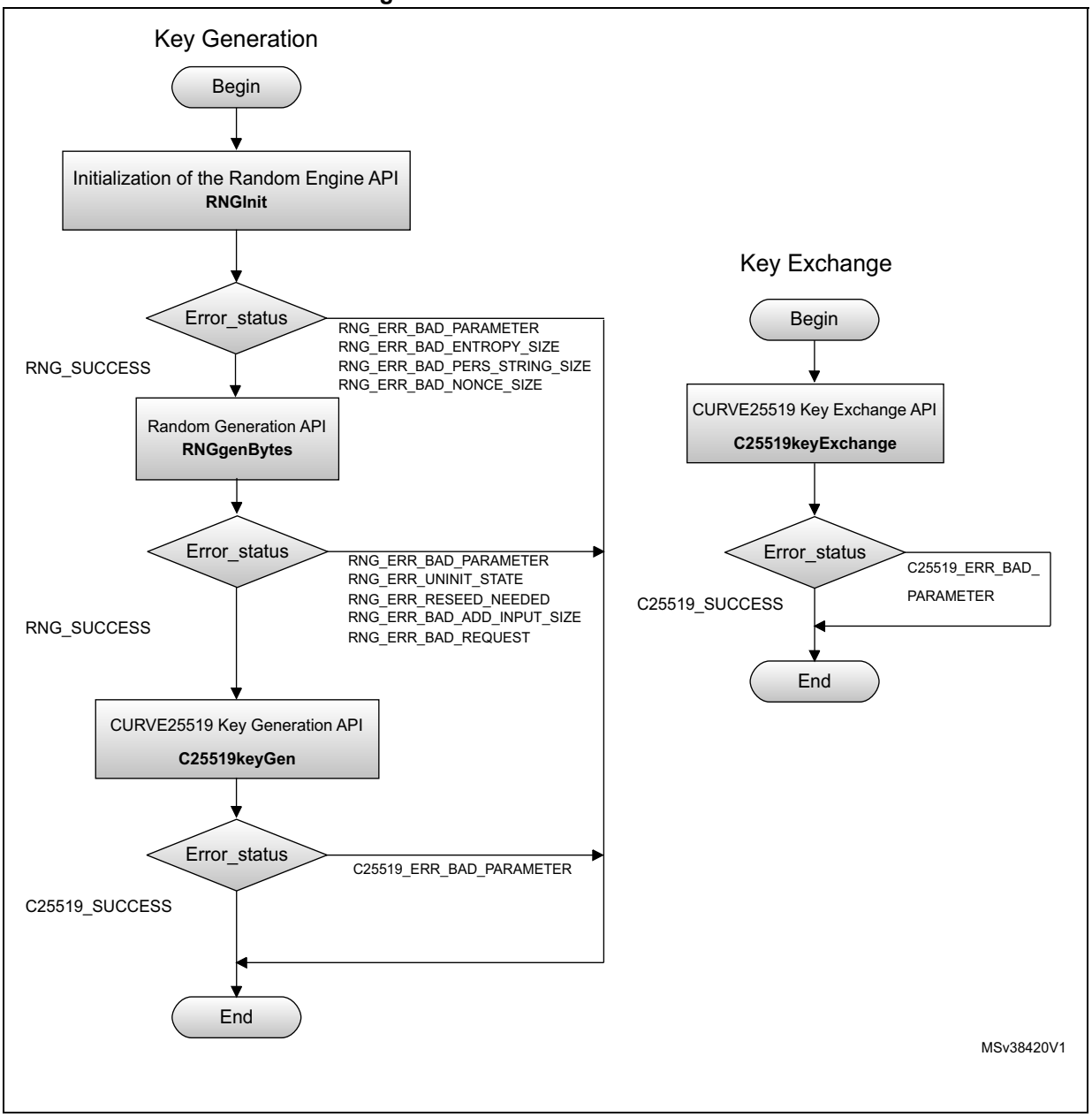
Table 88 describes the Curve25519 functions of the firmware cryptographic library.

Table 88. Curve25519 algorithm functions of the firmware library

Function name	Description
`C25519keyGen`	Generate a public key using the secret key
`C25519keyExchange`	Generate the shared secret using the secret and public keys

The flowcharts provided in *Figure 22* describe the Cuvre25519 algorithms.

Figure 22. Curve25519 flowcharts



8.2.1 C25519keyGen

Table 89. C25519keyGen

Function name	C25519keyGen
Prototype	`int32_t C25519keyGen (uint8_t *P_pPrivateKey, uint8_t *P_pPublicKey)`
Behavior	Generate a public key using the secret key
Parameter	– [in,out] *P_pPrivateKey Buffer of 32 random bytes, which holds the Private Key – [out] *P_pPublicKey Buffer of 32 bytes which holds the Public Key
Return value	– C25519_SUCCESS: Key Pair generated successfully – C25519_ERR_BAD_PARAMETER: P_pPrivateKey == NULL or P_pPublicKey == NULL

Note: P_pPrivateKey must contain 32 random bytes that is used for the key pair generation.

8.2.2 C25519keyExchange

Table 90. C25519keyExchange

Function name	C25519keyExchange
Prototype	`nt32_t C25519keyExchange (uint8_t *P_pSharedSecret, const uint8_t *P_pPrivateKey, const uint8_t *P_pPublicKey)`
Behavior	Generate a public key using the secret key
Parameter	– [out] *P_pSharedSecret Buffer of 32 bytes which stores the shared secret. – [in] *P_pPrivateKey Buffer of 32 bytes containing the Private Key. – [in] *P_pPublicKey Buffer of 32 bytes containing the other party's Public Key.
Return value	– C25519_SUCCESS Operation successful. – C25519_ERR_BAD_PARAMETER P_pPrivateKey == NULL or P_pSharedSecret == NULL or P_pPublicKey == NULL.

8.3 Curve25519 example

The following code give a simple example how to use Curve25519 of the legacy STM32 cryptographic firmware library.

```
#include "main.h"
const uint8_t Secret_Key_A[] ={0x07, 0x18, 0xA5, 0x7D, 0x3C, 0x16, 0x72};
const uint8_t Public_Key_A[] ={0x85, 0x20, 0xF0,0x74, 0x8B, 0x7D, 0xDC,};
const uint8_t Secret_Key_B[] ={, 0xAB, 0x08, 0x7E, 0x62, 0x4A, 0x8A, 0x4B};
const uint8_t Public_Key_B[] ={0xDE, 0x9E, 0xDB, 0x7D, 0x7B};
/* Generator, used in key generation from a
given private key using C25519keyExchange function */
const uint8_t G[32] ={0x09};
uint8_t result_buffer[32]; /* We'll store result here */
uint8_t private_key[32]; /* Buffer for the private key*/
uint8_t public_key[32]; /* Buffer for the public key */
int main(void)
{
  int32_t status = C25519_SUCCESS ;
  /* Curve25519 Keys Exchange */
  status = C25519keyExchange(result_buffer, Secret_Key_A, G);
  if (status == C25519_SUCCESS)
  {
      status = C25519keyExchange(result_buffer, Secret_Key_B, G);
      if (status == C25519_SUCCESS)
      {
          status = C25519keyExchange(result_buffer, Secret_Key_A,
Public_Key_B);
          if (status == C25519_SUCCESS)
          {
            /* add application traintment in case of SUCCESS */
          }
          else
          {
             /* add application traintment in case of FAILED */
          }
      }
      else
      {
          /* add application traintment in case of FAILED */
      }
  }
  else
  {
        //* add application traintment in case of FAILED */
  }
}
```

9 DES and Triple-DES algorithms

9.1 DES and Triple-DES description

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

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

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

The legacy STM32 cryptographic firmware 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 all the STM32 MCUs, using a software algorithm implementation.

For DES and Triple-DES library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For DES and Triple-DES library performance and memory requirements, refer to the legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\Fw_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.

9.2 DES library functions

Table 91 describes the DES functions of firmware cryptographic library.

Table 91. DES algorithm functions of the firmware library(1)

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

1. DDD = ECB or CBC.

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

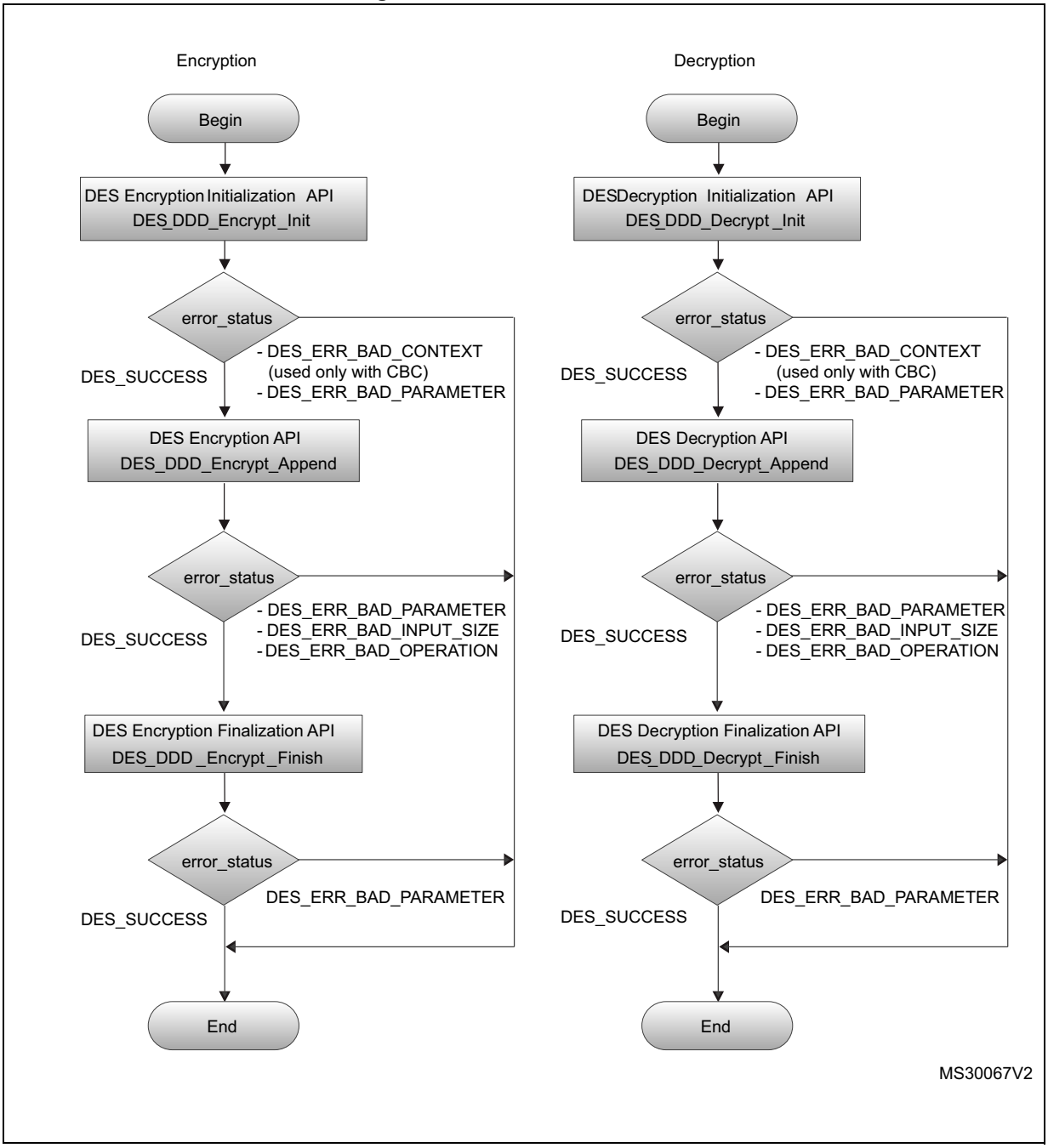
For example, to use ECB mode as a DES algorithm, user can use the following functions:

Table 92. 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

The flowcharts provided in *Figure 23* describe the DES algorithm.

Figure 23. DES DDD flowcharts



1. DDD = ECB or CBC.

9.2.1 DES_DDD_Encrypt_Init function

Table 93. DES_DDD_Encrypt_Init

Function name	DES_DDD_Encrypt_Init
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
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 notes below (This return value is only used with CBC algorithm)

Note: P_pDESDDDctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See Table 11: SKflags_et mFlags for details.

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

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.

DESDDDctx_stt data structure

Structure type for public key.

Table 94. 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 *Table 11: 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

9.2.2 DES_DDD_Encrypt_Append function

Table 95. DES_DDD_Encrypt_Append(1)

Function name	DES_DDD_Encrypt_Append

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 contains 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. – DES_ERR_BAD_OPERATION: Append not allowed.

1. DDD = ECB or CBC.

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

9.2.3 DES_DDD_Encrypt_Finish function

Table 96. DES_DDD_Encrypt_Finish(1)

Function name	DES_DDD_Encrypt_Finish
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 contains 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 = ECB or CBC.

Note: This function does not write output data and can hence be skipped. It is kept for API compatibility.

9.2.4 DES_DDD_Decrypt_Init function

Table 97. DES_DDD_Decrypt_Init(1)

Function name	DES_DDD_Decrypt_Init
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.
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 below (This return value is only used with CBC algorithm).

1. DDD = ECB or CBC.

Note: P_pDESDDDctx.mFlags must be set before calling this function. Default value is E_SK_DEFAULT. See Table 11: SKflags_et mFlags for details.

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

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.

9.2.5 DES_DDD_Decrypt_Append function

Table 98. DES_DDD_Decrypt_Append(1)

Function name	DES_DDD_Decrypt_Append
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 contains 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. – DES_ERR_BAD_OPERATION: Append not allowed.

1. DDD = ECB or CBC.

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

9.2.6 DES_DDD_Decrypt_Finish function

Table 99. DES_DDD_Decrypt_Finish(1)

Function name	DES_DDD_Decrypt_Finish
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 contains 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 = ECB or CBC.

Note: This function does not write output data and can ence be skipped.It is kept for API compatibility.

9.3 TDES library functions

Table 100 describes the TDES functions of firmware cryptographic library.

Table 100. TDES algorithm functions of the firmware library

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, it can be ECB or CBC.

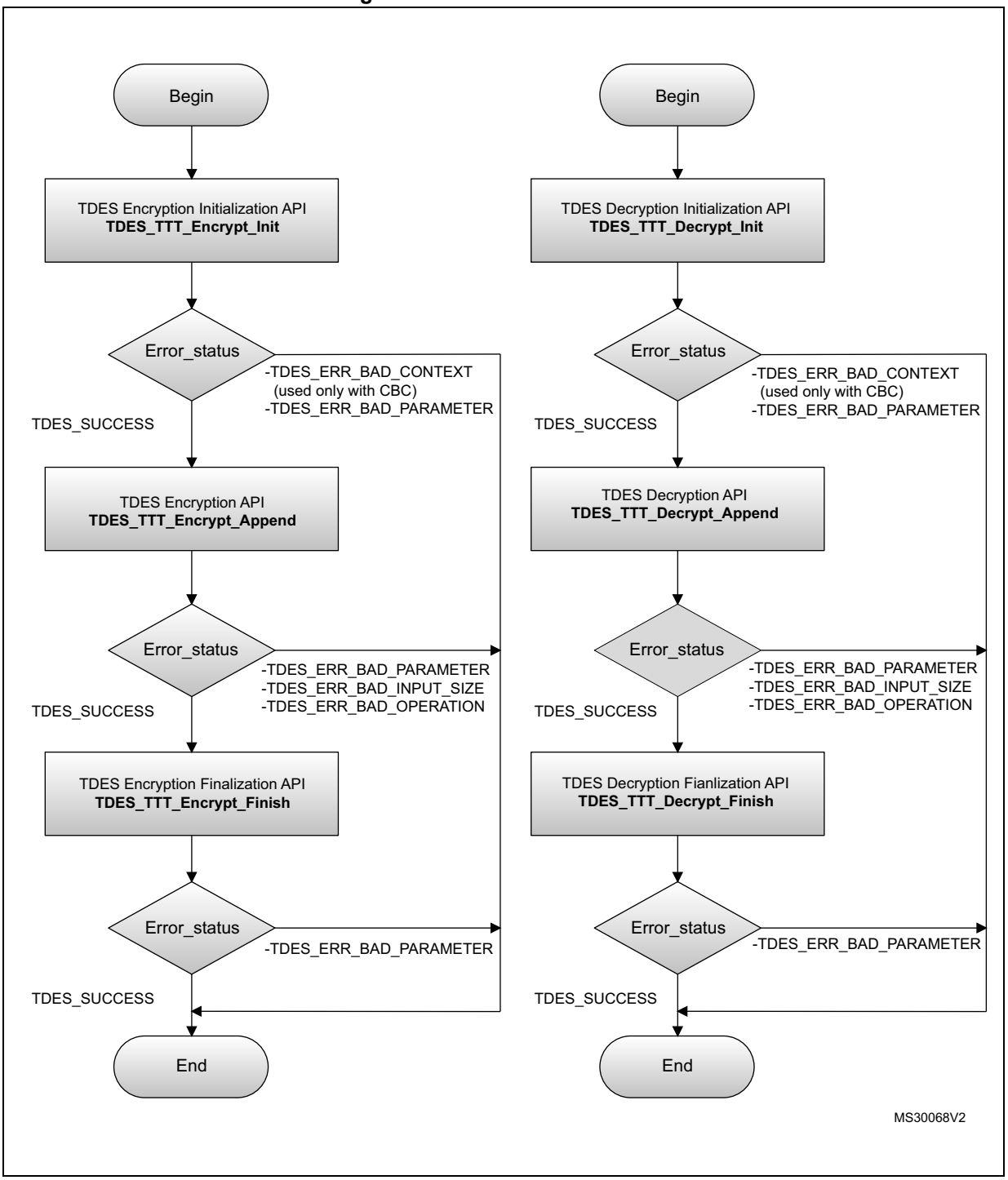
The flowcharts in *Figure 24* describe the TDES algorithm.

For example, to use ECB mode as a TDES algorithm, use the following functions:

Table 101. 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 24. TDES TTT flowcharts



1. TTT = ECB or CBC.

9.3.1 TDES_TTT_Encrypt_Init function

Table 102. TDES_TTT_Encrypt_Init(1)

Function name	TDES_TTT_Encrypt_Init
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.
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 below (This return value is only used with CBC algorithm).

1. TTT is ECB or CBC.

Note: P_pTDESTTTctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See Table 11: SKflags_et mFlags for details.

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

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.

TDESTTTctx_stt data structure

Structure type for public key.

Table 103. 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 *Table 11: 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

9.3.2 TDES_TTT_Encrypt_Append function

Table 104. TDES_TTT_Encrypt_Append(1)

Function name	TDES_TTT_Encrypt_Append

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 contains 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. – TDES_ERR_BAD_OPERATION: Append not allowed.

1. TTT is ECB or CBC.

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

9.3.3 TDES_TTT_Encrypt_Finish function

Table 105. TDES_TTT_Encrypt_Finish(1)

Function name	TDES_TTT_Encrypt_Finish
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 contains 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 does not write output data and can hence be skipped.It is kept for API compatibility.

9.3.4 TDES_TTT_Decrypt_Init function

Table 106. TDES_TTT_Decrypt_Init(1)

Function name	TDES_TTT_Decrypt_Init
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
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 below (This return value is only used with CBC algorithm)

1. TTT is ECB or CBC.

Note: P_pTDESTTTctx.mFlags must be set prior to calling this function. Default value is E_SK_DEFAULT. See Table 11: SKflags_et mFlags for details.

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

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.

9.3.5 TDES_TTT_Decrypt_Append function

Table 107. TDES_TTT_Decrypt_Append(1)

Function name	TDES_TTT_Decrypt_Append
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

Table 107. TDES_TTT_Decrypt_Append(1) (continued)

Function name	TDES_TTT_Decrypt_Append
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 contains 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 several times, provided that P_inputSize is a multiple of 8.

9.3.6 TDES_TTT_Decrypt_Finish function

Table 108. TDES_TTT_Decrypt_Finish(1)

Function name	TDES_TTT_Decrypt_Finish
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 contains 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 does not write output data and can hence be skipped. It is kept for API compatibility.

9.4 DES with ECB mode example

The following code give a simple example how to use DES-ECB of the legacy STM32 cryptographic firmware library.

```
#include "main.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;
    }
  }
}
```

10 ECC algorithm

10.1 ECC 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 all STM32 microcontrollers using a software algorithm implementation.

For ECC library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For the algorithms supported with hardware acceleration, refer to *Section 3.4.2: Legacy STM32 cryptographic hardware acceleration library*.

For ECC library performances and memory requirements, refer to:

- The legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\Fw_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.
- The legacy STM32 cryptographic hardware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\AccHw_Crypto\STM32XY\Documentation” where XY indicates the STM32.

10.2 ECC library functions

Table 109. ECC algorithm functions of firmware library

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

Table 109. ECC algorithm functions of firmware library (continued)

Function name	Description
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
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

Table 110 describes the ECC functions of the legacy STM32 cryptographic hardware acceleration library.

Table 110. ECC algorithm functions of hardware acceleration library

Function name	Description
AccHw_ECCinitEC	Initialize the elliptic curve parameters into a AccHw_EC_stt structure
AccHw_ECCfreeEC	De-initialize an AccHw_EC_stt context
AccHw_ECCinitPoint	Initialize an ECC point
AccHw_ECCfreePoint	Free Elliptic curve point
AccHw_ECCsetPointCoordinate	Set the value of one of coordinate of an ECC point
AccHw_ECCgetPointCoordinate	Get the value of one of coordinate of an ECC point
AccHw_ECCcopyPoint	Copy an elliptic curve point
AccHw_ECCinitPrivKey	Initialize an ECC private key
AccHw_ECCfreePrivKey	Free an ECC Private Key
AccHw_ECCsetPrivKeyValue	Set the value of an ECC private key object from a byte array
AccHw_ECCgetPrivKeyValue	Get the private key value from an ECC private key object
AccHw_ECCscalarMul	Computes the point scalar multiplication kP = k*P
AccHw_ECDSAinitSign	Initialize an ECDSA signature structure
AccHw_ECDSAfreeSign	Free an ECDSA signature structure
AccHw_ECDSAsetSignature	Set the value of the parameters (one at a time) of an AccHw_ECDSAsignature_stt
AccHw_ECDSAgetSignature	Get the values of the parameters (one at a time) of an AccHw_ECDSAsignature_stt

Table 110. ECC algorithm functions of hardware acceleration library (continued)

Function name	Description
AccHw_ECDSAverify	ECDSA signature verification with a digest input
AccHw_ECCvalidatePubKey	Checks the validity of a public key.
AccHw_ECCkeyGen	Generate an ECC key pair.
AccHw_ECDSAsign	ECDSA Signature Generation
AccHw_ECCgetPointFlag	Reads the flag member of an elliptic curve point structure
AccHw_ECCsetPointFlag	Set the flag member of an elliptic curve point structure
AccHw_ECCsetPointGenerator	Writes the elliptic curve generator point into a AccHw_ECpoint_stt

The flowcharts provided in *Figure 25*, *Figure 26* and *Figure 27* describe the ECC algorithms.

Figure 25. ECC sign flowchart

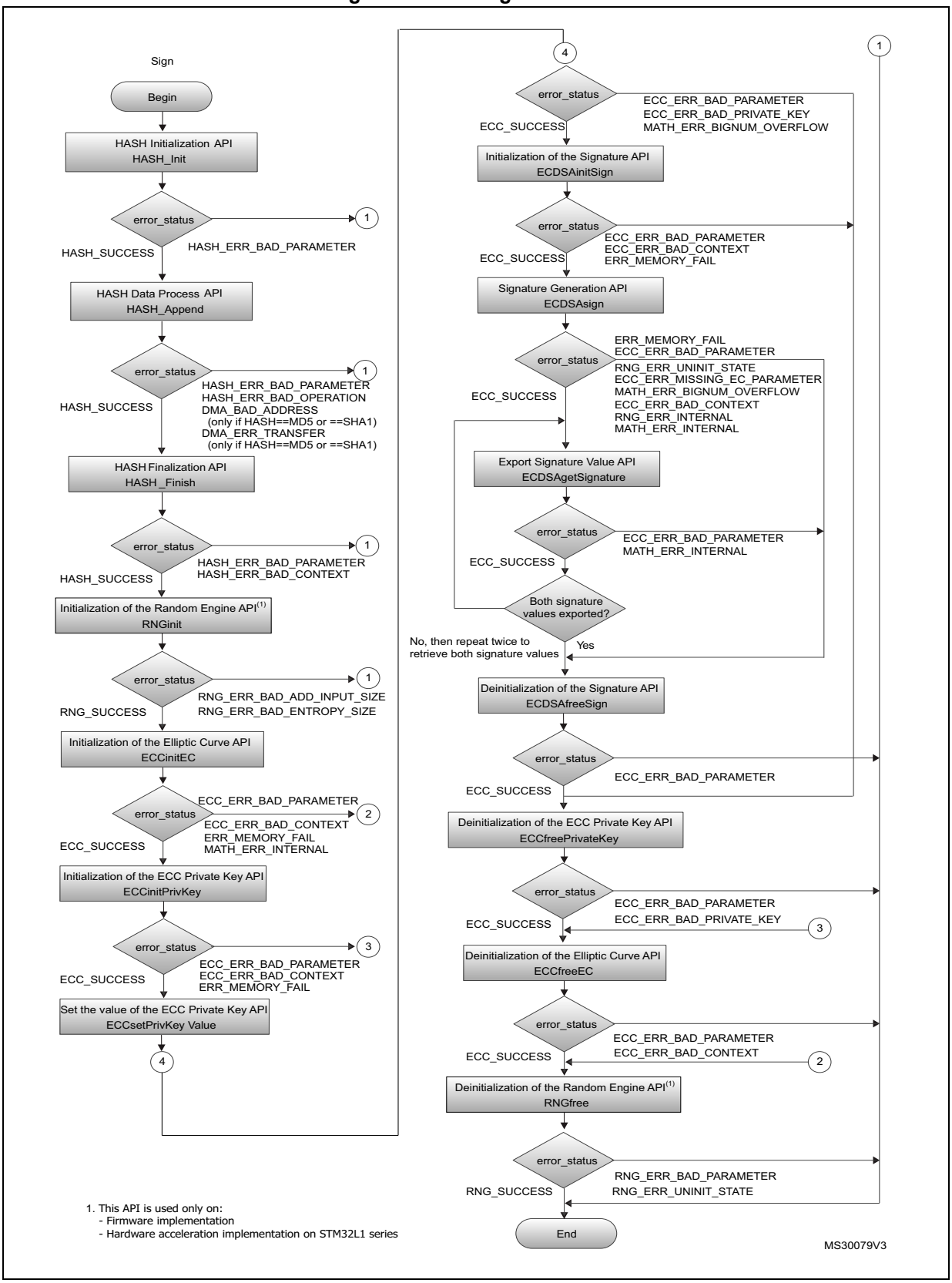


Figure 26. ECC verify flowchart

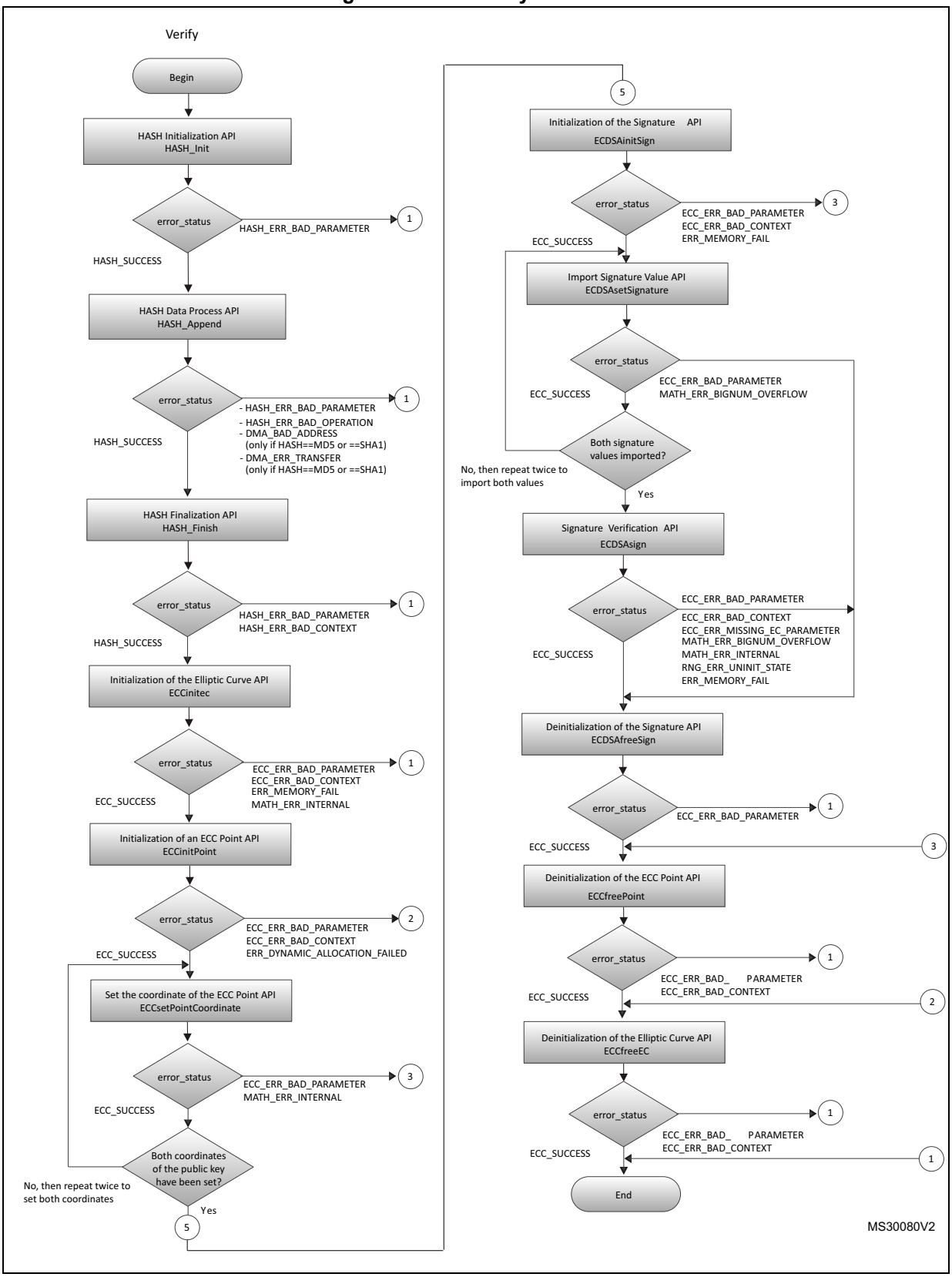
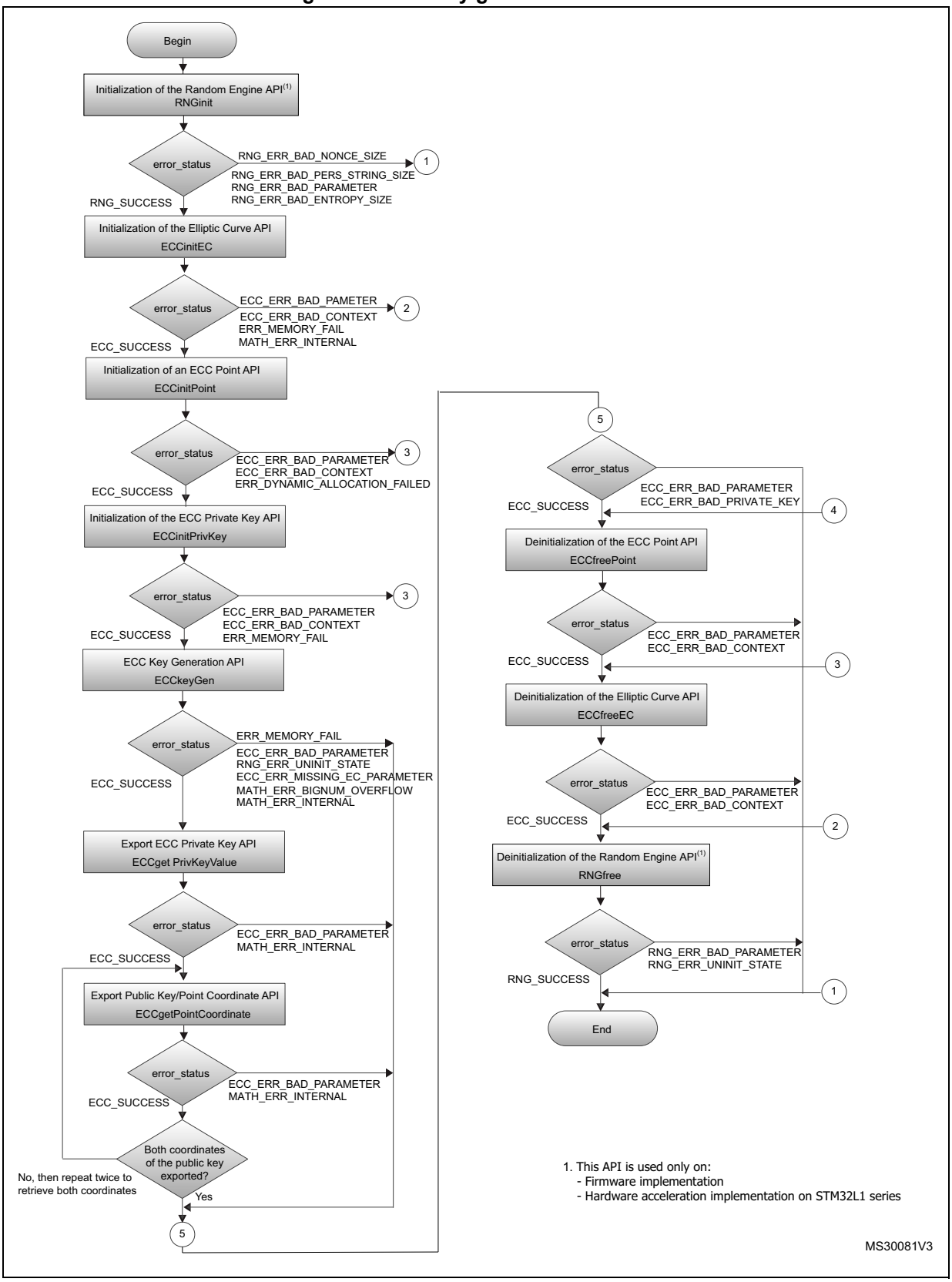


Figure 27. ECC key generator flowchart



10.2.1 ECCinitEC function

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

Table 111. 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 is 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. – MATH_ERR_INTERNAL Generic MATH error.

Note: This is the first EC operation to perform, it loads elliptic curve domain parameters. It is not needed to load every parameter, 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".*

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

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 112. 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"

Table 112. EC_stt data structure (continued)

Field name	Description
`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

10.2.2 ECCfreeEC function

Table 113. 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.

10.2.3 ECCinitPoint function

Table 114. 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 is initialized. – [in] *P_pECctx: The EC_stt containing the elliptic curve parameters. – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that is 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 initiated by ECCinitPoint, and freed by ECCfreePoint.

Table 115. 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.

10.2.4 ECCfreePoint function

Table 116. 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 is 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.

10.2.5 ECCsetPointCoordinate function

Table 117. 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

Table 117. ECCsetPointCoordinate function (continued)

Function name	ECCsetPointCoordinate
Parameter	– [in,out] *P_pECPnt: The ECC point that has 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. – MATH_ERR_INTERNAL Generic MATH error.

10.2.6 ECCgetPointCoordinate function

Table 118. 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 contains the returned coordinate – [out] *P_pCoordinateSize: Pointer to an integer that contains the size of the returned coordinate
Return value	– ECC_SUCCESS: Operation successful. – ECC_ERR_BAD_PARAMETER: One of the input parameters is invalid. – MATH_ERR_INTERNAL Generic MATH error.

Note: In version 2.1 (unlike previous versions) 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 is expanded to be a multiple of 4. In this case P_pCoordinateValue contains one or more leading zeros.

10.2.7 ECCgetPointFlag function

Table 119. 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

Table 119. ECCgetPointFlag function

Function name	ECCgetPointFlag
Parameter	– [in] *P_pECPnt The point whose flag isreturned
Return value	– ECC_ERR_BAD_PARAMETER (P_pECPnt == NULL). – CRL_EPOINT_GENERAL. – CRL_EPOINT_NORMALIZED. – CRL_EPOINT_INFINITY.

10.2.8 ECCsetPointFlag function

Table 120. 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 is set – [out] P_newFlag The flag value to be set
Return value	– None

10.2.9 ECCcopyPoint function

Table 121. 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 is 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. – ERR_MEMORY_FAIL There is not enough memory.

Note: Both points must be already initialized with ECCinitPoint.

10.2.10 ECCinitPrivKey function

Table 122. ECCinitPrivKey function

Function name	ECCinitPrivKey
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 is initialized – [in] *P_pECctx: EC_stt containing the elliptic curve parameters – [in,out] *P_pMemBuf: Pointer to membuf_stt structure that is 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.

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 123. ECCprivKey_stt data structure

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

10.2.11 ECCfreePrivKey function

Table 124. 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 is 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

10.2.12 ECCsetPrivKeyValue function

Table 125. 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. – MATH_ERR_BIGNUM_OVERFLOW P_privateKeySize is too big for the allowed private key size.

10.2.13 ECCgetPrivKeyValue function

Table 126. ECCgetPrivKeyValue function

Function name	ECCgetPrivKeyValue
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 contains the size in bytes of P_pPrivateKey
Return value	– ECC_SUCCESS: Operation successful. – ECC_ERR_BAD_PARAMETER: One of the input parameters is invalid. – MATH_ERR_INTERNAL Generic MATH error.

Note: 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 is expanded to be a multiple of 4. In this case P_pPrivateKey contains one or more leading zeros.

10.2.14 ECCscalarMul function

Table 127. 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 is 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 – [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: One of the inputs == NULL. – ECC_ERR_BAD_CONTEXT: P_pECctx->pmInternalEC == NULL. – ECC_WARN_POINT_AT_INFINITY: The returned point is the O point for the Elliptic Curve. – MATH_ERR_INTERNAL Generic MATH error. – ECC_ERR_MISSING_EC_PARAMETER Some required parameters are missing from the P_pECctx structure. – ECC_ERR_BAD_PRIVATE_KEY Private key (P_pECCprivKey) is not initialized or set. – ERR_MEMORY_FAIL There is not enough memory.

10.2.15 ECCsetPointGenerator function

Table 128. 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 is 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.

10.2.16 ECDSAinitSign function

Table 129. ECDSAinitSign function

Function name	ECDSAinitSign
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 is allocated and initialized – [in] *P_pECctx The EC_stt containing the Elliptic Curve Parameters – [in, out] *P_pMemBuf Pointer to the membuf_stt structure that is 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.

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

ECDSAsignature_stt data structure

Object used to store an ECDSA signature

Table 130. ECDSAsignature_stt data structure

Field name	Description
`BigNum_stt *pmR`	Pointer to parameter R
`BigNum_stt *pmS`	Pointer to parameter S

10.2.17 ECDSAfreeSign function

Table 131. ECDSAfreeSign function

Function name	ECDSAfreeSign
Prototype	`int32_t ECDSAfreeSign( ECDSAsignature_stt **P_ppSignature, membuf_stt *P_pMemBuf)`
Behavior	Free an ECDSA signature structure

Table 131. ECDSAfreeSign function

Function name	ECDSAfreeSign
Parameter	– [in,out] *P_pSignature: The ECDSA signature that is 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

10.2.18 ECDSAsetSignature function

Table 132. 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 is 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.

10.2.19 ECDSAgetSignature function

Table 133. ECDSAgetSignature function

Function name	ECDSAgetSignature
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 133. ECDSAgetSignature function (continued)

Function name	ECDSAgetSignature
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 contains 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. – MATH_ERR_INTERNAL Generic MATH error.

Note: 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.

10.2.20 ECDSAverify function

Table 134. 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 verifies the signature – [in] *P_pVerifyCtx: The ECDSA signature that is verified – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that is 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_INTERNAL Generic MATH error. – IGNATURE_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.*

ECDSAverifyctx_stt data structure

Structure used in ECDSA signature verification function.

Table 135. ECDSAverifyctx_stt data structure(1)

Field name	Description
`AccHw_ECpoint_stt *pmPubKey`	Pointer to the ECC Public Key used in the verification
`AccHw_EC_stt *pmEC`	Pointer to elliptic curve parameters

1. In case of using the hardware library this structure is "AccHw_ECDSAverifyctx_stt"

10.2.21 ECCvalidatePubKey function

Table 136. ECCvalidatePubKey function

Function name	ECCvalidatePubKey
Prototype	`int32_t ECCvalidatePubKey(const ECpoint_stt *P_pECCpubKey, const EC_stt *P_pECctx, membuf_stt *P_pMemBuf)`
Behavior	Checks the validity of a public key.
Parameter	– [in] *pECCpubKey: The public key to be checked – [in] *P_pECctx: Structure describing the curve parameters – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that is 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_pECctx is invalid – ERR_MEMORY_FAIL: Not enough memory. – ECC_ERR_MISSING_EC_PARAMETER P_pECctx must contain a, p, n, b

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

10.2.22 ECCkeyGen function

Table 137. 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_pECctx, membuf_stt *P_pMemBuf)`
Behavior	Generate an ECC key pair.

Table 137. ECCkeyGen function

Function name	ECCkeyGen
Parameters	– [out] *P_pPrivKey: Initialized object that contains the generated private key – [out] *P_pPubKey: Initialized point that contains the generated public key – [in] *P_pRandomState(1): The random engine current state – [in] *P_pECctx: Structure describing the curve parameters. This must contain the values of the generator – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that is 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_pECctx must contain a, p, n, Gx,Gy. – MATH_ERR_BIGNUM_OVERFLOW: P_pPubKey was not properly initialized. – MATH_ERR_INTERNAL Generic MATH error.

1. This input is used only on:
   - Firmware implementation
   - Hardware acceleration implementation on STM32L1 Series

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.

10.2.23 ECDSAsign function

Table 138. 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.

Table 138. ECDSAsign function

Function name	ECDSAsign
Parameter	– [in] *P_pDigest: The message digest that is signed – [in] P_digestSize: The size in bytes of the P_pDigest – [out] *P_pSignature: Pointer to an initialized signature structure that contains the result of the operation – [in] *P_pSignCtx: Pointer to an initialized ECDSAsignCtx_stt structure – [in,out] *P_pMemBuf Pointer to the membuf_stt structure that is 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. – RNG_ERR_INTERNAL Internal RNG error. – MATH_ERR_BIGNUM_OVERFLOW: P_pPubKey was not properly initialized. – MATH_ERR_INTERNAL Generic MATH error. – 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.*

ECDSAsignctx_stt data structure

Structure used in ECDSA signature generation function.

Table 139. ECDSAsignctx_stt data structure(1)

Field name	Description
`AccHw_ECCprivKey_stt *pmPrivKey`	Pointer to the ECC Public Key used in the verification
`AccHw_EC_stt *pmEC`	Pointer to elliptic curve parameters
`AccHw_RNGstate_stt *pmRNG(2)`	Pointer to an Initialized random engine status

1. In case of using the hardware library this structure is "AccHw_ECDSAverifyctx_stt"
2. This field exist only if using firmware library or hardware acceleration library of STM32L1 Series.

10.3 ECC example

The following code give a simple example how to use ECC of legacy STM32 cryptographic firmware library.

```
/* Initialize the ECC curve structure, with all the parameters */
EC_st.pmA = P_384_a;
EC_st.pmB = P_384_b;
EC_st.pmP = P_384_p;
EC_st.pmN = P_384_n;
EC_st.pmGx = P_384_Gx;
EC_st.pmGy = P_384_Gy;
EC_st.mAsize = sizeof(P_384_a);
EC_st.mBsize = sizeof(P_384_b);
EC_st.mNsize = sizeof(P_384_n);
EC_st.mPsize = sizeof(P_384_p);
EC_st.mGxsize = sizeof(P_384_Gx);
EC_st.mGysize = sizeof(P_384_Gy);
pub_x = pub_x_384;
pub_y = pub_y_384;
sign_r = sign_r_384;
sign_s = sign_s_384;
InputMessage =  InputMessage_384;
pub_x_size = sizeof(pub_x_384);
pub_y_size = sizeof(pub_y_384);
sign_r_size = sizeof(sign_r_384);
sign_s_size = sizeof(sign_s_384);
InputLength = sizeof(InputMessage_384);


    /* First hash the message to have a valid digest that will be signed
through RSA */
    status =
STM32_SHA256_HASH_DigestCompute((uint8_t*)InputMessage,InputLength,
 (uint8_t*)MessageDigest,&MessageDigestLength);
if (status == HASH_SUCCESS)
    {/* We prepare the memory buffer strucure */
      Crypto_Buffer.pmBuf =  preallocated_buffer;
      Crypto_Buffer.mUsed = 0;
      Crypto_Buffer.mSize = sizeof(preallocated_buffer);
/* Init the Elliptic Curve, passing the required memory */
/* Note: This is not a temporary buffer, the memory inside EC_stt_buf is
linked to EC_st As long as there's need to use EC_st the content of
EC_stt_buf should not be altered */
 status = ECCinitEC(&EC_st, &Crypto_Buffer );
      if (status == ECC_SUCCESS)
      {
```

```
  /* EC is initialized, now prepare to import the public key. First,
allocate a point */
     ECpoint_stt *PubKey = NULL;
        status = ECCinitPoint(&PubKey, &EC_st, &Crypto_Buffer);
      if (status == ECC_SUCCESS)
        {
          /* Point is initialized, now import the public key */
          ECCsetPointCoordinate(PubKey, E_ECC_POINT_COORDINATE_X, pub_x,
pub_x_size);
          ECCsetPointCoordinate(PubKey, E_ECC_POINT_COORDINATE_Y, pub_y,
pub_y_size);
/* Try to validate the Public Key. */
          status = ECCvalidatePubKey(PubKey, &EC_st, &Crypto_Buffer);
          if (status == ECC_SUCCESS)
          {/* Public Key is validated, Initialize the signature object */
            status = ECDSAinitSign(&sign, &EC_st, &Crypto_Buffer);
            if (status == ECC_SUCCESS)
            {ECDSAverifyCtx_stt verctx;
              /* Import the signature values */
              ECDSAsetSignature(sign, E_ECDSA_SIGNATURE_R_VALUE, sign_r,
sign_r_size);
              ECDSAsetSignature(sign, E_ECDSA_SIGNATURE_S_VALUE, sign_s,
sign_s_size);
             /* Prepare the structure for the ECDSA signature verification
*/
              verctx.pmEC = &EC_st;
              verctx.pmPubKey = PubKey;
              /* Verify it */
    status = ECDSAverify(MessageDigest, MessageDigestLength, sign,
&verctx,&Crypto_Buffer);
              if (status == SIGNATURE_VALID )
              {/* Signature has been validated, free our ECC objects */
                ECDSAfreeSign(&sign, &Crypto_Buffer);
                ECCfreePoint(&PubKey, &Crypto_Buffer);
                ECCfreeEC(&EC_st, &Crypto_Buffer);
              }
              else
              {/* Add application traitment in case of wrong signature
verification }
           }
            else
            {/* Add application traitment in case of bad signature
initialization}
          }
          else
          {/* Add application traitment in case of corrupted pub key or b
parameter}
```

```
        }
        else
        {/* Point allocation issues possible values of status}
      }
      else
      {/* Initialisation of elliptic curve issues}}
    else
    {/* Add application traintment in case of hash not success possible
values of status*/}
  }
}
```

11 ED25519 algorithm

11.1 ED25519 description

ED25519 is an elliptic curve digital signature algortithm, developed by Dan Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. It is a specific implementation of EdDSA using the Twisted Edwards curve and it can be used for key generation, document signing and signature verification.

This algorithm can run with all STM32 microcontrollers using a software algorithm implementation.

For ED25519 library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For ED25519 library performances and memory requirements, refer the legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\Fw_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.

11.2 ED25519 library functions

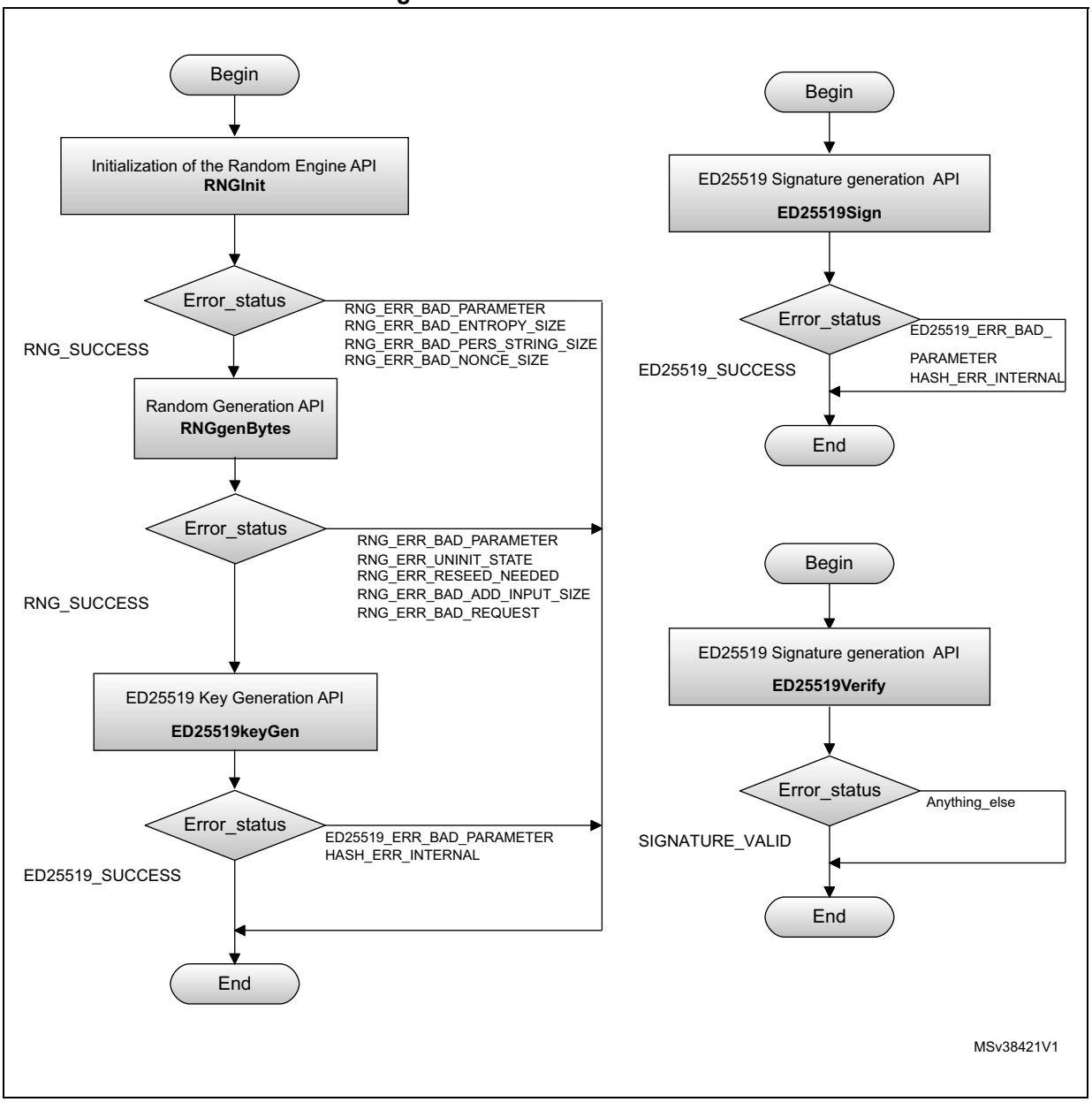
Table 140 describes the ED25519 functions of the firmware cryptographic library.

Table 140. ED25519 algorithm functions of the firmware library

Function name	Description
`ED25519keyGen`	Public key generation using the secret key
`ED25519sign`	Signature generation function
`ED25519verify`	Signature verification function

The flowcharts provided in *Figure 28* describe the ED25519 algorithms.

Figure 28. ED25519 flowcharts



11.2.1 ED25519keyGen

Table 141. ECDSAsign function

Function name	ECDSAsign
Prototype	int32_t ED25519keyGen (uint8_t *P_pPrivateKey, uint8_t *P_pPublicKey)
Behavior	Public key generation using the secret key
Parameter	– [in,out] *P_pPrivateKey Buffer of 64 bytes, holding random values in the first 32 bytes – [out] *P_pPublicKey Buffer of 32 bytes which holds the Public Key
Return value	– ED25519_SUCCESS Key Pair generated successfully. – ED25519_ERR_BAD_PARAMETER: P_pPrivateKey == NULL or P_pPublicKey == NULL. – HASH_ERR_INTERNAL Generic HASH internal error.

Note: The first 32 byte of P_pPrivateKey must contain random bytes that is used for the key pair generation.

11.2.2 ED25519genPublicKey

Table 142. ED25519genPublicKey

Function name	ED25519genPublicKey
Prototype	int32_t ED25519genPublicKey (uint8_t *P_pPublicKey, const uint8_t *P_pPrivateKey)
Behavior	Public key generation using the secret key
Parameter	– [out] *P_pPublicKey The public key that is generated – [in] *P_pPrivateKey Buffer of 64 bytes containing a valid private key
Return value	– ED25519_SUCCESS Key Pair generated successfully. – ED25519_ERR_BAD_PARAMETER: P_pPrivateKey == NULL or P_pPublicKey == NULL. – HASH_ERR_INTERNAL Generic HASH internal error.

Note: This functions is mainly used for testing purposes.

11.2.3 ED25519sign

Table 143. ED25519sign

Function name	ED25519sign
Prototype	int32_t ED25519sign (uint8_t *P_pSignature, const uint8_t *P_pInput, int32_t P_InputSize, const uint8_t *P_pPrivateKey)
Behavior	Signature generation function
Parameter	– [out] *P_pSignature Buffer of 64 bytes which holds the signature of P_pInput – [in] *P_pInput Message to be signed – [in] P_InputSize Size of the Message to be signed – [in] *P_pPrivateKey private key to be used to generate the signature
Return value	– ED25519_SUCCESS signature generated successfully. – ED25519_ERR_BAD_PARAMETER: P_pRandomState == NULL or P_pPrivateKey == NULL or P_pPublicKey == NULL. – HASH_ERR_INTERNAL Generic HASH internal error.

11.2.4 ED25519verify

Table 144. ED25519verify

Function name	ED25519verify
Prototype	nt32_t ED25519verify(const uint8_t *P_pInput, int32_t P_InputSize, const uint8_t *P_pSignature, const uint8_t *P_pPublicKey)
Behavior	Signature verification function
Parameter	– [in] *P_pInput message whose signature is to be verified – [in] P_InputSize Size of the message whose signature is to be verified – [in] *P_pSignature 64 bytes signature of P_pInput – [in] *P_pPublicKey Public key to be used to verify the signature
Return value	– SIGNATURE_VALID signature is VALID. – Anything_else signature is INVALID.

11.3 ED25519 example

The following code give a simple example how to use ED25519 of the legacy STM32 cryptographic firmware library.

```
#include "main.h"
uint8_t public_key [32];  /* Buffer for the public key */
uint8_t signature[64];
uint8_t Secret_key [64] =
  {0x4c, 0xcd, 0x08, 0x9b, 0x28, 0xff, 0x96, 0xda, 0x9d, 0xb6, 0xc3, 0x46,
0xec, 0x11, 0x4e, 0x0f,0x5b, 0x8a, 0x31, 0x9f, 0x35, 0xab, 0xa6, 0x24, 0xda,
0x8c, 0xf6, 0xed, 0x4f, 0xb8, 0xa6, 0xfb,0x3d, 0x40, 0x17, 0xc3, 0xe8, 0x43,
0x89, 0x5a, 0x92, 0xb7, 0x0a, 0xa7, 0x4d, 0x1b, 0x7e, 0xbc,0x9c, 0x98, 0x2c,
0xcf, 0x2e, 0xc4, 0x96, 0x8c, 0xc0, 0xcd, 0x55, 0xf1, 0x2a, 0xf4, 0x66,
0x0c};
const uint8_t ed25519_m[1] ={0x72};
{
  int32_t status = ED25519_SUCCESS;
  /* generate public key from the secret key */
  status = ED25519keyGen(Secret_key, public_key);
  if (status == ED25519_SUCCESS)
  {
      /* Call the signature */
      status = ED25519sign(signature, ed25519_m, 1, Secret_key);
      if (status == ED25519_SUCCESS)
      {
          /* Call the signature verification*/
          status = ED25519verify(ed25519_m, 1, signature, public_key);
          if (status == AUTHENTICATION_SUCCESSFUL)
          {
       /* add application traintment in case of ED25519 authentication is
successful */
          }
          else
          {
            /* add application traintment in case of FAILED */
          }
     }
     else
     {
        /* add application traintment in case of FAILED */
     }
  }
  else
  {
      /* add application traintment in case of FAILED */
  }
}
```

12 HASH algorithm

12.1 HASH 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 legacy STM32 cryptographic firmware 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
- SHA-384
- SHA-512

In addition, HKDF algorithm is also supported in this package to perform key derivation using the extraction-then-expansion approach described using HMAC in counter mode, and using the SHA hash functions.

Theses algorithms can run with all STM32 microcontrollers using a software algorithm implementation.

For HASH library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For HASH library performance and memory requirements, refer to the legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\Fw_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.

12.2 HASH library functions

Table 145 describes the HASH functions of the firmware cryptographic library.

Table 145. HASH algorithm functions of the firmware library

Function name	Description
HHH_Init	Initialization a Hash algorithm Context
HHH_Append	Process input data and the HASH algorithm context that is 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 is updated
HMAC_HHH_Finish	HMAC-HHH Finish function, produce the output HMAC-Hash algorithm tag
HMAC_HHH_Finish	HMAC-HHH Finish function, produce the output HMAC-Hash algorithm tag

Note: HHH represents the mode of operations of the Hash algorithm, it can be MD5, SHA1, SHA244, SHA256, SHA384 or SHA512

The flowcharts provided in *Figure 29* describe the HHH algorithm.

For example, to use SHA1 for HASH algorithm, call the functions:

Table 146. 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 is updated
`HMAC_SHA1_Finish`	HMAC-SHA1 Finish function, produce the output HMAC-SHA1 tag

Figure 29. Hash HHH flowcharts

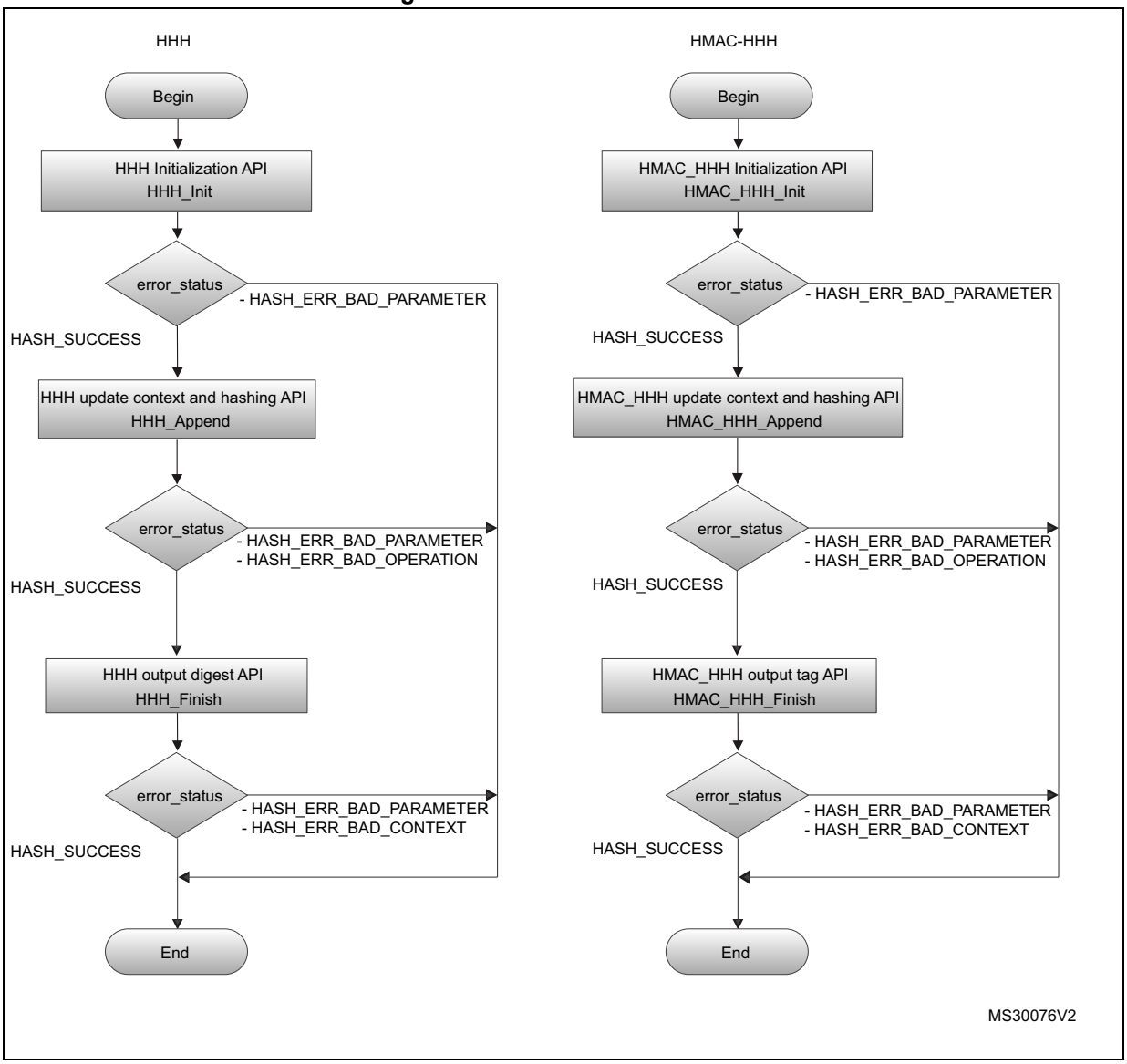
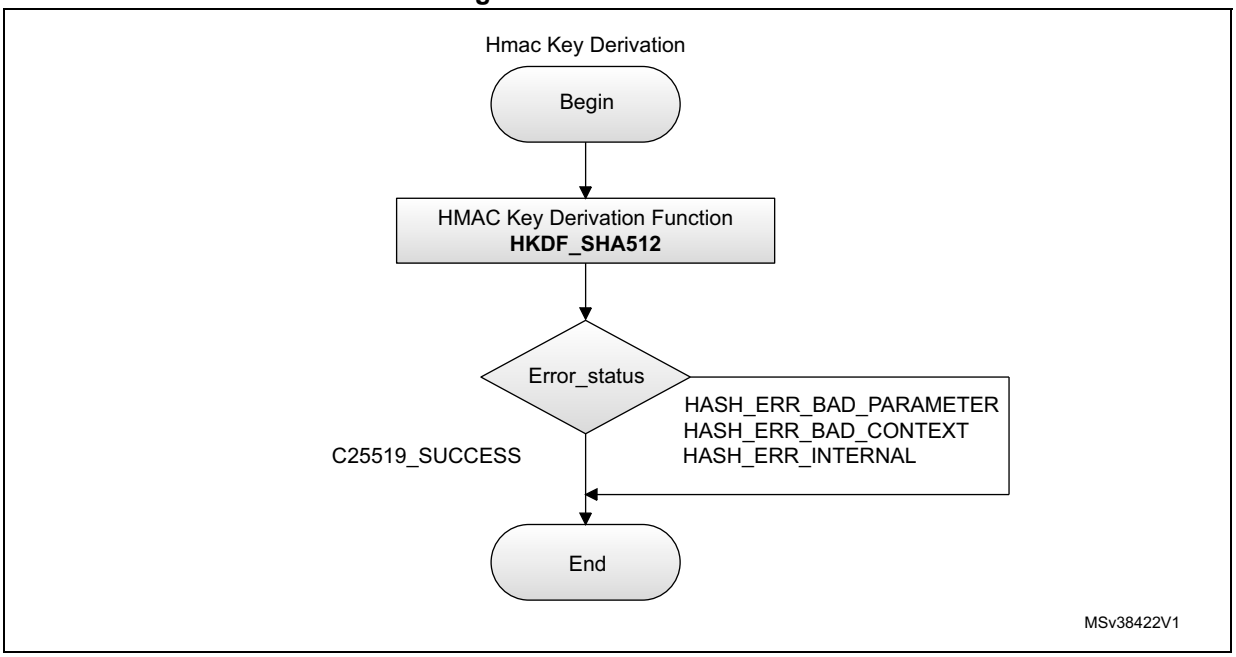


Figure 30. HKDF flowchart



12.2.1 HHH_Init function

Table 147. 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 is initialized
Return value	– HASH_SUCCESS: Operation successful. – HASH_ERR_BAD_PARAMETER: Parameter P_pHHHctx is invalid.

Note: HHH is MD5, SHA1, SHA224, SHA256, SHA384 or SHA512.

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

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

HASHctx_stt struct reference

Structure for HASH context.

Table 148. 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 *Table 149*

Table 148. HASHctx_stt struct reference (continued)

Field name	Description
`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 bit
`uint32_t amState[8]`	Internal: Keeps the internal state

HashFlags_et mFlags

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

Table 149. 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.

12.2.2 HHH_Append function

Table 150. 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 is updated – [in] *P_pInputBuffer:The data that is 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.

Note: HHH is MD5, SHA1, SHA224, SHA256, SHA384 or SHA512.

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

12.2.3 HHH_Finish function

Table 151. 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 contains 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: HHH is MD5, SHA1, SHA224, SHA256, SHA384 or SHA512.

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

12.2.4 HMAC_HHH_Init function

Table 152. 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 is initialized
Return value	– HASH_SUCCESS: Operation successful. – HASH_ERR_BAD_PARAMETER: Parameter P_pHMAC_HHHctx is invalid.

Note: HHH is MD5, SHA1, SHA224, SHA256 or SHA512.

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

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

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

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

HMACctx_stt struct reference

Structure for HMAC context.

Table 153. 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 *Table 149*
`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 *Table 148*

12.2.5 HMAC_HHH_Append function

Table 154. 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 HMAC_HHHctx_stt
Parameter	– [in,out] *P_pHMAC_HHHctx: The HMAC-HHH context that is updated – [in] *P_pInputBuffer:The data that is 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.

Note: HHH is MD5, SHA1, SHA224, SHA256, SHA384 or SHA512.

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

12.2.6 HMAC_HHH_Finish function

Table 155. 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 contains 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, SHA256, SHA384 or SHA512.

P_pHHHctx->mTagSize must contain a valid value, between 1 and CRL_HHH_SIZE.

12.2.7 HKDF_SHA512

The HKDF algorithm API is:

Table 156. HKDF_SHA512

Function name	HKDF_SHA512
Prototype	nt32_t HKDF_SHA512(const HKDFinput_stt *P_pInputSt, uint8_t *P_pOutputBuffer, int32_t P_OutputSize)
Behavior	HMAC Key Derivation function
Parameter	– [in] *P_pInputSt An already initialized HKDFinput_stt structure containing the inputs to HKDF – [out] *P_pOutputBuffer The output buffer, its size should be at least P_OutputSize – [in] P_OutputSize The number of output bytes required
Return value	– HASH_SUCCESS Operation successful. – HASH_ERR_BAD_PARAMETER One of the input pointer is NULL or P_OutputSize <= 0 or P_OutputSize > 255*64. – HASH_ERR_BAD_CONTEXT One of the pointers inside P_pInputSt is NULL while its size is greater than zero. – HASH_ERR_INTERNAL Internal Hash error.

The caller must correctly initialize at least the values of pmKey and mKeySize inside P_pInputSt. Also need to provide a valid pmInfo and mInfoSize to get a valid HASH results.

Table 157. HMACctx_stt struct reference

Field name	Description
`const uint8_t *pmKey`	Pointer for the HKDF Key
`int32_t mKeySize`	Size of the HKDF Key
`const uint8_t *pmSalt`	Pointer for the HKDF Salt
`int32_t mSaltSize`	Size of the HKDF Salt
`const uint8_t *pmInfo`	Pointer for the HKDF Info
`int32_t mInfoSize;`	Size of the HKDF Info
`int32_t mInfoSize;`	Size of the HKDF Info

12.3 HASH SHA1 example

The following code give a simple example how to use SHA-1 iof the legacy STM32 cryptographic firmware library.

```
#include "main.h"
 int32_t main()
 {
   uint8_t input[]={0x40,0x41};
   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)
      {
        /* add application traintment in case of SUCCESS*/
      }
         else
      {
        /* add application traintment in case of FAILED */
      }
      }
      else
      {
     /* add application traintment in case of FAILED */
      }
   }
   else
   {
   /* add application traintment in case of FAILED */
   }
 }
```

13 POLY1305 algorithm

13.1 POLY1305 description

Poly1305-AES is a cryptographic message authentication code (MAC) written by Daniel J. Bernstein. Poly1305 computes between 1 and 16 bytes of tag of a variable-length message, using a 32 bytes of key, and 16 byte of nonce.

These modes can run with all STM32 microcontrollers, using a software algorithm implementation.

For POLY1305 library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For the devices which support this algorithms with hardware, refer to *Section 3.4.2: Legacy STM32 cryptographic hardware acceleration library*

For POLY1305 library performances and memory requirements, refer to:

- The legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\Fw_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.
- The legacy STM32 cryptographic hardware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\AccHw_Crypto\STM32XY\Documentation” where XY indicates the STM32.

13.2 POLY1305 library functions

Table 158. POLY1305 algorithm functions in firmware implementation

Function name	Description
`Poly1305_Auth_Init`	Initialization for Poly1305 Authentication
`Poly1305_Auth_Append`	Poly1305 Authentication
`Poly1305_Auth _Finish`	Poly1305 Authentication finalization
`Poly1305_Verify _Init`	Initialization for Poly1305 verification
`Poly1305_Verify _Append`	Poly1305 Verification
`Poly1305_Verify _Finish`	Poly1305 Verification finalization

Table 159. POLY1305 algorithm functions of hardware acceleration library

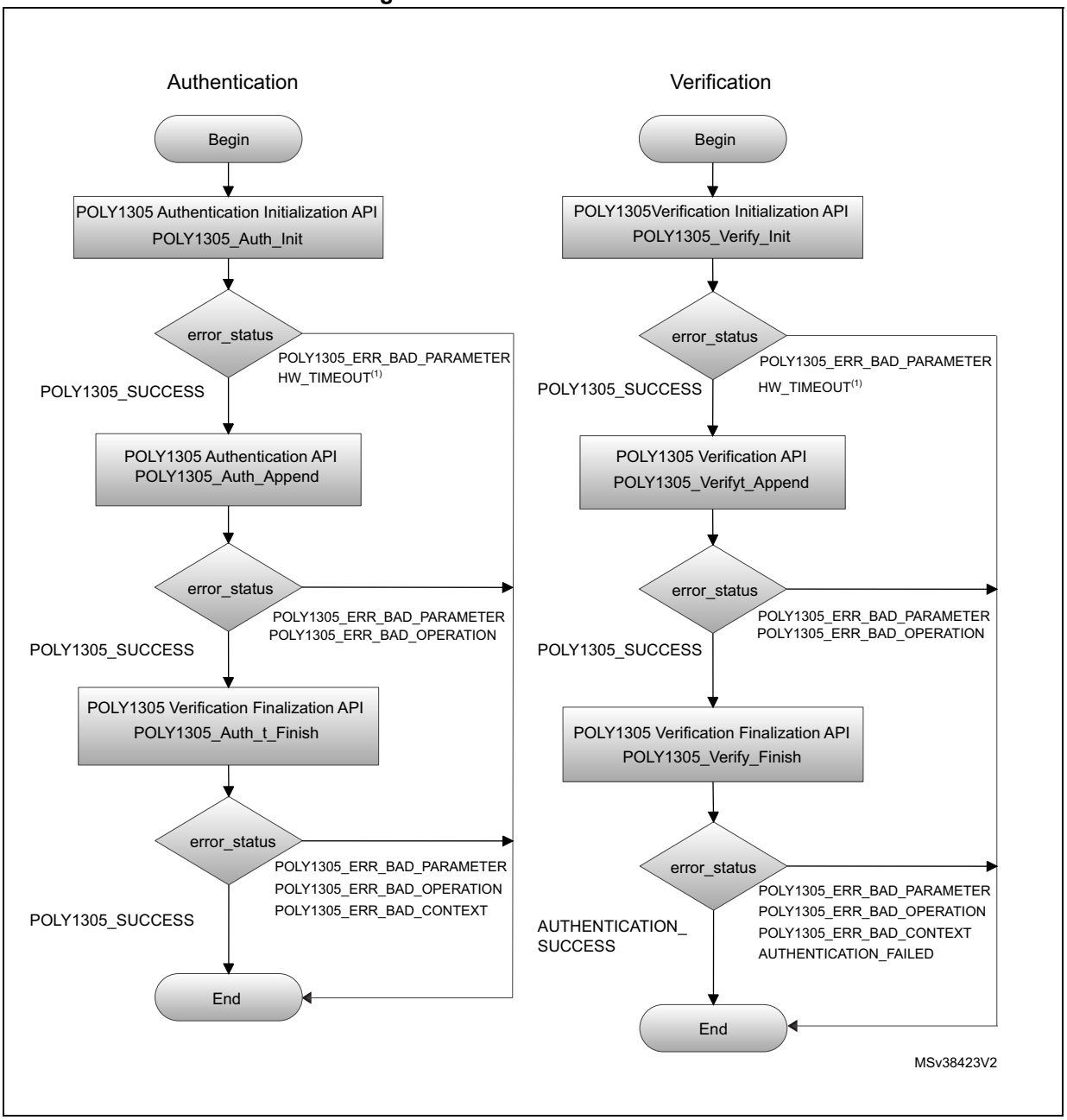
Function name	Description
`AccHw_Poly1305_Auth_Init`	Initialization for Poly1305 Authentication
`AccHw_Poly1305_Auth_Append`	Poly1305 Authentication
`AccHw_Poly1305_Auth _Finish`	Poly1305 Authentication finalization
`AccHw_Poly1305_Verify _Init`	Initialization for Poly1305 verification

Table 159. POLY1305 algorithm functions of hardware acceleration library

Function name	Description
AccHw_Poly1305_Verify _Append	Poly1305 Verification
AccHw_Poly1305_Verify _Finish	Poly1305 Verification finalization

The flowcharts provided in *Figure 31* describe the POLY1305 algorithms.

Figure 31. POLY1305 flowcharts



1. Used only with the algorithms featuring hardware acceleration.

13.2.1 Poly1305_Auth_Init

Table 160. Poly1305_Auth_Init

Function name	Poly1305_Auth_Init
Prototype	`int32_t Poly1305_Auth_Init (Poly1305ctx_stt *P_pPoly1305ctx, const uint8_t *P_pKey, const uint8_t *P_pNonce)`
Behavior	Initialization for Poly1305 Authentication
Parameter	– [in,out] *P_pPoly1305ctx Poly1305-AES context – [in] *P_pKey Poly1305-AES 32 byte key. See the note below – [in] *P_pNonce Poly1305-AES 16 byte Nonce (Number used Once) or NULL if no AES is required.
Return value	– POLY1305_SUCCESS Operation successful. – POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer. – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: The algorithms works with or without AES integrated. There are two possibilities:

- *If P_pNonce != NULL then Poly1305-AES is used: the first 16 bytes of P_pKey are the AES key used to encrypt the Nonce, the second 16 bytes compose the Poly1305 Key.*
- *If P_pNonce == NULL then only Poly1305 is used: the first 16 bytes of P_pKey compose the Poly1305 Key, while the second 16 bytes are directly added to the result of Poly1305. This is to support Poly1305 without AES.*

Poly1305ctx_stt data structure

Table 161. Poly1305ctx_stt(1) struct reference

Field name	Description
`uint32_t mContextId;`	Unique ID of this context. Not used in current implementation
`PolyFlags_et mFlags;`(1)	32 bit mFlags, used to perform keyschedule
`const uint8_t *pmKey;`	Pointer to original 32 bytes Key buffer
`const uint8_t *pmNonce;`	Pointer to original 16 bytes Nonce buffer
`const uint8_t *pmTag;`	Pointer to Authentication TAG. This value must be set in decryption, and this TAG is verified
`int32_t mTagSize;`	Size of the required Authentication TAG
`uint32_t r[5];`	Internal: value of r
`uint32_t h[5];`	Internal: value of h
`uint32_t pad[4];`	Internal: value of encrypted nonce

1. In case of using the hardware library this structure is "AccHw_Poly1305ctx_stt"

13.2.2 Poly1305_Auth_Append

Table 162. Poly1305_Auth_Append

Function name	Poly1305_Auth_Append
Prototype	`iint32_t Poly1305_Auth_Append (Poly1305ctx_stt *P_pPoly1305ctx, const uint8_t *P_pInputBuffer, int32_t P_inputSize)`
Behavior	Poly1305 Authentication
Parameter	– [in, out] *P_pPoly1305ctx Poly1305-AES context (Initialized with Poly1305_Auth_Init) – [in] *P_pInputBuffer Data to be authenticated – [in] P_inputSize Size (in bytes) of data at P_pInputBuffer
Return value	– POLY1305_SUCCESS Operation successful – POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer – POLY1305_ERR_BAD_OPERATION Call to Poly1305_Auth_Append is not allowed

Note: This function can be called several times, provided that P_inputSize is a multiple of 16. A single, last, call can be made with any value for P_inputSize.

13.2.3 Poly1305_Auth_Finish

Table 163. Poly1305_Auth_Finish

Function name	Poly1305_Auth_Finish
Prototype	`int32_t Poly1305_Auth_Finish (Poly1305ctx_stt *P_pPoly1305ctx, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)`
Behavior	Poly1305 Authentication Finalization
Parameter	– [in, out] *P_pPoly1305ctx Poly1305-AES context – [in] *P_pOutputBuffer where the TAG is writted – [in] *P_pOutputSize contains the size of the written tag
Return value	– POLY1305_SUCCESS Operation successful. – POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer. – POLY1305_ERR_BAD_OPERATION Call to Poly1305_Auth_Finish is not allowed. – POLY1305_ERR_BAD_CONTEXT P_pPoly1305ctx->mTagSize not properly set.

Note: This function requires to have properly set the value of P_pPoly1305ctx->mTagSize (between 1 and 16).

13.2.4 Poly1305_Verify_Init

Table 164. Poly1305_Verify_Init

Function name	Poly1305_Verify_Init
Prototype	`int32_t Poly1305_Verify_Init (Poly1305ctx_stt *P_pPoly1305ctx, const uint8_t *P_pKey, const uint8_t *P_pNonce)`
Behavior	Initialization for Poly1305 Verification
Parameter	– [in,out] *P_pPoly1305ctx Poly1305-AES contex – [in] *P_pKey Poly1305-AES 32 byte key. See note below – [in] *P_pNonce Poly1305-AES 16 byte Nonce (Number used Once). Or NULL if AES is not required.
Return value	– POLY1305_SUCCESS Operation successful. – POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer. – HW_TIMEOUT (used only in the algorithms with hardware acceleration) Time out has expired before setting up the hardware flag: a) "OFNE" for STM32F2, STM32F4 and STM32F7 Series b) "CCF" For STM32L0, STM32L1, and STM32L4 Series

Note: The algorithms works with or without AES integrated. There are two possible cases:

- *If P_pNonce != NULL then Poly1305-AES is used: the first 16 bytes of P_pKey are the AES key used to encrypt the Nonce, the second 16 bytes compose the Poly1305 Key.*
- *If P_pNonce == NULL then only Poly1305 is used: the first 16 bytes of P_pKey compose the Poly1305 Key, while the second 16 bytes are directly added to* the result of Poly1305. This is to support Poly1305 without AES.

P_pPoly1305ctx->pmTag must point to the TAG to be verified, whose size should be written in P_pPoly1305ctx.mTagSize.

This function is just a wrapper for Poly1305_Auth_Init.

13.2.5 Poly1305_Verify_Append

Table 165. Poly1305_Verify_Append

Function name	Poly1305_Verify_Append
Prototype	`int32_t Poly1305_Verify_Append (Poly1305ctx_stt *P_pPoly1305ctx, const uint8_t *P_pInputBuffer, int32_t P_inputSize)`
Behavior	Poly1305 Verification

Table 165. Poly1305_Verify_Append

Function name	Poly1305_Verify_Append
Parameter	– [in,out] *P_pPoly1305ctx Poly1305-AES context (initialized with Poly1305_Verify_Init) – [in] *P_pInputBuffer Data to be checked – [in] P_inputSize Size (in bytes) of data at P_pInputBuffer
Return value	– POLY1305_SUCCESS Operation successful. – POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer. – POLY1305_ERR_BAD_OPERATION Call to Poly1305_Verify_Append is not allowed.

Note: This function can be called several times, provided that P_inputSize is a multiple of 16.A single, last, call can be made with any value for P_inputSize.

This function is just a wrapper for Poly1305_Auth_Append.

13.2.6 Poly1305_Verify_Finish

Table 166. Poly1305_Verify_Finish

Function name	Poly1305_Verify_Finish
Prototype	`int32_t Poly1305_Verify_Finish (Poly1305ctx_stt *P_pPoly1305ctx, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)`
Behavior	Poly1305 Verification Finalization
Parameter	– [in,out] *P_pPoly1305ctx Poly1305-AES context – [in] *P_pOutputBuffer Not used, could be NULL – [in] *P_pOutputSize Not used, could be NULL
Return value	– POLY1305_ERR_BAD_PARAMETER P_pPoly1305ctx is a NULL pointer. – POLY1305_ERR_BAD_OPERATION Call to Poly1305_Verify_Finish is not allowed. – POLY1305_ERR_BAD_CONTEXT P_pPoly1305ctx->mTagSize not properly set or P_pPoly1305ctx->pmTag == NULL. – AUTHENTICATION_SUCCESSFUL TAG correctly verified. – AUTHENTICATION_FAILED Authentication Failed.

Note: This function requires to have properly set the value of P_pPoly1305ctx->mTagSize (between 1 and 16) and P_pPoly1305ctx->pmTag must pont to the TAG to be checked.

The caller should just check the return value to be equal (or different) to AUTHENTICATION_SUCCESSFUL and accept (reject) the message/tag pair accordingly.

13.3 POLY1305 example

The following code give a simple example how to use POLY1305 of the legacy STM32 cryptographic firmware library.

```
#include "main.h"
const uint8_t Key[] ={0xec, 0x07, 0x4c, 0x83, 0x55, 0x80, 0x74, 0x17};
const uint8_t Nonce[] ={0xfb, 0x44, 0x73, 0x50, 0xc4, 0xe8, 0x68, 0xc5};
const uint8_t Input[] ={0xf3, 0xf6};
/* Buffer to store the output data */
uint8_t OutputMessage[16];
/* Size of the output data */
  int32_t outputLength = 0;

const uint8_t Expected_Result[] =
  { 0xf4, 0xc6, 0x33, 0xc3, 0x04, 0x4f, 0xc1, 0x45,
    0xf8, 0x4f, 0x33, 0x5c, 0xb8, 0x19, 0x53, 0xde };

int main(void)
{
  Poly1305ctx_stt polyctx;
  uint32_t error_status = POLY1305_SUCCESS;
  /* Initialize Context by setting the size of the required TAG */
  polyctx.mTagSize = 16;
  /* Initialize operation */
  error_status = Poly1305_Auth_Init(&polyctx, Key, Nonce);

  if (error_status == POLY1305_SUCCESS)
  {
    error_status = Poly1305_Auth_Append(&polyctx,
                                        Input,
                                        sizeof(InputSize);

    if (error_status == POLY1305_SUCCESS)
    {
      error_status = Poly1305_Auth_Finish(&polyctx, OutputMessage ,
&outputLength);
    }
  }

  return error_status;
}
```

14 RNG algorithm

14.1 RNG 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.

These modes can run with all STM32 microcontrollers using a software algorithm implementation.

For RNG library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For the devices which support this algorithms with hardware acceleration, refer to *Section 3.4.2: Legacy STM32 cryptographic hardware acceleration library*.

For RNG library performances and memory requirements, refer to:

- The legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\Fw_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.
- The legacy STM32 cryptographic hardware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\AccHw_Crypto\STM32XY\Documentation” where XY indicates the STM32.

14.2 RNG library functions

Table 167 describes the RNG functions of the firmware cryptographic library.

Table 167. RNG algorithm functions of the firmware library

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

Table 168 describes the RNG functions of the hardware cryptographic library.

Table 168. RNG algorithm functions of the hardware acceleration library

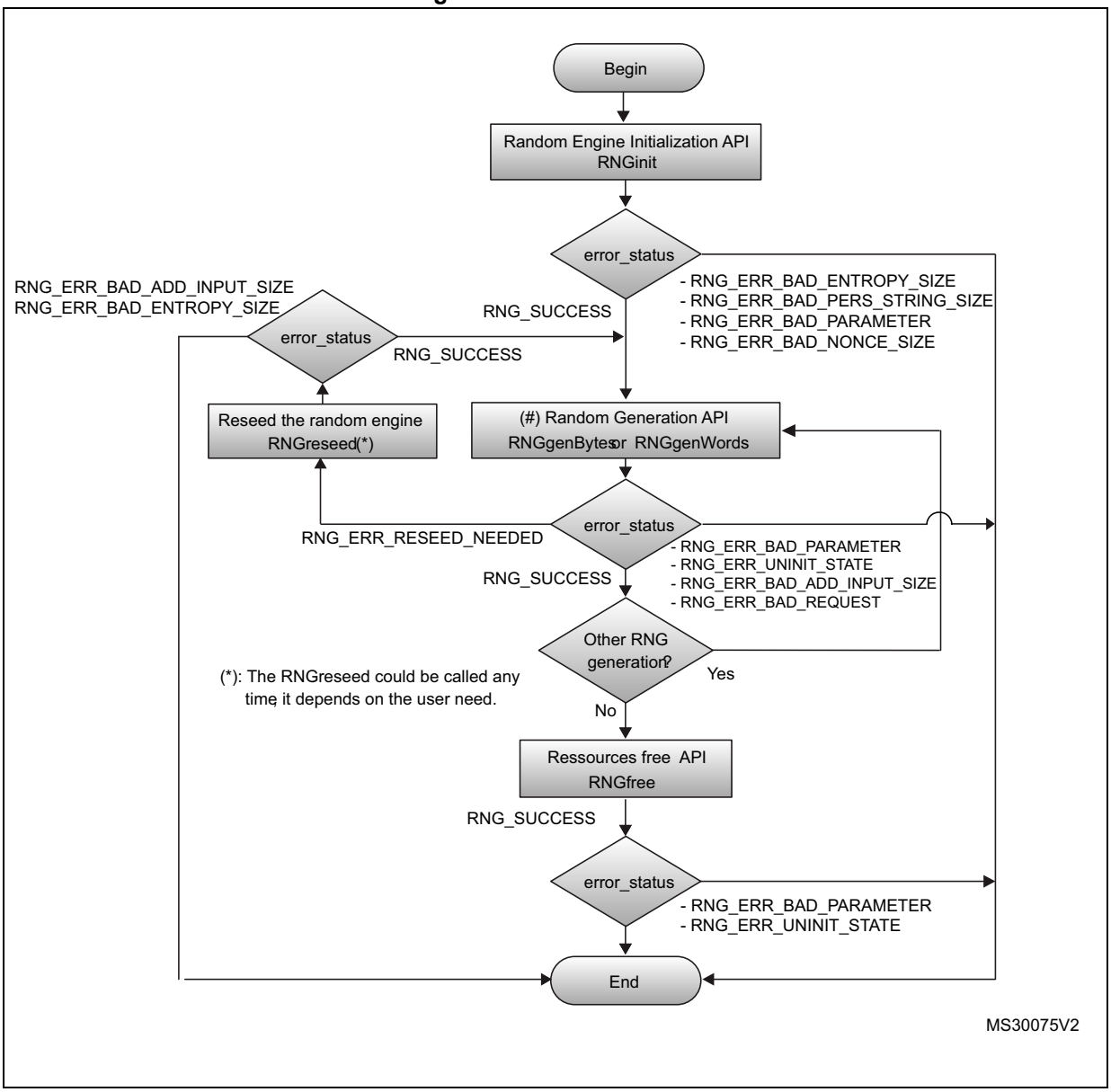
Function name	Description
`AccHw_RNGreseed`	Reseed the random engine
`AccHw_RNGinit`	Initialize the random engine
`AccHw_RNGfree`	Free a random engine state structure

Table 168. RNG algorithm functions of the hardware acceleration library

Function name	Description
`AccHw_RNGgenBytes`	Generation of pseudorandom octets to a buffer
`AccHw_RNGgenWords`	Generation of a random uint32_t array
`AccHw_RNGreseed`	Resend the random engine

The flowchart provided in *Figure 32* describe the RNG algorithm.

Figure 32. RNG flowchart



14.2.1 RNGreseed function

Table 169. 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 is 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 170. RNGreInput_stt(1) 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

1. In case of using the hardware library this structure is "AccHw_RNGreInput_stt"

RNGstate_stt struct reference

Structure that contains the by RNG state

Table 171. RNGstate_stt(1) struct reference

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

1. In case of using the hardware library this structure is "AccHw_RNGstate_stt"

14.2.2 RNGinit function

Table 172. RNGinit

Function name	RNGinit

Prototype	`int32_t RNGinit (`
	`const RNGinitInput_stt * P_pInputData,`
	`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 is initialized
Return value	– RNG_SUCCESS: Operation successful. – RNG_ERR_BAD_PARAMETER: Some of the inputs were NULL. – 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. – RNG_ERR_BAD_NONCE_SIZE: Wrong size for P_nonceSize. It must be less than CRL_DRBG_AES_MAX_NONCE_LEN.

RNGInitInput_stt struct reference

Structure that contains the by RNG state

Table 173. RNGInitInput_stt(1) struct reference

Field name	Description
`uint8_t * pmEntropyData`	Entropy data input
`int32_t mEntropyDataSize`	Size of the entropy data input (it should be greater than CRL_DRBG_AES128_ENTROPY_MIN_LEN and less than CRL_DRBG_AES_ENTROPY_MAX_LEN)
`uint8_t * pmNonce`	Nonce data (It can be NULL)
`uint32_t mNonceSize`	Size of the Nonce (It can be zero)
`int8_t* pmPersData`	Personalization String (It can be NULL)
`uint32_t mPersDataSize`	Size of personalization string (It can be zero)

1. In case of using the hardware library this structure is "AccHw_RNGInitInput_stt "

14.2.3 RNGfree function

Table 174. 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 is removed
Return value	– RNG_SUCCESS: Operation successful. – RNG_ERR_BAD_PARAMETER: P_pRandomState == NULL. – RNG_ERR_UNINIT_STATE: Random engine not initialized.

14.2.4 RNGgenBytes function

Table 175. 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 pseudo random 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. – RNG_ERR_BAD_ADD_INPUT_SIZE P_addInputSize > CRL_DRBG_AES_MAX_ADD_INPUT_LEN. – RNG_ERR_BAD_REQUEST P_nBytes > CRL_DRBG_AES_MAX_BYTES_PER_REQUEST (which is the maximum allowed value).

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

14.2.5 RNGgenWords function

Table 176. 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 is 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. – RNG_ERR_BAD_ADD_INPUT_SIZE P_addInputSize > CRL_DRBG_AES_MAX_ADD_INPUT_LEN. – RNG_ERR_BAD_REQUEST P_nBytes > CRL_DRBG_AES_MAX_BYTES_PER_REQUEST (which is the maximum allowed value).

14.3 RNG example

A simple random generation with C_SW_DRBG_AES128 is shown below:

```
#include "main.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);
 }
```

15 RSA algorithm

15.1 RSA description

This section describes RSA functions for signature generation/validation and encryption/decryption.

There are two structures that pass keys to the functions:

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

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
- RSA_PKCS1v15_Encrypt
- RSA_PKCS1v15_Decrypt

This algorithm can run with all STM32 microcontrollers using a software algorithm implementation.

For RSA library settings, refer to *Section 16: Legacy STM32 cryptographic library settings*.

For the modes supported with hardware acceleration, refer to *Section 3.4.2: Legacy STM32 cryptographic hardware acceleration library*.

For RSA library performances and memory requirements, refer to:

- The legacy STM32 cryptographic firmware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\Fw_Crypto\STM32XY\Documentation” where XY indicates the STM32 Series.
- The legacy STM32 cryptographic hardware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\AccHw_Crypto\STM32XY\Documentation” where XY indicates the STM32.

Caution: Due to PKCS # 1V1.5 specifications, there is a possibility of attack. For more details refer to PKCS # 1V1.5 specifications.

15.2 RSA library functions

Table 177. RSA algorithm functions of the legacy STM32 cryptographic firmware library

Function name	Description
`RSA_PKCS1v15_Sign`	PKCS#1v1.5 RSA signature generation function
`RSA_PKCS1v15_Verify`	PKCS#1v1.5 RSA signature verification function
`RSA_PKCS1v15_Encrypt`	PKCS#1v1.5 RSA encryption function
`RSA_PKCS1v15_Decrypt`	PKCS#1v1.5 RSA decryption function

Table 178. RSA algorithm functions of the legacy STM32 cryptographic hardware library

Function name	Description
`AccHw_RSA_PKCS1v15_Encrypt`	PKCS#1v1.5 RSA encryption function
`AccHw_RSA_PKCS1v15_Decrypt`	PKCS#1v1.5 RSA decryption function

The flowcharts provided in *Figure 33* and *Figure 34* describe the RSA algorithm.

Figure 33. RSA flowcharts

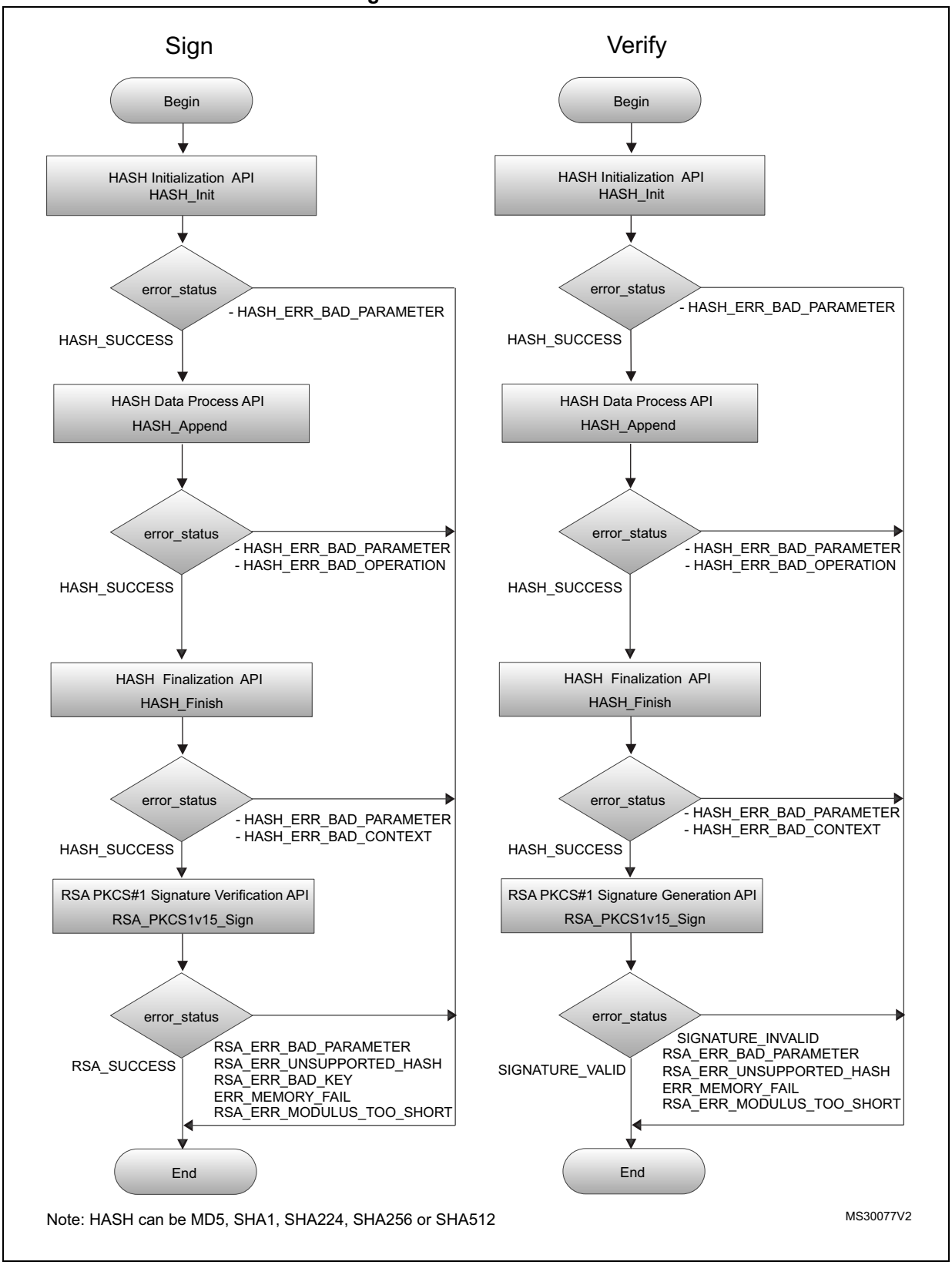
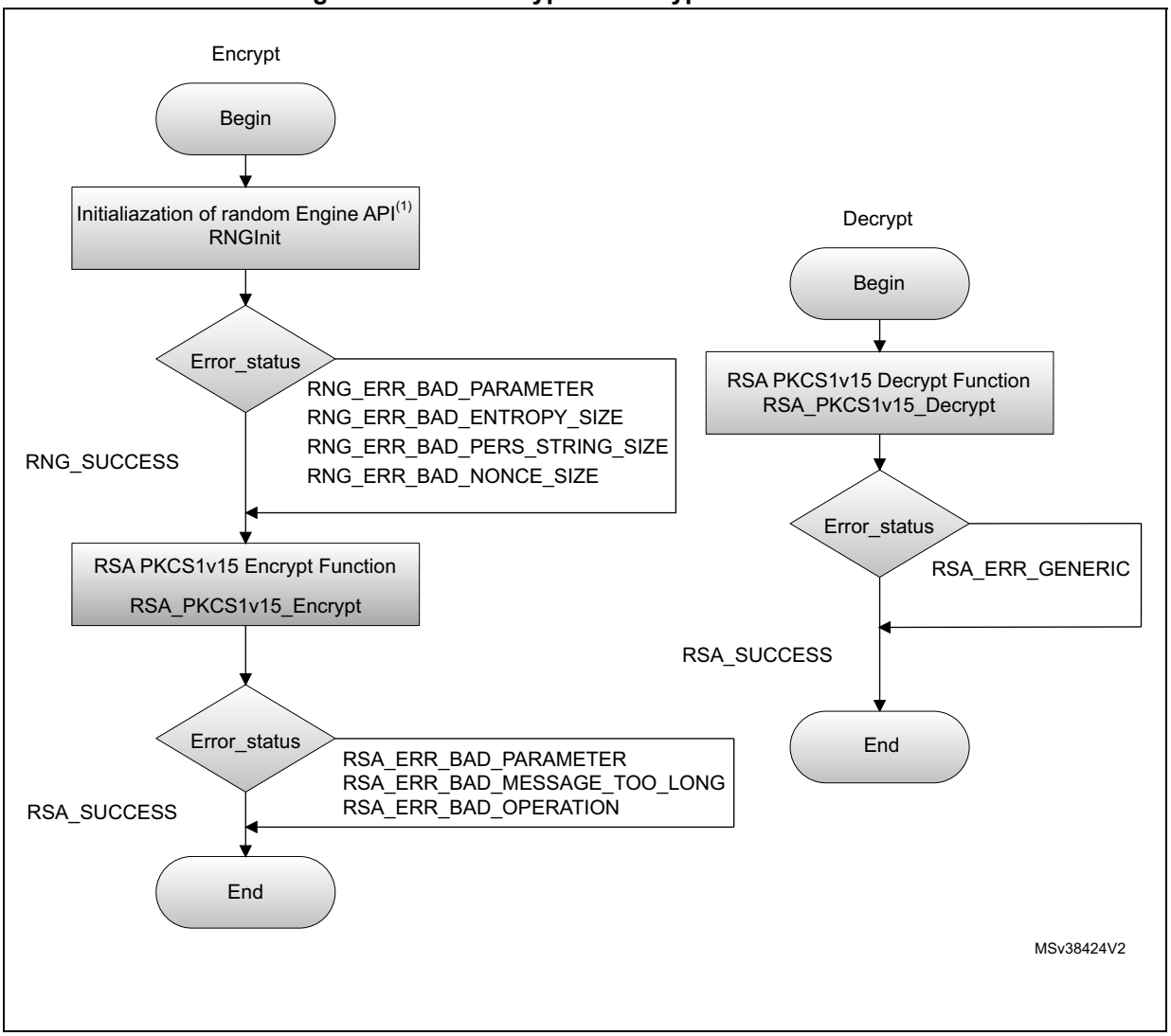


Figure 34. RSA Encryption/Decryption flowcharts



1. This API is used only on:
   - Firmware implementation
   - Hardware acceleration implementation on STM32L1 Series.

15.2.1 RSA_PKCS1v15_Sign function

Table 179. RSA_PKCS1v15_Sign function

Function name	RSA_PKCS1v15_Sign

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 is 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 is 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_MESSAGE_TOO_LONG The input is bigger than the modulus. – RSA_ERR_MODULUS_TOO_SHORT: RSA modulus too short for this hash type.

Note: 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 180. RSAprivKey_stt(1) 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

1. In case of using the hardware library this structure is "AccHw_RSAprivKey_stt"

membuf_stt data structure

Structure type definition for a pre-allocated memory buffer.

Table 181. 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

15.2.2 RSA_PKCS1v15_Verify function

Table 182. 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 is checked – [in] *P_pMemBuf Pointer to the membuf_stt structure that is 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 183. RSApubKey_stt(1) 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

1. In case of using the hardware library this structure is "AccHw_RSApubKey_stt"

15.2.3 RSA_PKCS1v15_Encrypt function

Table 184. RSA_PKCS1v15_Encrypt function

Function name	RSA_PKCS1v15_Encrypt function

Prototype	`int32_t RSA_PKCS1v15_Encrypt (`
	`const RSApubKey_stt *P_pPubKey,`
	`RSAinOut_stt *P_pInOut_st,`
	`RNGstate_stt *P_pRandomState,`
	`membuf_stt *P_pMemBuf)`

Behavior	Perform an RSA-PKCS#1 v1.5 Encryption using the public key
Parameter	– [in] *P_pPubKey The public key used to encrypt – [in] *P_pInOut_st The input data to encrypt – [in] *P_pRandomState The state of the random engine – [in] *P_pMemBuf Pointer to the membuf_stt structure that is used to store the internal values required by computation
Return value	– RSA_SUCCESS Operation successful. – RSA_ERR_BAD_PARAMETER if input parameters are not correct. – RSA_ERR_MESSAGE_TOO_LONG if input messagge to encrypt is too long (maxium size is Modulus Size - 11). – RSA_ERR_BAD_OPERATION if something was wrong in the RSA encryption or in the random number generation.

Note: P_pOutputData has to point to a memory area of suitable size (modulus size).

15.2.4 RSA_PKCS1v15_Decrypt function

Table 185. RSA_PKCS1v15_Decrypt function

Function name	RSA_PKCS1v15_Decrypt function

Prototype	`int32_t RSA_PKCS1v15_Decrypt (const RSAprivKey_stt *P_pPrivKey,`
	`RSAinOut_stt *P_pInOut_st,`
	`int32_t *P_pOutputSize,`
	`membuf_stt *P_pMemBuf)`

Behavior	Perform an RSA-PKCS#1 v1.5 Decryption using the private key
Parameter	– [in] *P_pPrivKey The private key used to decrypt – [in] *P_pInOut_st Structure keeping both inpunt, input size and pointer to output buffer – [out] *P_pOutputSize Pointer to the ouput decrypted data length [in] *P_pMemBuf Pointer to the membuf_stt structure that is used to store the internal values required by computation
Return value	– RSA_SUCCESS Operation successful. – RSA_ERR_GENERIC Generic decryption error.

Note: P_pInOut_st->mInputSize has be equal to the modulus size, and P_pInOut_st->pmOutput should be not smaller than modulus size - 11.

Care must be taken to ensure that an opponent cannot distinguish whether an error occurred, by error message or timing information, or he may be able to obtain useful information about the decryption of the ciphertext, leading to a strengthened version of Bleichenbacher's attack.

This function checks the padding in constant time, but nevertheless returns an RSA_ERR_GENERIC as the standard specifies, that must be handled with care by the caller.

15.3 RSA Signature generation/verification example

The following code give a simple example how to use RSA signature/validation of the legacy STM32 cryptographic library.

```
#include "main.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); }
}
```

16 Legacy STM32 cryptographic library settings

16.1 Configuration parameters

Table 187 describes the configuration parameters used to build the legacy STM32 cryptographic library. It is defined in the following files:

- “Middlewares\ST\STM32_Cryptographic\Inc\config.h” for the firmware package
- “Middlewares\ST\STM32_Crypto_AccHw\Inc\AccHw_config.h” for the hardware acceleration package

16.2 STM32_GetCryptoLibrarySettings

To get information about the legacy STM32 cryptographic firmware library settings and version, call the STM32_GetCryptoLibrarySettings() function in the application layer.

Table 186. STM32_GetCryptoLibrarySettings(1)

Function name	STM32_GetCryptoLibrarySettings
Prototype	`int32_t STM32_GetCryptoLibrarySettings(STM32CryptoLibVer_TypeDef * LibSettings)`
Behavior	Get the legacy STM32 cryptographic firmware library settings
Parameter	– [in,out] *STM32CryptoLibVer_TypeDef: Pointer to structure that is used to store the internal library settings
Return value	Do not care

1. In case of using the hardware library this function is "AccHw_STM32_GetCryptoLibrarySettings".

Table 187. STM32CryptoLibVer_TypeDef(1) data structure

Field name	Description	Default Build Settings
uint8_t X	Used to get the X parameter of the current legacy STM32 cryptographic library version	3
uint8_t Y	Used to get the Y parameter of the current legacy STM32 cryptographic library version	1
uint8_t Z	Used to get the Z parameter of the current legacy STM32 cryptographic library version	3, 4 or 5
uint8_t Type	Used to get the Type of the version. This parameter can be a value of @ref TypeConf	SW in the firmware library HW in the hardware library
uint8_t CortexConf	Used to get the Cortex®. This parameter can be a value within: – 0x1: of Cortex®-M0 – 0x2: Cortex®M0+ – 0x4:Cortex®-M3 – 0x8: Cortex®-M4 – 0x10: Cortex®-M7 – 0x20: Cortex®-M33	Depends on Cortex®
uint8_t IdeConf	Used to get the IDE used to compile the library. This parameter can be a value of @ref IdeConf	Depends on IDE used
uint8_t IdeOptimization	Used to get the compiler optimization. This parameter can be any combination of values of @ref IdeOptimization	Depends on Optimization used
uint8_t FpuConf	Used to get the FPU configuration used to compile the library. This parameter can be a value of @ref FpuConf	Depends on the FPU settings that are used
uint8_t EndiannessConf	Used to get the option value used to specify the memory representation of the platform. This parameter can be a value of @ref EndiannessConf	The option LITTLE ENDIAN is enabled
uint8_t MisalignedConf	Used to get if the CRL_CPU_SUPPORT_MISALIGNED is defined or not. This parameter can be a value of values @ref MisalignedConf	Depends on Cortex®
uint8_t EncDecConf	Used to get which functionalities of encryption and decryption functionalities are included. This parameter can be any combination of values of @ref EncDecConf	The operations of encryption and decryption are included
uint16_t SymKeyConf	Used to get the Symmetric algorithms included in the library. This parameter can be any combination of values of @ref SymKeyConf(1)	In firmware libraries all algorithms are included. For the algorithms included in each one of the hardware libraries, please refer to *Section 3.4.2: Legacy STM32 cryptographic hardware acceleration library*

Table 187. STM32CryptoLibVer_TypeDef(1) data structure (continued)

Field name	Description	Default Build Settings
uint16_t SymKeyModesConf	Used to get the Modes of Operations for Symmetric Key Algorithms included in the library. This parameter can be any combination of values of @ref SymKeyModesConf	In firmware libraries all modes operations are included. For the modes operations included in each one of the hardware libraries, please refer to *Section 3.4.2: Legacy STM32 cryptographic hardware acceleration library*
uint16_t AsymKeyConf	Used to get the Asymmetric algorithms included in the library. This parameter can be any combination of values of @ref AsymKeyConf	In firmware libraries all algorithms are included. For the algorithms included in each one of the hardware libraries, please refer to *Section 3.4.2: Legacy STM32 cryptographic hardware acceleration library*
uint16_t HashConf	Used to get the Hash and Hmac algorithms included in the library. This parameter can be any combination of values of @ref HashConf	In firmware libraries all algorithms are included. For the algorithms included in each one of the hardware libraries, please refer to *Section 3.4.2: Legacy STM32 cryptographic hardware acceleration library*
uint16_t MACConf	Used to get the MAC algorithms included in the library. This parameter can be any combination of values of @ref MACConf	In firmware libraries all algorithms are included. For the algorithms included in each one of the hardware libraries, please refer to *Section 3.4.2: Legacy STM32 cryptographic hardware acceleration library*
uint8_t DrbgConf	Used to get if the deterministic random bit generator is inculded. This parameter can be any combination of values of @ref DRBGConf	The options DRBG_AES128 and CRL_RANDOM_REQUIRE_RESEED are included in the firmware library and the hardware library for the STM32L1 Series
uint8_t AesConf	Used to get the AES algorithm version used. This parameter can be a value of @ref AESConf	In firmware libraries The value of CRL_AES_ALGORITHM selected is 2 In hardware libraries The value of CRL_AES_ALGORITHM selected is 1 for the high size optimization and 2 for the high speed optimization

Table 187. STM32CryptoLibVer_TypeDef(1) data structure (continued)

Field name	Description	Default Build Settings
uint8_t RsaConf	Used to get the RSA Window size selected. This parameter can be a value of @ref RSAConf	In firmware libraries The value of CRL_AES_ALGORITHM selected is 2 In hardware libraries The value of CRL_AES_ALGORITHM selected is 1 for the high size optimization and 2 for the high speed optimization
uint8_t GcmConf	Used to get the algorithm to be used for polynomial multiplication in AES-GCM. This parameter can be a value of @ref GCMConf	In firmware libraries The value of CRL_GFMUL selected is 2 In hardware libraries The value of CRL_GFMUL selected is 0 for the high size optimization and 2 for the high speed optimization If the AES-GCM is not supported by the library the value of CRL_GFMUL selectedis 0

1. In case of using the hardware library this structure is "AccHw_STM32CryptoLibVer_TypeDef "

17 FAQs

This section gathers some of the most frequent questions on legacy STM32 cryptographic library package and provides some solutions and tips.

Table 188. FAQs

No.	Question	Answer/solution
1	Since the library is delivered on binary format, if I use one cryptographic algorithm, will the other irrelevant resources of the library be included in the final application footprint?	No. The compiler only considers the external functions actually called in the application. As a result, irrelevant resources are not included in the final application footprint.
2	If I use many APIs for different algorithms, will the code size of my application be the sum of the code size of each API?	No. It is less than the sum of each API code size as they share some common firmware blocks.
3	The project is compiled without errors, but the encryption result is wrong when running the application.	Possible causes of the issue may be: – Stack size too low. – Heap size too low. – CRC disabled (only for the firmware library).
4	Are examples provided with the ready-to-use tool-set projects?	Yes. The legacy STM32 cryptographic firmware library package provides a rich set of examples (31 examples). They come with preconfigured projects for several chains: IAR™, Keil® and GCC.
5	If I change the options defined in the file "config.h" or "AccHw_config.h", will this have an impact on my project?	Yes, it may has an impact. In particular if you change options like "CRL_GFMUL" that modifies the interface structures.
6	Does legacy STM32 cryptographic library use the embedded cryptographic peripherals in STM32 products?	In the Fw_crypto sub-package all algorithm are based on firmware implementation without using the embedded cryptographic peripherals. In the AccHw_Crypto sub-package the embedded cryptographic peripherals is used to enhance algorithms benchmarks on dedicate devices for more details please refer to Section 3.4.2: legacy STM32 cryptographic hardware acceleration library.
7	Can I use both the cryptographic API from the HAL drivers and the legacy STM32 cryptographic firmware library in the same application?	Yes, you can, since there is no dependency between HAL drivers and legacy STM32 cryptographic library.
8	How to install patch version 3.1.1?	Download the patch package from STMicroelectronics website. Extract the patch in your local directory in the same path as the X-CRYPTOLIB package
9	How to get FPU library?	FPU library names are prefixed with ‘_FPU'.

18 Revision history

Table 189. Document revision history

Date	Revision	Changes
27-Aug-2015	1	Initial release.
16-Dec-2015	2	Updated: – *Introduction* in cover page. – *Section 3*, *Section 3.3*, *Section 3.4*, *Section 4.1*, *Section 4.2.1*, *Section 4.2.4*, *Section 4.3.1*, *Section 5.1*, *Section 6.1*, *Section 7.1*, *Section 8.1*, *Section 10.1*, *Section 11.1*, *Section 12.1*, *Section 13.1*, *Section 14.1*, *Section 15.1*, *Section 16.1* – *Figure 7*, *Figure 11*, *Figure 12*, *Figure 13*, *Figure 14*, *Figure 15*, *Figure 16*, *Figure 17*, *Figure 23*, *Figure 25*, *Figure 27*, *Figure 31*, *Figure 34* – *Table 10*, *Table 13*, *Table 16*, *Table 21*, *Table 24*, *Table 25*, *Table 26*, *Table 27*, *Table 27*, *Table 31*, *Table 33*, *Table 39*, *Table 40*, *Table 43*, *Table 47*, *Table 48*, *Table 49*, *Table 50*, *Table 51*, *Table 52*, *Table 53*, *Table 54*, *Table 57*, *Table 58*, *Table 59*, *Table 61*, *Table 62*, *Table 137*, *Table 160*, *Table 161*, *Table 164*, *Table 180*, *Table 183*, *Table 186*, *Table 187*, *Table 188* – Software version from V3.0.0 to V3.1.0. *Section 3.1*, *Section 3.3*, *Section 4.1*, *Section 5.1*, *Section 6.1*, *Section 7.1*, *Section 8.1*, *Section 9.1*, *Section 10.1*, *Section 11.1*, *Section 12.1*, *Section 13.1*, *Section 14.1Section 15.1* Added: – *Section 3.3.1*, *Section 3.3.2*, *Section 3.4.1*, *Section 3.4.2* – *Table 6*, *Table 12*, *Table 19*, *Table 37*, *Table 46*, *Table 56*, *Table 110*, *Table 130*, *Table 135*, *Table 139*, *Table 159*, *Table 168*, *Table 178* – Note 1 on *Figure 12*
19-May-2018	3	Updated *Table 3: Legacy STM32 cryptographic firmware libraries*, *Table 188: FAQs*
19-Jul-2018	4	Updated: *Section 12.2.7: HKDF_SHA512*, note on *Section 3.2: Architecture*, *Table 188: FAQs*

Table 189. Document revision history

Date	Revision	Changes
15-Jan-2020	5	In the whole document, replaced crypto by cryptographic or cryptography depending on the context, except when referring to X-CUBE-CRYPTOLIB. Minor English corrections. Added IAR trademark. Added STM32H7, STM32G0, STM32G4, STM32L5 and STM32WB Series in *Figure 6: Legacy STM32 cryptographic library architecture and implementation in a full project*. In *Section 3.4.1: Legacy STM32 cryptographic firmware library*, added Arm Cortex-M33 in the list of cores for which STM32 crypto library is complied, added CM33 and fpu options. Added STM32G0, STM32G4, STM32L5 and STM32WB Series in *Table 3: Legacy STM32 cryptographic firmware libraries*. Updated *Table 187: STM32CryptoLibVer_TypeDef data structure*. Updated parameters for *ED25519verify*. Added caution note in *Section 15.1: RSA description*. Modified FAQ number 9 related to FPU in *Table 188: FAQs*.
23-Nov-2020	6	Updated *Figure 6: Legacy STM32 cryptographic library architecture and implementation in a full project*, *Table 3: Legacy STM32 cryptographic firmware libraries* and *Table 188: FAQs*
14-Jan-2021	7	Updated *Table 3: Legacy STM32 cryptographic firmware libraries*
04-Jun-2021	8	Replaced RPN X-CUBE-CRYPTOLIB with X-CUBE-CRYPTO-V3 Replaced everywhere “STM32 cryptographic” with “legacy STM32 cryptographic”. Updated *Table 186: STM32_GetCryptoLibrarySettings* and *Table 187: STM32CryptoLibVer_TypeDef data structure*